When is Temporal Planning *Really* Temporal?

by

William Albemarle Cushing

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2012 by the
Graduate Supervisory Committee:

Subbarao Kambhampati, Chair
Chitta Baral
Hasan Davalcu
David E. Smith
Daniel S. Weld

ARIZONA STATE UNIVERSITY

December 2012

ABSTRACT

In this dissertation I develop a deep theory of temporal planning well-suited to analyzing, understanding, and improving the state of the art implementations (as of 2012). At face-value the work is strictly theoretical; nonetheless its impact is entirely real and practical. The easiest portion of that impact to highlight concerns the notable improvements to the format of the temporal fragment of the International Planning Competitions (IPCs). Particularly: the theory I expound upon here is the primary cause of—and justification for—the altered (i) selection of benchmark problems, and (ii) notion of "winning temporal planner".

For higher level motivation: *robotics*, *web service composition*, *industrial manufacturing*, *business process management*, *cybersecurity*, *space exploration*, *deep ocean exploration*, and *logistics* all benefit from applying domain-independent automated planning technique. Naturally, actually carrying out such case studies has much to offer. For example, we may extract the lesson that reasoning carefully about *deadlines* is rather crucial to planning in practice. More generally, effectively automating specifically *temporal* planning is well-motivated from applications. Entirely abstractly, the aim is to improve the theory of automated temporal planning by distilling from its practice.

My thesis is that the key feature of computational interest is *concurrency*. To support, I demonstrate by way of *compilation methods*, *worst-case counting arguments*, and analysis of algorithmic properties such as *completeness* that the more immediately pressing computational obstacles (facing would-be temporal generalizations of classical planning systems) can be dealt with in theoretically efficient manner. So more accurately the technical contribution here is to demonstrate: The computationally significant obstacle to automated temporal planning that *remains* is just *concurrency*.

For Cassie

ACKNOWLEDGEMENTS

This investigation into Temporal Planning begins in 2006 with an extensive and fascinating collaboration with Mausam and Daniel Weld. Their ideas have profoundly shaped its entire course. Two summers spent with David E. Smith and Jeremy Frank were invaluable towards obtaining real world perspective on the issues. J. Benton and I have been loosely collaborating on Temporal Planning for many years. Fellow student Kartik Talamadupula contributed no less in the way of years of valuable discussion. Particularly through discussion at ICAPS, Maria Fox, Derek Long, Hector Geffner, Jussi Rintanen, Brian Williams, Sylvie Thiebaux, Jörg Hoffmann, Dave Musliner, Robert Goldman, Mark Boddy, and Malte Helmert have all directly and profoundly impacted my work. Amanda Coles and Andrew Coles deserve tremendous credit for their year over year gains in both empirical effectiveness and temporal expressiveness—as do their competitors—after all, without measurable progress, what is there to analyze? Terry Zimmerman volunteered invaluable review of drafts. Likewise am I grateful to my entire committee for undertaking the task: Chitta Baral, Hasan Davalcu, David Smith, Daniel Weld, and Subbarao Kambhmpati all deserve credit and thanks (several times over). I am grateful that I got to join the community at this time in its growth. It is hard to over -thank all of the many organizers of the main conference, and the competitions, for the enormous investments into the basic infrastructure that power so much of our celebrated empirical gains.

It should be said once: I have written all herein. However, writing is nothing without its reader. Hence I shall write *we* throughout: last, and greatest, thank *you*.

TABLE OF CONTENTS

Page

LIST OF FIGURES x

CHAPTER

1 Introduction 1

1.1 Overview of the Dissertation 5

1.2 Summary 13

1.3 Organization of Background and Related Work 14

1.4 Related Work: Temporal Planning Systems 15

1.4.1 Sequential Planning 15

1.4.2 Conservative Temporal Planning Systems 16

1.4.3 Interleaved Temporal Planning Systems 17

1.5 Related Work: Temporal Planning Languages 20

1.6 Related Work: Temporal Planning Applications 25

1.6.1 The AI Effect in relation to Temporal Planning Research . . 31

2 Definitions and Notation for Variations on Temporal Planning 34

2.1 Sequential Planning Definitions 38

2.1.1 Intuition 38

2.1.2 Machinery: State Transition Functions 44

2.1.3 Plans, Executions, and Solutions 46

2.1.4 A STRIPS Reformulation of Action Definitions 47

2.2 Conservative Temporal Planning Definitions 50

2.2.1 Intuition 50

2.2.2 Machinery: Vault Transition Functions 55

2.2.2.1 Problem Structures: Durations, Deadlines, and Situations 56

CHAPTER Page

2.2.2.2 Interpretation Structures: Locks, Vaults, and Vault Transition Functions 58

2.2.3 Plans, Executions, and Solutions 62

2.3 Interleaved Temporal Planning Definitions 64

2.3.1 Intuition 65

2.3.2 Machinery: Debt Transition Functions 70

2.3.3 Plans, Executions, Solutions 74

2.3.3.1 Inducing Effect-Schedules from Action-Schedules 76

2.4 Summary 80

3 Foundational Theory for Temporal Planning 83

3.1 Sequential Planning Theory 91

3.1.1 Formal Reduction to a State Transition System 91

3.1.2 Landmark Analysis in Support of Forcing Compilations . . . 93

3.2 Conservative Temporal Planning Theory 102

3.2.1 Rescheduling: Dominance of Left-Shifting 105

3.2.1.1 Sketch of Left-Shifted Dominance 106

3.2.2 Reordering: Deordering and Behavior-Equivalence 112

3.2.2.1 Timelines, Behaviors, and Behavior-Equivalence . 116

3.2.2.2 (Not) Mutually Exclusive and Behavior-Preserving Reordering 120

3.2.2.3 Formal Proof of the Deordering Theorem 128

3.2.2.4 Significant Corollaries of Deordering/Behavior-Equivalence 128

3.2.3 Formal Reduction to a State Transtion System 131

3.2.4 Discussion and Summary 138

CHAPTER Page

3.3 Interleaved Temporal Planning 142
3.3.1 Rescheduling: Dominance of Slackless Schedules 142
3.3.2 Reordering: Deordering and Behavior-Equivalence 155
3.3.3 Formal Reduction to a State Transition System 159
3.3.3.1 Time is Discrete 162
3.3.3.2 PDDL-style Decomposition Is Simple 167
3.4 Discussion of Novelty and Significance 170
3.4.1 Parallelism versus Concurrency (TGP is suboptimal) 175
3.4.1.1 Intuition: Lecture Scheduling 175
3.4.1.2 Definitions 177
3.4.1.3 A Counterexample in Blocksworld to Optimality of TGP 179
3.4.1.4 Discussion: Pros/Cons 183
3.4.2 PDDL is Unsound 191
3.4.2.1 Our Philosophy 194
3.4.2.2 Intervals of Definition versus Extent 200
3.4.2.3 Selected Philosophy of Temporal PDDL 207
3.4.2.4 A Practical Shortcoming of Mixed Discrete-Continuous Planning 209
3.4.2.5 Selected Semantic Flaws of Temporal PDDL . . . 211
3.4.2.6 Counterexample to VAL-Soundness: The Bomb-Defusing Domain 213
3.4.2.7 Discussion: Explanation and Resolution of Unsoundness of VAL/PDDL 218
3.4.2.8 Conclusions on ITP versus PDDL 223

CHAPTER Page

4 Analysis of Expressiveness of Temporal Planning Languages 227
4.1 Definitions: Required Concurrency, Causally Compound, Sublanguages 230
4.2 Causally Compound Actions Characterize Required Concurrency . . 237
4.2.1 Brief Discussion of Figure 4.2 238
4.2.2 First Claim: Necessity 239
4.2.3 Second Claim: Sufficiency 241
4.3 Sequential Planning 'Supports' Optional Concurrency 248
4.4 Temporally Expressive Languages are Compilation-Equivalent . . . 253
4.4.1 Scope 256
4.4.2 Compilations 259
4.4.2.1 Compiling Out Primitive Transitions 262
4.4.2.2 Compiling Many Parts into Few 273
4.5 Conclusions upon the Analysis of Temporal Planning Languages . . 292
5 Worst-Case Analysis of Forward-Chaining Temporal Planners 296
5.1 First-Fit Classical Planners 300
5.1.1 Discussion 309
5.2 Decision Epoch Planners 313
5.2.1 Discussion of Figure 5.1 313
5.2.2 Setup 314
5.2.3 DE Planners are Incomplete and Nonsystematic 317
5.2.4 Formal Definition of Decision Epoch Planners as Search through Temporal Situations 323
5.2.5 Discussion 333
5.3 Temporally Lifted Planners 337

CHAPTER Page

5.3.1 Discussion of Figure 5.2 and Figure 5.3 342

5.3.1.1 Shallow Technical Remarks on Figure 5.2 342

5.3.1.2 True Ambition: Figure 5.3 343

5.3.2 Combining Deordering and Slacklessness 344

5.3.3 Technical Significance and Limitations 349

5.3.4 A Solution to Required Concurrency? 351

5.3.5 Wrapping Up 356

5.4 Summary 358

6 Conclusion 362

6.1 Key Lessons 364

6.2 Limitations 367

6.3 Review of the Dissertation 369

6.4 Summary 373

REFERENCES 374

APPENDIX

A Mathematics and Automated Planning Background 393

A.1 Mathematical Underpinnings 394

A.2 Domain-Independent Automated Planning Background 397

A.2.1 Syntax, Semantics, and State Transition Systems 397

A.2.2 Inference: Compilation, Reachability Heuristics, and Landmarks 401

A.2.2.1 Compilation 401

A.2.2.2 Unreachability Analysis 404

A.2.2.3 Landmark Analysis 408

APPENDIX Page

B Formal Proof of the Dominance of Left-Shifting 411

B.1 Faster is Locally Better 412

B.2 Locally Better is Globally Better 415

B.3 Fastest is Best 417

C The Deferred Case Analysis for Compiling to the Minimal Temporally Expressive Sublanguages of Interleaved Temporal Planning 421

LIST OF FIGURES

Figure Page

2.1 Sussman's Anomaly in single action schema BLOCKSWORLD. 39

2.2 The solution to Sussman's Anomaly. 39

2.3 The State Space of BLOCKSWORLDon Three Blocks 40

2.4 A conservative temporal encoding of (Single Handed) BLOCKSWORLD. . . . 51

2.5 A temporal encoding of Multi-Handed BLOCKSWORLD. 53

2.6 A solution to the Inverting Towers Problem of Figure 2.5. 53

2.7 A Fragment of CTP Situations. 54

2.8 Build the tower c-a-b, from the tower a-b-c. 65

2.9 The smallest solution to the problem of Figure 2.8. 65

2.10 A schedule equivalent to the plan in Figure 2.9? 66

2.11 An encoding of BLOCKSWORLD for interleaved action concurrency. 67

2.12 A path through ITP-situations 68

2.13 A variation on a motivating example for temporal PDDL. 69

3.1 Implementations reject solvability of this PDDL-syntax. 87

3.2 How to implement a left-shift by 1 bit in the ADL-fragment of PDDL. . . 165

3.3 How to implement $c := \max(a, b)$, see Figure 3.2. 166

3.4 Swap block a and block c. 179

3.5 An example of divergence between PDDL theory and practice. 191

4.1 Exploiting temporary availability naturally requires concurrency. 228

4.2 The taxonomy of temporal sublanguages and their expressiveness. . . . 236

4.3 Permitting causally compound actions permits required concurrency. . . 237

4.4 Conservative temporal planners can handle causally primitive actions. . 240

5.1 An abstract look at FFC and DE search-trees 312

5.2 An example TEMPO search tree for fixing a fuse 336

Figure Page

5.3 A reduced TEMPO search tree for fixing a fuse 342

5.4 How to uniquely associate labeled partial-orders with total-orders 345

A.1 A terrible way to compute the Fibonnaci numbers. 394

Chapter 1

Introduction

Over the past decade the computational performance of temporal planners has steadily and considerably improved [36, 36, 37, 39, 54, 56, 57, 64, 85, 87, 93, 120, 129, 142, 146, 184, 198, 201, 208]. These successes pivot, in no small part, upon the creation of a standard temporal planning language (PDDL) [71]; lacking a standard, empirical comparison is often "apples to oranges". Of equal note are the temporal tracks of the International Planning Competitions [60, 76, 81, 82, 105, 141]. In particular we may view the raw data as in-depth empirical evaluation. Finally, we would be remiss if failing to consider the role that classical planning research plays. Specifically, much of temporal planning research consists of no more—and no less—than generalizing the techniques of classical planners [16, 20, 29, 87, 103, 116, 174]. In short, temporal planning has made great strides forward in terms of computational performance.

As much as could be expected—for cramming a decade into a paragraph—that story is true enough. Indeed, all throughout, we must bear in mind that the greatest single truth is that temporal planning has come far. (There is, after all, nothing to be said in favor of critical analysis of the insignificant!) With that said though, let the critique commence. Two opposing attitudes upon temporal planning both go too far: theory is too pessimistic, and practice is too optimistic.

From a theoretical standpoint, providing nice guarantees upon reasoning about time (alongside action, change, and causality) is 'impossible'. For example, even guaranteeing merely *sound* conclusions is challenged from ancient times by the

paradoxes of Zeno of Elea [145]. Still today we have many competing mathematical formulations of time and causality, by which we may conclude that none are more than partially satisfactory; regarding automation the significant flaw is that most/all carry grim computational properties [2, 4, 70, 88, 109, 112, 151, 165, 173, 175]. Specifically temporal planning is typically regarded as grossly infeasible at best (if not downright *undecidable*) as it 'has' to deal with continuous time: Optop [151], ZENO [168], temporal PDDL [71], and PDDL+ [70] all espouse such perspective. Even without continuous time we still have daunting theoretical results such as 2EXPSPACE-completeness [148, 175].

Meanwhile, from a practical standpoint, it is abundantly clear that intelligent real-world behavior is frequently accomplished despite its many theoretical complexities (time, money, fuel, tools, uncertainty, sensing, communication, adversaries, ...). For example, dogs are fairly intelligent. To be sure, even the most real-world of *automated* systems are rather more limited than even just dogs with respect to general capability: Artificial Intelligence remains far from 'solved'. Nonetheless the highly specialized capabilities of existing applications of automated decision making in particular (and Artificial Intelligence in general) are already quite impressive and highly practical [1, 12, 37, 110, 137, 154, 168, 169, 183]. As far as specifically temporal planning systems go, real-world or no: the state of the art *seems* impressive enough, *i.e.*, as just cited and introduced.

The truth, of course, lies between 'solved' and 'impossible'. The meta-research issue is that temporal planning has been associated with many different facets of a planning problem, including *durative actions*, *deadlines*, *concurrency*, *processes*, *trajectory constraints*, and *continuous change*. Lacking any particular standard, researchers, naturally, cherry-picked their favorite features, giving rise to many fla-

vors of temporal planning. Which renders empirical comparison virtually meaningless. For analogy, consider “Alice got a B in History whereas Bob got an A in Geometry.”: precious little of interest follows, *i.e.*, it could also be, or not be, the case that “Alice got an A in Geomtery whereas Bob got a C in History.”.

The issue applies to *most* automated planning research: previous to 1998. At that time—the advent of the International Planning Competitions (IPCs)—the specification of PDDL (standing for Planning Domain Definition Language) was intended to, and did, resolve much of the issue [152]. Specifically the effort was a grand success with respect to so-called “Classical Planning”. Consider that, in context, we may indeed fairly say just “Classical Planning”. In contrast is its smallest name in broader context: “domain-independent automated single-agent deterministic completely-modeled fully-observable closed-world discrete sequential … planning”. That the smaller name *is* fair attests conformance to the PDDL standard, and so further attests significance of the empirical measurements. Said measurements themselves attest the concrete value of performing meaningful empirical evaluation [7, 60, 76, 82, 105, 141, 143]: the state-of-the-art in the computational performance of classical planners has come far [66, 174].

Unfortunately, as we showed in 2007 [43], despite the temporal generalization of PDDL *circa* 2002 [71], at least two formally distinct interpretations of “temporal planning” remain at large: thinly veiled classical planning, and planning that copes with *required concurrency*. To make matters worse (or ‘conveniently’ better), it is the ‘classical’ approaches that perform best on the given benchmarks. *Which means precious little:* We could instead just choose all the temporal benchmarks beyond their grasp. The value of an experiment whose outcome may be fixed in advance … is small at best.

So about the only reasonably true conclusion of interest to be drawn from the competition results, *i.e.*, without recourse to far deeper analysis, is as follows. The temporal competitions are 'rigged' in favor of a far simpler *alternative* to their official specification. In particular the empirical results heavily discriminate against any and all of the techniques implemented within the faithful entrants. So if we use those results to judge the quality of the research, then we shall be fooled into considering only the ideas of those who won chiefly by virtue of spiting the spirit of the rules. That would be in contrast to winning only by virtue of implementing truly —when evaluated fairly—more empirically effective temporal planning techniques.

So has temporal planning research really progressed far? There *could* very well have been substantial progress in performance. Indeed, the research itself is compelling enough on theoretical grounds. The existing empirical evidence, though, leaves very much to be desired. This is not easily addressed. The flaw is not in the experimental setup *per se*. Rather, the issue is that there are (at least) two camps both claiming the right to define "Temporal Planning": in logically contradictory—*and computationally meaningful*—fashion.

1.1 OVERVIEW OF THE DISSERTATION

The broad aim of the dissertation is to develop a deeper understanding of the relationship between temporal and classical planning. It grows out of our initial inquiry: a too short treatment of this complex topic [43]. Interestingly enough—*despite* its inevitable, and here fixed, flaws—the work already begins to have the desired impact [76, 105].

The Thesis. The thesis we pursue is that *concurrency* is "The Feature" best characterizing the computational relationship between the state-of-the-arts in Classical Planning and Temporal Planning. More specifically, we all *ought* to, and here do, sharply distinguish between three (rather than two) types of temporal planning (with classical planning the degenerate case):

- In Sequential Planning, concurrency is forbidden.
- In Conservative Temporal Planning, concurrency is (strictly) optional.
- In Interleaved Temporal Planning, concurrency is (potentially) required.

The justification—demonstrating so is our mission throughout—is that each is computationally more general than the preceding (so for pure theory), and each characterizes well enough the capabilities of increasingly-capable and particular subsets of state-of-the-art planners (so for a practical perspective).

Chapter 2. We begin the main matter, *i.e.*, starting in Chapter 2, with a formal account of these three kinds of temporal planning, along with their underlying intuitions.

Sequential Planning is just our particular notational spin on classical planning. The name emphasizes the form of plans: sequences of actions.

Extending that to Conservative Temporal Planning (CTP) corresponds to problems where some additional temporal information (action durations) can (only) be used to improve plan quality by better 'packing' the actions in time, *i.e.*, by (re)scheduling the sequences.

Finally, as far as the dissertation goes, extending to Interleaved Temporal Planning (ITP) allows for problems where concurrency of actions is causally necessary (rather than necessary merely for improving plan quality). The name refers to the specific mechanism: actions are permitted to be compound (*i.e.*, to consist of multiple smaller parts), and all such causally interacting parts must be scheduled disjointly in time (*i.e.*, they must be "interleaved").

Chapter 3. In Chapter 3 we continue by building up foundational theory for each kind of planning in turn. Our focus is on studying the relations between them by way of compiling/reducing all into a common, widely understood, form. Specifically we concern ourselves with three crucial intuitions of great consequence towards the implementation of effective temporal planners.

- It should be possible to ascribe clear semantics in terms of *reduction* to state transition systems. In nondegenerate forms of temporal planning the naïve formulations will be infinitely large, due to explicitly encoding values of time.

  It should also be possible to ascribe only slightly less clear semantics in terms of *finite* state transition systems. In other words it should be 'obvious' how to implement a correct and halfway-reasonable, *i.e.*, brute-force, planner.

- It should be possible to, with great flexibility, *reschedule* plans without effecting feasibility.

  It is this intuition that fuels the notion that brute-force temporal planning could even be possible. In contrast are forms of temporal planning beyond our scope; for example, in motor-control problems, such as the *cart-pole balancing problem*, precise timing decisions have gross effect upon whether plans actually work.

- It should be possible to freely *reorder* a planner's thoughts concerning locally independent actions.

  For example, the order that a planner chooses to decide upon whether to include each of the activities "Alice feeds a cat in Boston." and "Bob drives to work in New York." into a plan ought to be provably insignificant. It is this intuition that fuels the notion that deeper research from the planning perspective is worthwhile: cleverer-than-brute-force approaches ought to exist. In contrast are pure-search (a.k.a. puzzle-type) problems, where 'merely brute-force' is far cleverer than it sounds [108, 130].

With two forms of nondegenerate temporal planning and three intuitions each: that is six important theorems we shall formulate and prove. Three are key.

Our formalization of reduction for CTP, Theorem 3.17, establishes that Conservative Temporal Planning Problems and Sequential Planning Problems both reduce into the *same* underyling state-space. As far as practice goes, the result amounts to saying that classical planners are 'already' general enough to support CTP. The theoretically significant difference will concern finding *fastest* plans (*i.e.*, plans of minimum duration, a.k.a. makespan-optimal plans): ultimately we expect that

generalizing to Multi-Objective Search, rather than Single-Objective Search as for cheapest plans, will perform best. We shall take that as excuse enough to sharply distinguish Conservative Temporal Planning from Sequential Planning.

For comparison, our reduction for ITP, Theorem 3.22, turns out nowhere near as computationally promising—downright ugly does justice—as for Conservative Temporal Planning. We shall take that as (yet greater) excuse to sharply distinguish Interleaved Temporal Planning from both of its special cases.

While that angle turns out too ugly to be directly practical, there is still plenty of reason to suppose that reasonable approaches *do* exist for Interleaved Temporal Planning; it is only that the state-space perspective happens to be ill-suited to effective temporal reasoning. Among other reasons to remain hopeful—such as the existence of a number of apparently effective, practical, implementations of temporal planning—shall be our formulation and proof of reordering for ITP, Theorem 3.21. Namely the theorem takes on the lion's share of automatically abstracting to CTP whenever those worst-case issues of theoretical significance just so happen to be 'coincidentally' absent.[1] More specifically note that, with respect to the thesis (which stated here reads as roughly: "ITP is more challenging than CTP due to all and only required concurrency."), the technical significance goes toward automatically exploiting absences of required concurrency.

Then up through Chapter 3 we shall have: (i) reimagined the specifications of several forms of state-of-the-art temporal planning, (ii) armed ourselves with a

[1]A deep lesson that emerges from the practice of AI is that real-world problems only *look* tough —*i.e.*, the suggested worst-cases are daunting, but actual inputs overwhelmingly test just 'best'-case performance—or concisely, theory is almost always too pessimistic to be taken literally. The insight crops up over and over again (do we ever fail to make the point?); a personal favorite is the *phase transition* at 4 clauses to 3 propositions of random 3-SAT problems [181]. Closer to home, we may study the *why* behind the impressive performance of the planner FF on classical benchmarks [114].

deep understanding of several of their key theoretical properties, and in so doing (iii) identified workable approaches to all but our most general case. Yet shorter: we shall have constructed an elaborate theory of temporal planning. Next we will turn our attention to evaluating the accuracy and utility of the theory as applied to the practice of temporal planning. We will do so by analyzing (selected fragments of) the underlying formal *languages* and *algorithms*.

Chapter 4. We shall start off the language analysis by noting that every so-called implementation of the standard is, technically, distinct. Such differences all, ultimately, matter; but some are clearly far more significant than others, and many are safely ignored until staring at code itself. For an obvious example, *continuous change* is, hands-down, significant at 'any' level of abstraction. In contrast (and non-obviously), *deadlines* can typically be ignored by satisficing temporal planners. At least, that sacrifice, thus far, seems to be empirically tolerable. For a clearly insignificant feature: expressibility of *negative preconditions* is widely recognized as inconsequential.

With respect to temporal entrants of the IPCs: *almost* all points of distinction are rather more interesting to ignore than they are to pay attention to. Roughly, we shall prove that claim. In particular we shall characterize which temporal planners can, and which cannot, understand required concurrency by examining how each reinterprets the common syntax. Moreover we shall show that on either side of the expressiveness divide all remaining points of distinction may be 'dealt with' by various compilation techniques. Then in the end we shall be able to conclude that: Taking Conservative Temporal Planning and Interleaved Temporal Planning as representatives of the temporal state-of-the-art is reasonable enough. (The so-

called “classical planning” literature already establishes that a single name *is indeed* reasonable enough for the degenerate case: we need not reprove similar statements for Sequential Planning.)

Note the several potential meta-applications (design of future IPCs and revision of temporal PDDL are the top two).

Chapter 5. Completing the main matter, we will evaluate several existing temporal planning algorithms from our point of view, *i.e.*, as if the dissertation was specification. Specifically we shall examine the ‘state-space’ approaches, which, we note, presently dominate empirically. Of the ideas that have been tried we call attention to three, which we dub: First-Fit Classical Planner (FFC Planner), Decision Epoch Planner (DE Planner), and Temporally Lifted Planner (TEMPO Planner) [6, 36, 54, 57, 64, 87]. The theoretical properties to-be-evaluated are completeness and systematicity. Roughly: Does the approach consider every plan (i) at least once, and (ii) at most once? These are highly valuable properties. For example, in practice, we may deliberately sacrifice both as fuel towards improving computational performance (consider respectively: *local search* and *macro execution*).

To begin we will demonstrate that Conservative Temporal Planning poses relatively little difficulty to algorithm designers; albeit there is one challenge of note. In particular we shall see that approaches that rely on rescheduling the plans produced by their classical planning cores (*cf.* MIPS-CONSERVATIVE [57] and MIPS [58]), *i.e.*, all FFC planners, are complete and systematic for Conservative Temporal Planning—*sans* deadlines. So in terms of implementation cost and computational performance such systems are barely distinguishable from their classical planning cores.

With deadlines, though, things become more interesting. As we shall have proven in Chapter 3, completeness for deadlines, which is equivalent to attaining duration-optimality, calls for generalizing to Multi-Objective Search. The smallest implementation cost is negligible. Specifically, as we shall have already proven, it suffices to turn off duplicate state elimination: the smallest implementation cost is a few engineer-minutes. However, we should expect the computational consequences of such a simple-minded approach to be large, perhaps intolerably so. (Because the 'right' way is to *actually* generalize, from Single-Objective Search, to Multi-Objective Search.) We cannot draw any firm conclusions, for there is no empirical ground to stand upon, *i.e.*, more or less: the PDDL standard excludes the expression of deadlines, and only one implementation (CPT) anyways shoots for duration -optimality [202]. We can and shall, though, close the treatment with a rather more theoretically reasonable, *i.e.*, promising, approach: Theorem 5.4.

Then stepping beyond CTP, the difficulty that algorithm designers face is attaining completeness for ITP, yet retaining the ability to leverage the recent advances in classical planning technique. The difficulty is due to the emphasis on state-space: as we will have seen in Chapter 3, that perspective is ill-suited to temporal reasoning. For contrast, the plan-space perspective is well-suited to temporal reasoning [12, 84, 142, 146, 188, 208].

We shall see that the "key trick" is to twist our perspective from "state-space" to "forward-chaining". More specifically, both DE planners and TEMPO planners have been advanced as attractive solutions: inasmuch as both rely on forward-chaining they achieve the desire to leverage much of recent classical planning technique.

However, (once more [43, 147]) we shall see that DE planners leave much to be desired: these are neither complete, Theorem 5.6, nor systematic, Theorem 5.10!

(At the time, the incompleteness result contradicted widely held beliefs, and published proofs, about the nature of the design.) We shall see that the reason for the failings is that such planners rely on *eagerly* making all scheduling decisions.

To *lazily* make scheduling decisions, we may instead apply constraint reasoning technique. Doing so is harder to implement, but still straightforward enough to be taken seriously. In particular we shall prove that TEMPO planners are complete and systematic, Theorem 5.13. In fact we shall go further by applying our Chapter 3 result regarding reordering, Theorem 3.21. Namely, we shall demonstrate the heart of a method for having TEMPO planners automatically simplify to FFC-style planning 'when possible', Theorem 5.16. Specifically we shall be aiming at improving performance when required concurrency could have—but coincidentally will not have —appeared. So given our overarching theory: we will end our technical content with, as desired, a theoretically reasonable approach towards effective Interleaved Temporal Planning in practice.

1.2 SUMMARY

A decade of practice has ensued since the introduction of a standard for temporal planning. The experience makes clear that it is high time for a revision to the standard. In particular, two generalizations from classical planning to temporal planning have proven to be too distinct from one another, and too important in their own right, to be lumped together. We shall investigate the technical issues, ultimately answering the *whats* and *whys* for revision. Grossly over-simplifying—use at your own risk—said reasons are all ‘the same’:

Concurrency is “The Feature”.

1.3 ORGANIZATION OF BACKGROUND AND RELATED WORK

Section 1.4 is a brief survey of *temporal planning systems*. Roughly, no two speak the same formal language: Section 1.5 surveys the various proposals for standardizing *temporal planning language*. Normally we shall restrict our attention to the planning languages, domains, and systems of the IPCs. Section 1.6 takes a much broader view in discussion of *temporal planning applications*. Specifically we take pains to respect the significant gap between real applications of temporal planning, which are domain-dependent, and the domain-independent systems of temporal planning research.

The *uninitiated* in domain-independent automated planning literature are especially referred to our discussion of its applications (*i.e.*, Section 1.6).

Appendix A covers technical background: *mathematical notation* and selected *automated planning techniques*. Some aspects of the treatment here are notably unconventional. For example, the notation reflects a *strong* bias favoring the *functional programming paradigm*, *i.e.*, to the point of taking <u>sets</u> to be mathematically nonprimitive.[2]

[2] *More* abstractly, the convention is: if it is written down, then, with rare exception, its existence is decidable and its identity computable.

1.4 RELATED WORK: TEMPORAL PLANNING SYSTEMS

We categorize existing systems into one of four levels of temporal expressiveness, the last level being a catch-all for *everything* beyond our scope.

1.4.1 Sequential Planning

There are many icons of classical planning, noncomprehensively:

- STRIPS [66] is remembered for its input language (and sometimes for Triangle Tables). It defines the paradigm.
- SNLP [149] adds nice theory to Partial-Order Causal-Link Planning. Specifically, SNLP implements partially-ordered plans as an equivalence reduction on totally-ordered plans. We shall apply that insight as well.
- SHOP [155, 156] is a (simple-)Heirarchical Task Network Planner. It is regarded as domain-dependent, as it relies on the heirarchies for search control. It is perhaps better regarded as a qualitative kind of Interleaved Temporal Planner.
- TLplan [6] applies Linear Temporal Logic for search control.
- GraphPlan [16] defines the eponymous Planning Graph. The level heuristics with and without mutexes are the instantiations of $h^m(\cdot)$ for $m = 2$ and $m = 1$.
- BlackBox [128] reduces Planning Graphs to Boolean Satisfiability.
- GPCSP [52] reduces Planning Graphs to Constraint Satisfaction Programming.

- OPTIPLAN [196] reduces Planning/Causal Graphs to Integer Linear Programming.
- LPG-SEQUENTIAL [86] (our name) applies Stochastic Local Search to Planning Graphs.
- FF [114, 116] derives a heuristic from Planning Graphs to guide State-Space Search (a.k.a. forward-chaining). This, very effective, heuristic is called the Relaxed Plan Heuristic; FF has many derivatives.
- HSP [20] applies $h^2(\cdot)$ to guide Regression Search (also forward-chaining).
- ALTALT [161] applies various Planning Graph Heuristics to guide Regression.
- MIPS [58] applies Symbolic Search (a.k.a. Inference) to State-Space.
- GAMER [130] extends MIPS, and is 2008 optimal state-of-the-art. It is noted for eschewing admissible heuristics; these are, counter-intuitively, empirically ineffective on top of its symbolic search and 'endgame database'.
- LAMA [174] adds landmarks to FD, and is 2008 satisficing state-of-the-art.
- FD [103] is to SAS [9] and the Causal Graph what FF is to STRIPS and the Planning Graph. Extensions are 2011 state-of-the-art, optimal or satisficing.

1.4.2 Conservative Temporal Planning Systems

Syntactically, CTP-systems generalize by permitting action durations. Examples:

- TGP [188] appears to define the paradigm. It generalizes GRAPHPLAN.
- TPSYS [77] generalizes the Planning Graph of GRAPHPLAN to time in a different manner than TGP. The result is reminiscent of SAPA's heuristic (below).

- CPT [202] applies/generalizes a Partial-Order Causal-Link perspective alongside Constraint Satisfaction Programming technique.

- TP4 [100] applies/generalizes Regression Search, guided by $h^2(\cdot)$.

- HSP_a^* [96] extends TP4 by solving fragments of $h^m(\cdot)$ for $m > 2$ prior to search.

- SGPlan [29] applies problem-decomposition and meta-planning. Caution is required in interpreting its empirical results: it is typically regarded as a domain-dependent planner given the nature of its preprocessingcode [105].

- The baseline in 2008 [105] is 'FF', more accurately MetricFF [115], along with post-scheduling by First-Fit.

- YAHSP2 [201] and daeyahsp2 [56] are 2011 state-of-the-art. The first applies First-Fit to extend the classical planning core; the second further wraps that with genetic programming techniques to find even better schedules.

- LPG-td [84, 87] extends LPG by applying First-Fit.

- MIPS-conservative [57] (our name) extends MIPS [58] by applying First-Fit.

- The authors of TFD, see below, place TFD in this category.

1.4.3 Interleaved Temporal Planning Systems

Syntactically, ITP-systems permit access to subintervals. Examples:

- VAL [119] is the official plan validator for temporal PDDL.

- LPGP [142] is the first domain-independent complete approach to temporal PDDL. It applies reduction to Linear Programming by way of a novel temporal generalization of Planning Graphs.

- TLP-GP [146], similarly to LPGP, reduces to Disjunctive Temporal Networks.
- TM-LPSAT [184], similarly to LPGP, reduces to LPSAT [205].
- TLplan [6] introduces a viable approach (Decision Epochs) to forward-chaining for temporally expressive planning. Using Linear Temporal Logic for search control, its domain-dependent performance is quite impressive.
- TALplanner [135] is broadly similar to, and by report outperforms, TLplan. It uses a different (non-modal) form of temporal logic.
- SAPA [54, 198], building on FF [116], applies the Decision Epoch idea from Bacchus and Ady in TLplan [6] to domain-independent temporal planning. Its contribution is a particular form of (Relaxed, Metric) Temporal Planning Graph.
- Sharaabi [129] is a fork of SAPA aiming at greater expressiveness regarding invariants. Notably, it branches on unit time (also Decision Epochs).
- TFD [64] is to FD what SAPA is to FF.
- LMTD [120] extends TFD with better exploit of landmarks.
- LPG-interleaved [85] (our name) extends LPG-td [84], using the LPGP [142] approach to generalizing the Planning Graph to interleaved concurrency. By virtue of inheriting stochastic local search, it obtains noteworthy empirical performance.
- VHPOP [208] applies Partial-Order Causal-Link technique. It is notable for being a convenient complete implementation.

- CRIKEY [93] is likely the first complete forward-chaining approach to temporal PDDL. It applies temporal lifting to delay scheduling decisions into reasoning about Simple Temporal Networks.

- POPF2 [36] is the 2011 extension of CRIKEY [93], notable for applying the insight of SNLP [149] in converse. (It uses theory of partial-orders to reduce search effort.) For interleaved concurrency, it is 2011 state-of-the-art.

We shall not study such systems, but it is worth keeping in mind that there is no particular ceiling for temporal expressiveness. That is, quite a few systems go beyond mere interleaving, some, including some of the above, go *well* beyond; examples include EUROPA [12], HSTS [121], ASPEN [32], IxTeT [90], Deviser [200], FORBIN [46], O-Plan [40, 193], CIRCA [154], UPMurphi [169], Kongming [137], PROTTLE [140], Optop [151], COLIN [37], and ZENO [168].

1.5 RELATED WORK: TEMPORAL PLANNING LANGUAGES

It is difficult to overstate the significance of language/interface design and standardization [50, 51]. The following reviews the various proposed formalisms towards automated planning.

PDDL 1.2 [152] formalizes syntax for a certain interpretation of "classical planning". Features include the following.

- All of ADL [166] is subsumed (so also STRIPS [66]). For example:
  - *parameterized propositions+actions*:

    ```
    (:predicates (loc ?x ?l) ...),
    ```

    ```
    (:action move :parameters (?s ?d) :precondition ... :effect ...),
    ```

  - *quantified boolean formula* as preconditions:

    ```
    :precondition (and (exists (?h) (and (available ?h) (hand ?h))) ...), and
    ```

  - *quantified+conditional effects* are all supported:

    ```
    :effect (and (forall (?x) (when (in ?x B)
                      (and (not (loc ?x ?s)) (loc ?x ?d))))
             ...).
    ```

- The ADL subset of PDDL 1.2 is what later versions actually built upon [5].

  For curiosity, the features dropped (some are reinvented) by later versions are: *numeric fluents*, *domain axioms*, *maintennance constraints*, *object creation+destruction*, *open worlds*, *action decompositions* (HTNs), *search control advice*, and *series+parallel length* as notions of plan quality.

PDDL 2.1 [71] extends the ADL subset with:

- *numeric fluents*: `(:functions (distance ?s ?d - city) (speed ?p - plane) ...),`

- *durative actions*:

```
(:durative-action fly :parameters (?p - plane ?s ?d - city)
 :duration (= ?duration (/ (distance ?s ?d) (speed ?p)))
 :condition (and (over all ...) (at start ...) (at end ...))
 :effect (and ⟨ContinuousEffects⟩ (at start ...) (at end ...))),
```

  which *decompose* (with subtleties) into all and only:

  - `(over all ...)` invariants and continuous effects,
  - an `(at start ...)` ADL-primitive, and
  - an `(at end ...)` ADL-primitive,

- *continuous effects*: `:effect (and (decrease (fuel ?p) (* #t (fuel-burn-rate ?p))) ...)`,

- *durations as (numeric) parameters*:

```
(:durative-action refuel :parameters (?p - plane ?s - city)
 :duration (and )
 :condition (and (at start (and (> ?duration 0)
   (<= ?duration (/ (- (fuel-capacity ?p) (fuel ?p)) (fuel-fill-rate ?p)))
   (<= ?duration (/ (fuel ?s) (fuel-fill-rate ?p)))
   ...)) ...)
 :effect ...),
```

  and

- *flexible plan quality metrics*:

```
(:metric (minimize (+ (* total-time (money-over-time)) total-cost))).
```

In practice the most relevant sublanguages are Level 2 (numeric fluents) and Level 3 (durative actions and numeric fluents). That is more or less because every more expressive planner supports different and differing formalisms. Our scope stops here as well, *i.e.*, at Level 3, so we (mostly) omit further examples of syntax.

PDDL 2.2 [59, 194] adds: *timed initial literals*, and *derived predicates*. Respectively these are the poor man's form of *exogenous events* and *domain axioms*.

PDDL+ [70] recasts the more expressive levels of PDDL 2.1 into a form better suited to reduction to Hybrid Automata [111]. The additional syntax concerns: *processes* and *events*.

OPT [151] also supports *processes* and *events*. *Action decompositions*, as in HTN planning, are supported as well. Its semantics are based in Nonstandard Analysis [179]. Due to which, a key difference from PDDL+ is that *infinitesimal* is a formally defined concept in OPT.

PDDL 3.0 [82] adds: *preferences* (*i.e.*, *soft constraints*), and *trajectory constraints* (in a woefully impoverished, but quite useful, form of Linear Temporal Logic).

PDDL 3.1 [9, 79] adds *object-valued fluents*, as in `(= (loc spirit) waypoint-alpha)`. The semantics are clear enough in simple cases, and may be compiled out by introducing additional parameters. Precisely defining the meaning in all cases is less straightforward. For example, consider a nested use, say `(>= (sun-visibility (loc ?rover)) ...)`, within some larger quantified formula: adding myriad parameters would be less than attractive.

It is also interesting to review systems/literature that implicitly/explicitly take a stand on language:

- Smith and Weld, in defining TGP [188], implicitly define the natural extension (Durative-STRIPS) of STRIPS to *nonuniform action durations*. We could surmise that (1) Durative-STRIPS ought to be *considered important*, and (2) STRIPS ought to be considered as insisting upon the *uniform+unit case*.

- ZENO [168], TALPLANNER [135], Allen [3], and IXTET [90] all prefer particular —each differing—flavors of *Temporal Logic* for defining syntax and, espe-

cially, semantics. The differences between them are irrelevant; the relevance is that all are *far more expressive* than the ceiling we impose.

- ASPEN [32], and ANML [187] following it, propose ALGOL/C/block-structured syntax. For semantics, perhaps the most notable features are (1) access to *arbitrary subintervals* of action executions, (2) access *beyond* intervals of action executions, and (3) support for *action decomposition* as 'syntactic sugar' standing for complex temporal relationships (*i.e.*, reduce to (1) and (2)).

- HSTS [121] and EUROPA [12] propose treating *actions* and *fluents* as indistinguishable for temporal planning purposes. There are compelling arguments to be made in favor. Nonetheless, it seems that in practice more is lost than is gained. In particular, that language (*i.e.*, NDDL) fails to build in *persistence* of fluents, meaning that the language formalizes the 'worst' solution (write the axiom down explicitly) to the Frame Problem. At a more practical level, the choice makes applying concepts such as reachability heuristics difficult.

An especially notable feature of real world applications is the support of *resources*. That is, *time* and *resources* appear to go hand-in-hand in the real world:

- Geffner is hardly alone in taking the lack of resources in PDDL as a flaw [80].

- In favor of explicitly supporting all those variations on resource usage that *limit concurrency/parallelism* are: IPP [132], Rintanen and Jungholt [177], TP4 [100], PARCPLAN [62], SIPE [204], METRICFF [115], and GRT [171, 172].

- Proposals for building in complex theories of resources—particularly those supporting *temporary surplus*—into the planning language include: Cesta and Stella [26], O-Plan [55], ZENO [168], TALplanner [134], and IxTeT [90].

1.6 RELATED WORK: TEMPORAL PLANNING APPLICATIONS

Every planner is a temporal planner.

Hence: every application of planning is an application of temporal planning. The question to ask is, then: To what degree? It is important to divide that in two:

- What degree of temporal reasoning does the application itself call for?
- How responsible for temporal capabilities is the planning *component*?

The importance is firstly because the first answer is *overwhelmingly* that the real world requires proficiency in temporal reasoning. Secondly, to be perfectly blunt, the majority of real applications are quite successful *without* applying 'significantly temporal' reasoning at the highest levels of deliberation (*i.e.*, at the level of planning). Then for our purpose it is enough to glean just that the following research question is well-motivated from real world applications.

What are the *least* forms of Temporal Planning?

We get ahead of ourselves, however. Let us examine several of these alluded to real (or close enough) applications of planning:

- NASA stretches the meaning of *remote control* to outer space: automated tools for supporting decision-making and execution-time control are *essential*. The research from specifically the automated planning community sees some application in *planetary robotic science (rovers)* [1], *ground-based astronomy*, *satellite-based astronomy* [186], *solar-system scale networking*, *deep-space probes* [185], *general space-station operations*, and *human factors in space-station operations*.

- *Underwater robotic exploration/science* at the Monterey Bay Aquarium Research Institute (MBARI) benefits from the ability to withstand pressures too expensive to reinforce humans against. The downsides are many, mitigating which calls for the application of—and motivates deeper research in—Artificial Intelligence technique. It is, for example, surprisingly easy to get confused about which way is up, currents can drag one far off-course, and currents take obstacle-avoidance to a whole new level of technical challenge. Solutions that work on land (or air, or vacuum) to similar problems fail to cross mediums; the Global Positioning System, for example, is not effective even at shallow depth (simply because radio waves reflect, refract, and scatter at the air+water boundary). Decidedly *temporal* forms of planning prove useful [37, 138].

- Let us not dwell *too* long upon the frightening but inevitable subject of applying Robotics and Artificial Intelligence to *warfare*. Whether we say *Department of Defense*, *Department of Homeland Security*, or, more aggressively, *Army/Navy/Airforce* does not alter either the *necessity* or *difficulty* of automating reasoning in battles of chiefly wit at life-or-death stakes. Recall that the mere *possibility* of obtaining a technological advantage immediately and always becomes a *necessity*. Specifically the ability to mass-produce cheap and effective 'brain-power', given the extreme lethality of our weapons once applied, is precisely such possibility-turned-necessity.

  Let us first look at motivation from warfare to temporal planning. For the sake of abstractness, consider that war puts in sharp relief the necessity of

significant communication and coordination capabilities. From the *single-agent planning perspective*:

- Established *commitments* to assisting team-members look like *temporally extended goals* (so consider *soft trajectory constraints*: PDDL 3.0).
- The flip-sides of such contracts are to *receive* the assistance, such look like *exogenous events* when projecting down to a single-agent perspective (so consider *timed initial literals*: PDDL 2.2).
- Naturally: *commitment* follows *negotiation* follows generation of *reasonable proposals*. Reasoning ahead of time about potential *coordination opportunities* can be approached from the angle of *conditional commitments*. It may help to think of subtracting the *exogenous* out of "exogenous events" and "temporally extended goals". That is, the reasoning problem will feature: choices shall impose events and constraints in complex—*i.e.*, *temporal*—fashion. So consider that the commitment to finish a *durative action*, once started, models the general case. In other words, consider that an under-appreciated fact about PDDL 2.1 is that it supports (barely, and cryptically, to be sure, but supports nonetheless) the single-agent side of *proposing* and later *committing* to *coordination opportunities*.

Then for the sake of concreteness, consider a soldier planning a route back to base. Whether human or robotic, the soldier should be aware that planning to travel through dangerous terrain involves considering the option, and cost, of setting up *covering fire*. Furthermore the soldier should assess any competing longer—but safer—routes. Which route is best will naturally be highly

situation-dependent (hence the importance of explicit modeling and deliberation). (Incidentally, the greatest challenge in practice for this scenario is called *situational awareness*: actually solving the model once obtained is, at present time, much better understood.) Specifically this scenario—reasoning about the advantage of scheduling covering fire over a specific subinterval of a longer navigation activity—modeled abstractly enough (which is easily argued as *better* motivated than a detailed model), is the *same up to renaming of symbols*[3] as the less-obviously-useful scenario depicted in Figure 2.13.

So now let us look to warfare from planning. At present time—but it is, indeed, only a matter of time—there are reasonably few *real* and *direct* applications of automated planning to warfare. (There are, though, plenty of systems for such purposes as the training of soldiers.) The most directly relevant (close enough to real) application is Boddy's implementation of (what is called the *red team* side of, or in that context, the *black hat* side of) *cyber-security* [19]. Notably, Boddy heavily exploits FF [116].

Let that be enough consideration of automation towards the traditional science fiction nightmare.

- *Shipping* (*i.e.*, sending cargo by boat) poses many difficult, large-scale, optimization subproblems. These are ripe for (semi-)automation: decisions have impact in the millions of dollars, and there are too many for humans to manage effectively. The traditionally best approaches to automation in *supply chain management* are from the *scheduling*, or *Operations Research*, side of the fence. Recent work on specifically the *Fleet Repositioning* subproblem

[3] 'Domain-independent' is impossible to nail down, but one feature *widely* agreed upon is this: A domain-independent planner performs identically when symbols are arbitrarily renamed.

demonstrates notable success from the temporal planning side [195]. The cited work is especially interesting at the technical level because it applies both extremes and a hybrid middle-ground (which, as could be expected, performs best empirically).

- It is natural to dismiss *games*, particularly *videogames*, as frivolous. So let us be perfectly clear: *Videogames are serious business*. The videogame industry is a *multibillion* dollar industry with worldwide impact.

  As with any form of entertainment, customers are *demanding*: they take their money elsewhere if the product is (noticeably) flawed. The upshot is that videogame software is held to the *highest standards* (of quality, reliability, functionality, *etc.*).[4] In turn, such fact means: The application area of videogames has been, and continues to be, pushing the forefront of technology in each and every subdiscipline of applied Computer Science (not the sole force, mind, but a strong one). For example, nowadays, much of the challenge revolves upon networking and database issues, *i.e.*, supporting massively—3 million concurrent users, for example—multiplayer games. For another, a decade past, the leaps and bounds in the areas of *Visualization*, *Image Processing*, and *Graphics* (in general) were fueled in largest part by the industry.

  We in particular should, because it is the 'negative example', consider the videogame industry. That is, while "Applied Artificial Intelligence" certainly exists in spades in videogames, such is largely separate from the research

[4]The only software arguably better tested/verified would be tiny pieces of functionality in the area of, for example, aviation. Of course the quality bound for entertainment is lower than for preservation of human life: but not by much. (The question is not objective value, but rather how tolerable bugs are: intolerable in either setting.) The functionality demands are far higher for entertainment: the overall task of Quality Assurance is reasonably said to be harder.

> ("Pure AI"). (At least, such is true regarding planning capability; perhaps machine learning can claim some impact.) The reason is that videogames are *too* hard: the research (miserably) fails to meet the computational constraints. So instead videogame AI is a 'black magic', a.k.a. an engineering discipline: a large set of *ad hoc* techniques that only experts apply with ease.

That 'unprincipled' approaches work well in practice is a deep and powerful lesson about the nature of intelligence, as follows. Intelligent behavior is remarkably easy to generate in practice by *apparently* dumb method. In fact it requires great cleverness (and enormous effort) to find the apparently dumb method that actually works: there are myriad 'objectively' simple methods, and most fail. For our purposes, the takeaway is just as originally stated: It is compelling to pursue far deeper understanding of the *least* forms of Temporal Planning.

The following is an opinion piece that is potentially useful to those uninitiated in the domain-independent automated planning literature. It is only my opinion, surely subject to change, and just as surely not even said particularly well (for example, it may come off as critique, which is unintended). Still—taken with salt—it may help to better place this work in, say, the broader context of all of Computer Science. For setup: There is a certain complex abstract truth about the entire field that is quickly grasped, rarely stated, and ultimately of little significance. To put it succinctly, the entire field speaks at one level of interpretation removed from reality; to properly place the research in broader context (at least) one *meta-* should be prefixed to every claim.

1.6.1 The AI Effect in relation to Temporal Planning Research

There is a deep point about Artificial Intelligence, Applications, and Motivation that we should make. AI has *profoundly* yet only *subtly* effected reality. In particular, the field is justified in laying claim to 'every' *fundamental advance*—but not more than that—in Computer Science.

If that appears false, then consider: *Intelligence* is a fundamentally mysterious thing. As soon as an AI researcher invents a comprehensible method for endowing machines with novel capability: it cannot be, or at least does not get to be, called "intelligence". That is just because we understand the method ("comprehensible"), but do not—perhaps cannot—understand "intelligence". So another term is employed to label the novel capability, and some new discipline grows up around it.

There is an interesting relationship to the point about dumb methods producing intelligent behavior. Research in novel disciplines moves from the so-called principled to the measurably effective. Getting there involves rewriting/reimagining the principles, and trying again; eventually, we find the (what becomes) conceptually simple method to solve (what is no longer) the hard problem.

Regarding said novelty and following growth: It is a disservice to belittle the enormous work that separates *prototypes* and *real systems*. Likewise is it a disservice to belittle the enormous work that separates *dreams* and *prototypes*.

Then, regarding AI, Applications, and Motivation, the fact of the matter is as follows. To be perfectly honest, *direct* motivation—of the kind powerful enough to procure funding, shall we say—goes one way: from the real world to AI. That is because, at the end of the day, AI can only be a mathematics (because when useful, it gets renamed).

As such, AI has 'no value', and researchers in AI 'do nothing useful'. Engineers, in contrast, create useful things. Out of all people that create or do useful things, engineers are special to us: engineers take mathematics to be—at least a little bit—useful. (Naturally, we may wear both hats.)

So if we want to be well and truly accurate about applications of the theory herein: the 'application' is to clarify thought. Then we 'should' say everywhere at least "meta-meta-practice" and "meta-practice": it takes at least one more level of interpretation to do something useful with the ideas here. That is because between here and reality is *at least* one whole brain's worth of complexity.

Realistically, the amount of blood, sweat, and tears it takes to field what is recognizably a real system with automated planning capability (say, the Mars Rovers) is the product of several years and dozens, if not hundreds or yet greater, of people (*i.e.*, in dollars: several millions, if not billions, of dollars). To place things in proper perspective then, neither this dissertation, nor any other, can possibly claim more than a drop in such buckets. That is because disserations are 'cheap'. For example, this dissertation cost about a quarter million dollars to make: one graduate student supported over 2006–2012 (discount to reflect more than one purpose).

Let us then recall, so as to better appreciate, the opening motivation: fixing methodological flaws of the temporal tracks of the International Planning Competitions. First of all, we should understand as point of fact that the competitions are the largest-scale and best-executed 'empirical evaluation' of domain-independent automated planning 'systems'. To be realistic, that means:

> The IPCs just so happen to be the best, or at least most visible, simulation-based analysis of planning research prototypes. That the format is a so-called competition is immaterial, and in particular the prize purse

> is—deliberately—laughably small. The real prize is that the field pays attention to the authors of the top systems. So fame is the prize, and deservedly so when we are exceedingly careful in the design and execution of the event. If so, the event is for the better. If not, the event does more harm than good.

Secondly, we should understand that fixing methodological flaws is important, as just strongly implied. That is because none of the competing systems are real, nor are any of the so-called benchmarks. So there is no objective ground truth to the numbers: the measurements do *not* amount to scientific observation of anything physical. That does not make them useless! It just means that their value is contingent upon further analysis. It also means that, given the lack of direct connection to any sort of physical truth, it is especially important that the experiments be designed as well as possible. They *are* already designed very well. I argue here that accepting the thesis—the present state of the art taking part in the competition consists of three kinds of temporal planning—would yield improved experimental design.

Chapter 2

Definitions and Notation for Variations on Temporal Planning

This chapter formally specifies two practically-motivated temporal planning languages (as extensions of the basis in Sequential Planning): *Conservative* Temporal Planning, and *Interleaved* Temporal Planning. The endeavor is interesting unto itself. Specifically, as a practical matter, carefully designing ones formal language is an important step along the way to efficient implementation. Or, if you prefer: Here we condense several lessons learned, from the practice of temporal planning, into carefully circumscribed definitions of the task to be performed.

We focus on one aspect: the definition of permitted concurrency. At one extreme is to entirely forbid concurrency, as in Sequential Planning [66, 103, 116, 149, 174]. At the other is to permit 'everything' [37, 110, 137, 151, 154, 168, 169]. In between are the implicit (*i.e.*, procedurally-defined) semantics of a number of implementations [6, 16, 29, 39, 54, 56, 58, 64, 85, 87, 100, 119, 135, 142, 146, 188, 202]. Truthfully these systems all differ from one another in minor details. We pretend though that there are only two kinds of temporal planning between the extremes, and, here, define them.

So, including the extremes, we distinguish four approaches: sequential, conservative, interleaved, and synchronous. The **sequential** approach is the degenerate case: define concurrency as impossible. The **conservative** approach is to permit concurrent actions so long as the result is equivalent—but achieved sooner—to some sequencing of them. The **interleaved** approach is to permit concurrent actions so long as a decomposition into their effects is effectively sequential; *i.e.*, the

approach is to decompose actions into the conservative perspective. The simplest variation permits an action to decompose into only effects at its endpoints. Everything more general is lumped under the heading "synchronous", and is beyond our scope. The *cart-pole balancing problem* is an example; all such forms of planning involving continuum-many actions are beyond scope.

Organization. The definitions come in 3 main sections, followed by a summary of the whole, Section 2.4. Our notation is not terribly unusual; that which is less typical is discussed in Appendix A.

- Section 2.1 formalizes sequential planning problems. The key mechanisms are state transition functions, which implement effects upon fluents.
- Section 2.2 formalizes conservative temporal planning problems. The key mechanisms are vault transition functions, which implement locks (management of mutual exclusions between effects) upon fluents.
- Section 2.3 formalizes interleaved temporal planning problems. The key mechanisms are debt transition functions, which implement decomposition of compound actions into effects.

The key mechanisms are independent of one another; each kind of planning directly inherits from its special cases. Said inheritance is indeed meant as precisely as a software engineer could hope: implementations may easily re-use code. Which also means that the definitions may be re-read in any order.

The Standard Temporal Planning Language. Version 2.1 of the Planning Domain Definition Language (PDDL 2.1) is proscriptive rather than descriptive, but otherwise may be taken as roughly equivalent in purpose and, up to "Level 3", roughly

equivalent in scope [71]. One contribution here is to improve accuracy: certain oddities of its temporal semantics are universally rejected in practice. The treatment here is meant to be complementary. For example we skip formalizing, among other things: syntax in general, parameters in particular, and the useful distinction between domains and problems. Those familiar with the standard may very well be able to infer all but the notation by skipping to the summary of the core concepts at the end of the chapter (Page 80 as labeled, *i.e.*, really twelve pages further along).

Formatting, Conventions and Emphasis. Emphasis with no particular connotation is formatted *thusly*. Foreign words are identically formatted: *exempli gratia*. “We” means “You and I”, and particularly “our contribution” is meant as credit to the reader.

Notions whose precise technical meaning are being specifically called out are formatted as <u>foo</u>, chiefly when said definition has yet to appear. An opposing connotation is conveyed with single-quotes (*i.e.*, the technical meaning should be ignored): classical planning is ‘feasible’.

Definitions may be highlighted as in: **foo** means nothing in particular. For variety, the following are perfect synonyms for definition: *means*, *is*, *(precisely) when*, *denotes*, and *written as*. (We reserve *iff* and friends for propositions.) Plain $x+0 = x$ could refer to *definition* or *proposition*; to emphasize equality-by-definition write: $x + 0 \coloneqq x$.

Assignment is (dubiously) denoted by $\coloneqq$ as well. Equally dubiously treat as synonymous all of: *let*, *(re)define*, *assign*, *with*, and *where*. Other texts write, for example, $\leftarrow$ to draw the subtle distinction.

Variables/fields/accessors are formatted as in: *State*(*Initial*) is the value of the field called *State* in the structure denoted by *Initial*. Any more complicated computation on structures, such as the cost of a path, is formatted: $\mathrm{cost}(P) \coloneqq \sum_{e \in E(P)} w(e)$.

Context is deemphasized by omitting, subscripting, or abbreviating arguments. For example, vertex i of path P for some index i is preferably written just v, followed closely by v_i. Less preferable are the increasingly formal expressions: $v_{P,i}$, $v_i \in V(P)$, $v(P, i)$, and $v(P(i))$. Unimportant subexpressions are abbreviated by $\cdot$, as in: $\max(\cdot, \infty) = \infty$.

In general, our notation is only as formal/precise as rather detailed *psuedocode*. Meaning that (ultimately significant) issues such as the many precise meanings of equality are regarded as implementation details.

2.1 SEQUENTIAL PLANNING DEFINITIONS

This section presents the semantics for sequential planning (a.k.a. classical planning). The basic idea is to reduce to state transition systems. Any complete definition of a planning language needs to answer three key questions:

- What is a plan?
- What does it mean to execute a plan from a given situation?
- When do a plan, and its execution, constitute a solution?

The presentation we give for sequential planning is non-standard in that it is not based in STRIPS. (For completeness, we also reduce to STRIPS.) The intent is to better support generalizing to temporal planning.

The remainder of the section consists of three parts. We begin with an alternative encoding of BLOCKSWORLD highlighting some of the aspects of the definitions following. We go on to develop the formal machinery of, and underlying, state transition functions. Finally, with machinery in place: What are plans, executions, and solutions? Respectively: action-sequences, state-sequences, and goal-achieving executable plans.

2.1.1 INTUITION

The reader is hopefully familiar—perhaps all too painfully—with the concept of stacking blocks on top of one another, and further with the standard formal encodings of BLOCKSWORLD [76]. The following example is a less standard encoding meant to portray, by contrast, some corresponding less standard aspects of the formal treatment. In particular the treatment is inspired by SAS (and ADL) rather than

```
1: (constants  a b c table
2:             (Block {a, b, c})
3:             (Place {a, b, c, table})
4:             ((clear table) True))
5: (fluents    (below x ∈ Block) ∈ Place
6:             (clear x ∈ Block) ∈ 𝔹)
7: (action     (move b ∈ Block y ∈ Place)
8:             (let ((x (below b)) (v (= y table)))
9:             (not (= b y))
10:            (= (clear b) True)
11:            (= (clear y) True)
12:            (:= (below b) y)
13:            (:= (clear x) True)
14:            (:= (clear y) v)))
15: (init  (:= (below a) table)   (:= (below b) table)   (:= (below c) a)
16:        (:= (clear a) False)   (:= (clear b) True)    (:= (clear c) True))
17: (goal (= (below a) b)         (= (below b) c))
```

Figure 2.1: Sussman's Anomaly in single action schema BLOCKSWORLD.

```
(move c table)   ; move block c to the table, so all blocks are on the table, then
(move b c)       ; move block b on top of block c, achieving one of the goals, and finally
(move a b)       ; move block a on top of block b, achieving the other goal.
```

Figure 2.2: The solution to Sussman's Anomaly.

STRIPS [9, 66, 166]. As a result the treatment happens to better reflect the internals of leading approaches to sequential planning [76, 103, 105, 174].

Example: Implicit-Hand BLOCKSWORLD. Sussman's anomaly [192] can be compactly encoded as in Figure 2.1. The solution is given in Figure 2.2. Ultimately the meaning of the problem comes down to finding a path through state space: which is depicted in Figure 2.3. Walking through our psuedo-syntax (Figure 2.1):

1. We explicitly declare the existence of 4 objects: 3 blocks (called simply "a" through "c") and a table. In this encoding, the hand/gripper is implicit.

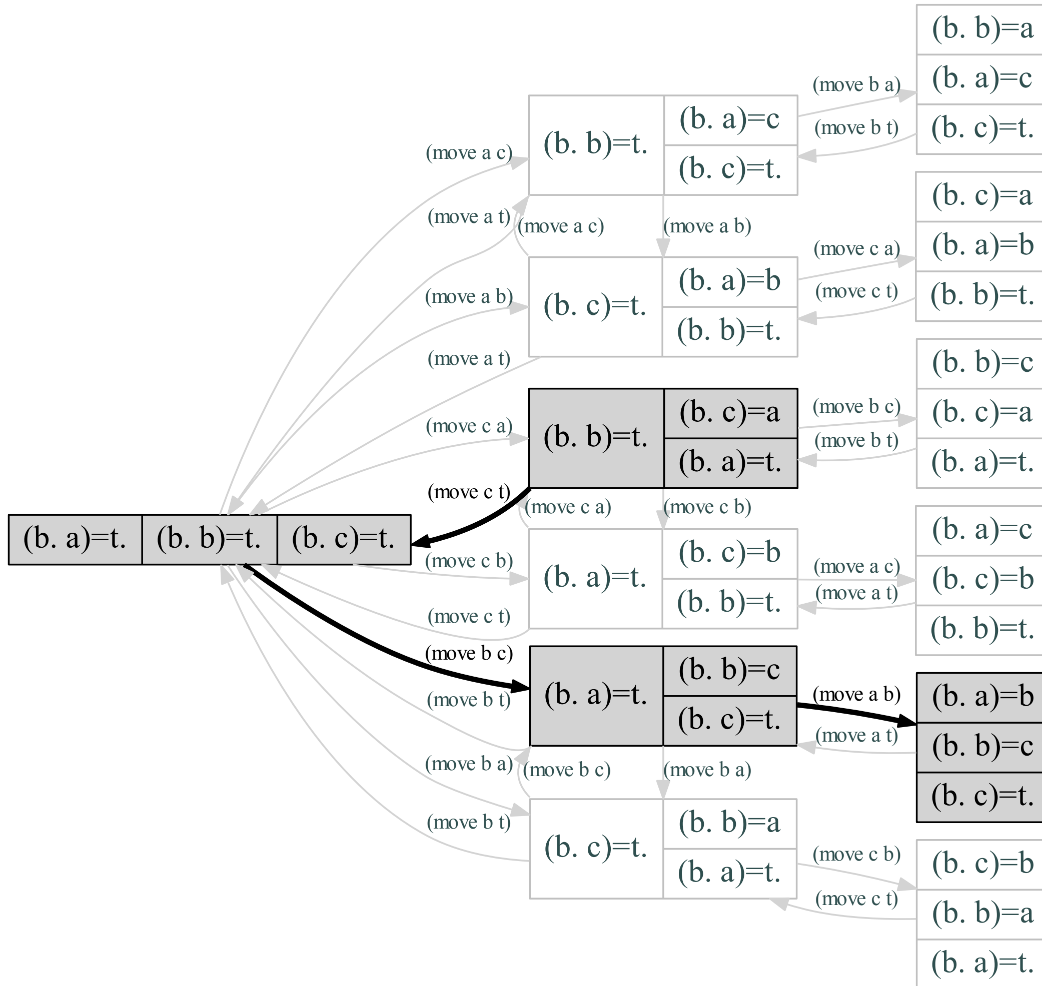

Figure 2.3: The reachable component of the state space of Sussman's Anomaly. "t." is for "table", and "b." is for "below". "(clear x) = `True`" when x is a tower top, *e.g.*, all blocks are clear in the leftmost state. The solution path is highlighted.

2. Then we declare two important subtypes of objects, *i.e.*, the blocks, and
3. the places that blocks can rest (upon each other and the table).
4. Rounding out the declaration of constants is a dubious hack; the meta-effect of declaring "(clear table)" as an always true constant is that any number of blocks may rest upon the table.
5. The important fluents to track are the locations of each block: "(below $x \in$ Block) $\in$ Place". The possible places are on top of another block (*i.e.*, no pyramids *etc.*), or, on the table. A tighter encoding would spell out that a block cannot rest on itself: "(below a) $\in$ {b, c, table}", "(below b) $\in$ {a, c, table}", and "(below c) $\in$ {a, b, table}".
6. We setup manual tracking of whether anything is above a block: "(clear $x \in$ Block) $\in \mathbb{B}$". (In more expressive contexts, *i.e.*, by exploiting quantified formula, we could skip declaring these as independent fluents.)
7. Encoding movement comes down to updating the location of the object, assuming physical plausibility. Also we need to spell out each logical consequence we care to track, namely, whether each block is clear [139].
8. First we setup two local variables: one for what is beneath the block, and the other for whether moving the block to the destination will prevent other blocks from also moving there. The formal definitions, and the state of the art, are only this expressive in a loose sense: caveats apply [67, 104].[1]

[1] The formal definitions lack the power to setup local variables: these can be turned into parameters instead. The formal definitions also lack the capability to state parameters: so the distinction is moot. To fake support for either/both we may appeal to *grounding* [71]. During grounding, for the sake of exposition, silently drop supposed effects on constants. That is typically a poor way to define the semantics of constants. Here it allows the example to consist of just 1 action schema.

9. We must prevent moving a block onto itself: "(not (= b y))".

10. Only the top blocks of towers may be manipulated, *i.e.*, the block b must be clear: "(= (clear b) `True`)". Just "(clear b)" would have the same meaning.

11. The destination y must also be clear: "(= (clear y) `True`)".

12. We declare the primary effect of moving b to y: "(:= (below b) y)".

13. Whatever was below block b (block x) becomes clear: "(:= (clear x) `True`)".

14. The destination y becomes not clear (v = `False`), except for the table (for which v = `True`): "(:= (clear y) v)".

15. Initially, block "b" rests alone, while block "c" rests on top of block "a".

16. We also initialize the clear/not-clear status of each block.

17. The goal is to build an alphabetical tower.

STRIPS *versus* SAS *Encodings of* BLOCKSWORLD. For the sake of discussion, consider the behavior of the ultimately naïve approach to sequential planning: take reduction to path-finding problems in state transition systems at face-value. The standard PDDL representation of BLOCKSWORLD in STRIPS-style uses (with b blocks): $b^2 - b$, or less efficiently b^2, booleans to encode "(on ?a ?b)", b booleans to encode "(on-table ?a)", b booleans to encode "(clear ?a)", b booleans to encode "(holding ?a)", and finally 1 boolean to encode "(hand-empty)", for a total of $(b^2 - b) + b + b + b + 1 = b^2 + 2b + 1$, or $b^2 + 3b + 1$, boolean fluents [7, 66]. So for three blocks, $b = 3$, state descriptions consist of 16 boolean values. A literal interpretation as a state transition system begins, then, by building a graph over $2^{16} = 65536$ states. In

contrast, a reasonably powerful theorem-prover, along the lines of the translator in FD [103], can first re-encode in SAS-style using only: b-many b-valued fluents, or less efficiently $(b + 1)$ many values, to encode what is below each block, and b booleans to encode whether each block is clear. (Or we could just encode in SAS -style to begin with.) For $b = 3$ that considerably improves the performance of the ultimately naïve approach by reducing the number of conceivable states (from 65536) to just $2^b b^b = 216$ states.

Now, no planner would really be so foolish as to write down, up-front, all those states permitted merely by the fluent definitions.[2] 'Surely' no planner would do more work than considering just the 13 truly reachable states of this problem, however it is encoded (see Figure 2.3).[3] In other words, the notion that representation matters because someone might take a naïve approach is, clearly, a paper-thin argument. Real issues and compelling arguments are much more complex [14, 72, 78, 104, 106, 158, 162, 167, 189]. Still, in the end, the conclusion remains the same. Namely, it remains compelling to consider representation languages that permit, as sketched here, the direct specification of useful/exploitable domain knowledge (*e.g.*, a block cannot be located in two places at once, and the hand may be abstracted away).

[2] The *anomaly* is that some planners were unable to solve the problem given any amount of time. Bounding foolishness of machines is non-trivial.

[3] With the hand abstracted away, the precise count of reachable states is given by counting the number of ways to partition the set $[b]$, representing the blocks, into lists, representing the towers. So for $b = 1$ block there is just 1 reachable state, and the sequence continues with 3, 13, 73, and 501 reachable states. By 14 blocks the count already surpasses three trillion reachable states [163].

Then observe that the research challenge is to, in practice, solve the single-source cheapest-path problem in *sub-linear* time. In other words, we must (have our planners), in practice, work above the level of graphs. The notion with counting/bounding the worst-case sizes of the graphs is to stand in for more practically useful relaxed queries, *i.e.*: heuristics.

2.1.2 Machinery: State Transition Functions

This section defines the supporting machinery for sequential planning semantics, chiefly, we define the state transition function S'_a of each action a. A **sequential planning problem** $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ consists of its:

- fluent definitions, *FluentDefs*,
- action definitions, *ActionDefs*,
- initial situation, *Initial*, and
- goal (expression), *Goal*.

The **fluent definitions** *FluentDefs* are a **fluent**-indexed collection over the possible values Values_f of each fluent f: $\mathit{FluentDefs} := (\mathit{Values})_{\mathit{Fluents}}$. A **state** is a simultaneous **assignment** of legal values to all fluents. The set of all states is denoted by $\mathit{States} := \bigtimes \mathit{FluentDefs}$. In other words each $S^{\in \mathit{States}}$ is a fluent-indexed tuple assigning legal values: $S(f) \in \mathit{Values}_f$ holds for all $f^{\in \mathit{Fluents}}$. An assignment is just a partial state; the set of all assignments over a designated subset of fluents $F' \subseteq \mathit{Fluents}$ is given by restricting to F' before taking the product. That is, let $\mathit{States}(F') := \bigtimes \mathit{FluentDefs}\restriction_{F'}$ denote all assignments over the fluents F'.

The **initial situation** *Initial*, a.k.a. the **initial state**, is a state: $\mathit{Initial}^{\in \mathit{States}}$.

The **goal** *Goal* is most generally some arbitrary boolean expression over the propositions expressing that a given fluent currently has a given value. So the goal is some boolean function over every "$f = v$" (with $v^{\in \mathit{Values}_f}$). It is not uncommon to consider much more restricted forms. We shall ignore the implementation and simply regard the goal as a total truth-function on states, $\mathit{Goal}^{\in \mathit{States} \xrightarrow{\text{total}} \mathbb{B}}$, and accordingly write $\mathit{Goal}(S)$ to mean that state S satisfies the goal *Goal*.

The **action definitions** $\mathit{ActionDefs} := (\mathit{eff})_{\mathit{Actions}}$ are an **action**-indexed family of effects. The **effect** $\mathit{eff}_a^{\in \mathit{States}(\mathit{Depends}_a) \to \mathit{States}(\mathit{Writes}_a)}$ of action a is a function: from assignments on the fluents $\mathit{Depends}_a$ that a **depends** on, to assignments on the fluents Writes_a that a alters or **writes** to. The fluents that action a only **reads** from are, then, denoted by $\mathit{Reads}_a := \mathit{Depends}_a \setminus \mathit{Writes}_a$.

Definition 2.1 (State Transition Function)**.** The **state transition function** S'_a of action a maps a state S to the next state by overwriting the fluents Writes_a with the values specified by the effect $\mathit{eff}_a(S \upharpoonright_{\mathit{Depends}_a})$. For notation, define $S'^{\in \mathit{States} \to \mathit{States}}_a$ by:

$$S'_a := \left\{ S \mapsto S \oplus \mathit{eff}_a(S \upharpoonright_{\mathit{Depends}_a}) \right\}. \tag{2.1}$$

If the effect is in fact defined in state S, then say action a is **executable** in state S:

$$S \in \mathrm{Dom}(S'_a) \iff S \upharpoonright_{\mathit{Depends}_a} \in \mathrm{Dom}(\mathit{eff}_a). \tag{2.2}$$

We say that actions depend on the fluents they write to, regardless of whether or not the value ultimately written, to that or any other fluent, actually depends on the previous value in any way. In notation, we demand: $\mathit{Writes}_a \subseteq \mathit{Depends}_a$. Likewise effects always *happen*, and are in that sense not actually proper conditional effects despite appearances [166]. In particular all the fluents in Writes_a are considered to be explicitly overwritten even if the value itself remains the same. This has no bearing on strictly sequential plans. For the purpose of generalizing to temporal planning though it is helpful to begin with this strong distinction between:

- the scope of an effect, and

- the precise transition function it implements within that scope.

2.1.3 Plans, Executions, and Solutions

With the machinery of state transition functions in place then we can address the key semantic questions: What are plans, executions, and solutions? Continue to let $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ denote a sequential planning problem.

Definition 2.2 (Plan)**.** A **sequential plan** $\left(a^{\in \mathit{Actions}}\right)_{[n]}$ of length n is a sequence of actions. The type is written $\mathit{Actions}^* = \bigcup_{k \geq 0} \left\{(a)_{[k]} \mid a \in \mathit{Actions}\right\}$.

Definition 2.3 (Execution)**.** The **exection** $\left(S^{\in \mathit{States}}\right)_{[0,n]}$, a state-sequence, of an n-length sequential plan $X = \left(a^{\in \mathit{Actions}}\right)_{[n]}$ from state S_0 is given by applying each action in turn. For each index $i^{\in [n]}$, let $a = a_i$ for the current action, let $X' = X \upharpoonright_{[i-1]}$ for the plan thus far, and define the **result** $\mathrm{Result}(X', S_0) \coloneqq S_{i-1}$ of executing the plan thus far as merely the current state. Then the execution is determined iteratively by applying the state transition function associated with the current action, supposing the action is indeed executable:

$$S_i \coloneqq S'_a(S_{i-1}). \tag{2.3}$$

If every action is executable in turn, so the execution of X is indeed defined, then say X is **executable** from S_0.

Definition 2.4 (Solution)**.** A **solution** is a sequential plan $X^{\in \mathit{Actions}^*}$, executable from the initial state $\mathit{Initial}$, such that the final result $\mathrm{Result}(X, \mathit{Initial})$ of the execution satisfies the goal Goal. So X is a solution precisely when $\mathit{Goal}(\mathrm{Result}(X, \mathit{Initial}))$

holds. Let $\text{Solutions}(\mathcal{P})$ denote the set of all solutions:

$$\text{Solutions}(\mathcal{P}) \coloneqq \{X \mid \mathit{Goal}(\text{Result}(X, \mathit{Initial}))\}. \tag{2.4}$$

2.1.4 A STRIPS Reformulation of Action Definitions

We permit effects to be as general as partial functions of assignments; this has the meta-effect of rolling together all the usual fields. For completeness we separate out the more familiar fields, for each action a and state S:

- Let the proposition $\text{Pre}_{S,a} \coloneqq S\restriction_{\mathit{Depends}_a} \in \text{Dom}(\mathit{eff}_a)$, denoting executability, be the **precondition** of a in S.
- Let the assignment $\text{Del}_{S,a} \coloneqq S\restriction_{\mathit{Writes}_a}$, the current fluent-value mappings, be the **deletes** of a in S.
- Let the assignment $\text{Add}_{S,a} \coloneqq \mathit{eff}_a(S\restriction_{\mathit{Depends}_a})$, the new fluent-value mappings, be the **adds** of a in S.

The collection $(\text{Pre}, \text{Del}, \text{Add})_{\mathit{States},\mathit{Actions}}$ is a reasonably intuitive reformulation of *ActionDefs* into 'STRIPS'. More specifically, the state transition functions are identical. The state resulting from executing a STRIPS action (P, D, A) is typically defined: So long as the precondition P holds, then the result is given by carrying out the deletes D and adds A in that order.

Proposition 2.1. *Let* $\mathit{eff}^{\in \mathit{States}(\mathit{Depends}) \to \mathit{States}(\mathit{Writes})}$ *stand for the* 'SAS' *definition of action a. Consider the corresponding* 'STRIPS' *definition for the action a with respect to state* S*:* $(P, D, A) = (\text{Pre}_{S,a}, \text{Del}_{S,a}, \text{Add}_{S,a})$ *as above.*

The two definitions of execution coincide:

$$S'_a(S) = S \setminus D \cup A \qquad \textit{iff P holds in } S, \tag{2.5}$$

$$= S \oplus \mathit{eff}(S\restriction_{Depends}) \qquad \textit{iff defined.} \tag{2.6}$$

Proof. By definitions, verbosely: The precondition $P = S\restriction_{Depends} \in \mathrm{Dom}(\mathit{eff})$ is the definition of whether the effect $A = \mathit{eff}(S\restriction_{Depends})$ is defined. So executability coincides: it suffices to show that the results are also identical.

The fluents that change are $\mathrm{Dom}(A) = \mathit{Writes}$. Then:

$$\begin{aligned}
S \oplus \mathit{eff}(S\restriction_{Depends}) &= S \oplus A && \text{by the definition of } A.\\
&= S\restriction_{\overline{Writes}} \cup A && \text{by the definition of overwriting,}\\
&= S \setminus S\restriction_{Writes} \cup A && \text{by the definition of anti-restrictions,}\\
&= S \setminus D \cup A && \text{by the definition of } D. \qquad \square
\end{aligned}$$

As-is the precondition has too complicated a form (*i.e.*, disjunctive) for STRIPS proper. To complete the reduction it suffices to split actions up over the individual assignments that effects are defined upon. Which corresponds to the exponential strategy for conditional effects; polynomial strategies are also possible [158].

Proposition 2.2. *Let* $(\mathrm{Pre}, \mathrm{Del}, \mathrm{Add})_{States,Actions}$ *be the reformulation thus far. The proposition* $\mathrm{Pre}_{S,a}$ *expressing that action a is executable in state S naturally splits into simple conjunctions of literals:*

$$\mathrm{Pre}_{S,a} \iff Q \in \mathrm{Dom}(\mathit{eff}) \text{ such that } \mathrm{And}_{f\in Depends}(S(f) = Q(f)). \tag{2.7}$$

Proof. (1) The conjunction $\text{And}_{f \in \text{Dom}(Y)}(X(f) = Y(f))$ is equivalent to demanding that Y be a restriction of X, written $Y = X\restriction_{\text{Dom}(Y)}$. (2) Assignments in $\text{Dom}(\mathit{eff})$ are upon the fluents $\mathit{Depends}$. Then the proposition holds because the right-hand side rewrites to the definition of $\text{Pre}_{S,a}$:

$$\begin{aligned} \text{Pre}_{S,a} &\iff Q \in \text{Dom}(\mathit{eff}) \text{ such that } Q = S\restriction_{\text{Dom}(Q)} && \text{by (1),} \\ &\iff S\restriction_{\mathit{Depends}} \in \text{Dom}(\mathit{eff}) && \text{by (2).} \end{aligned}$$ □

Corollary 2.3. *Actions can be compiled into an equivalent, potentially exponentially large, set of* STRIPS-*actions:*

$$(\text{Pre}_{a,Q} = Q,\ \text{Del}_{a,Q} = Q\restriction_{\mathit{Writes}},\ \text{Add}_{a,Q} = \mathit{eff}(Q))_{a \in \mathit{Actions},\ Q \in \text{Dom}(\mathit{eff})} \tag{2.8}$$

encodes *ActionDefs* as STRIPS-actions.

Proof. Let a be an arbitrary action, $Q^{\in \text{Dom}(\mathit{eff}_a)}$ an assignment its effect is defined upon, and S any extension of Q to a full state (so a state in which a is executable). The corollary makes three claims. (Size) For even just all boolean fluents, $\text{Dom}(\mathit{eff})$ could be as large as $2^{|\mathit{Depends}|}$. (Syntax) Each field $\text{Pre}_{a,Q}$, $\text{Del}_{a,Q}$, and $\text{Add}_{a,Q}$ is a partial state: hence equivalent to a set of atomic propositions [66]. (Semantics) The deletes and adds are correct by the first proposition: $\text{Del}_{a,Q} = \text{Del}_{S,a}$ and $\text{Add}_{a,Q} = \text{Add}_{S,a}$ hold. By the second proposition the preconditions $\text{Pre}_{a,Q} = Q$ are correct.

□

2.2 CONSERVATIVE TEMPORAL PLANNING DEFINITIONS

This section presents the semantics for conservative temporal planning (CTP). The name emphasizes the definition of permitted concurrency between actions: actions are permitted to be concurrent only when the outcome is entirely unambiguous. The implementation is by locks, as in concurrent programming.

The remainder of the section consists of three parts. We elaborate on the meaning of and motivations behind the conservative interpretation by way of two further encodings of Blocksworld. We go on to develop the additional machinery grounding the high-level intuitions, specifically, we make precise the notions of vaults (collections of locks) and vault transition functions. Finally, with machinery in place: What are plans, executions, and solutions? Respectively: dispatch-sequences, (state, vault)-sequences, and deadline-goal-achieving executable plans.

2.2.1 Intuition

The basic idea is to permit two actions to execute concurrently only if it is totally implausible that they could interfere. Precisely, if action a writes to anything that action b depends upon, or vice-versa, then say action a and action b are **mutually exclusive**. Naturally, it is forbidden to concurrently execute mutually exclusive actions. This is an extremely conservative stance to take. (For example, even commutative actions are considered mutually exclusive.) Why should we take it?

Example: Sequential Blocksworld *within Temporal Planning.*. Recall the encoding of Blocksworld given for sequential planning (Figure 2.1). We could adapt that model to temporal planning by just ascribing a duration to movements, however, the result would no longer correctly model our physical understanding of Blocksworld:

```
(constants  a b c table [hand] (Block {a b c}) (Place {a b c table}) ((clear table) True))
(fluents    [(empty hand) ∈ {True}] (clear b ∈ Block) ∈ 𝔹 (below b ∈ Block) ∈ Place)
(action     (move b y)[[duration = 2]]  (let ((x (below b)) (v (= y table)))
            (not (= b y)) (= (clear b) True) (= (clear y) True)
            (:= (below b) y) (:= (clear x) True) (:= (clear y) v)
            ;; Model the hand! Seems pointless, but...
            [(= (empty hand) True)]
            [(:= (empty hand) True)]  ))
```

Figure 2.4: A conservative temporal encoding of (Single Handed) Blocksworld.

as follows. So long as we move disjoint blocks to disjoint places: the fluents involved, with respect to the details of that encoding, are disjoint as well. Then it is "totally implausible" that such actions could interfere. Which means that we would end up permitting concurrency. But with only one hand it is impossible to act concurrently!

The modeling mistake lies in *entirely* abstracting away that hand. To capture the physical impossibility we should instead write as in Figure 2.4. The change is to retain the constant-valued fluent "(empty hand)". The effect of leaving behind an explicit model of the hand, however trivial it is, is that every pair of actions is considered to interact by virtue of reading from and writing to this common 'fluent'. Then, as desired, every pair of actions is considered mutually exclusive and so no concurrency is permitted.

The point to note is that we resolved the issue without modeling at a significantly more detailed level. So for example, the number of states remains the same no matter how many constant-valued fluents we add. Then in short, a reason to take the conservative interpretation is as just shown: so that we can enforce true mutual exclusions—real physical constraints—that have been otherwise hidden due to the level of abstraction, without having to significantly back down from said level.

Constants and Constant-Valued. The reader might have noted the delicate point that "(empty hand)" is said to be a *fluent* and "(clear table)" is said to be a *constant*. While the formal definitions are unable to make this distinction (because constants are discarded in a fully instantiated interpretation) the notion is that: in temporal planning, while we may drop effects upon terms declared constant, we may *not* do so for terms merely proven to be constant-valued when not changing. For contrast, in sequential planning, it is perfectly legitimate to treat constant-valued fluents as constants proper; such is the distinction between Figures 2.1 and 2.4.

(uses x). It matters little what such constant-valued 'fluents' are called, and even less what their sole value is said to be. We find it less distracting, at the level of examples,[4] to write just "(uses x)" to convey that the enclosing action requires exclusive access to object x for unstated reasons.

So: What of an example for permissible concurrency?

Example: Concurrency via *Multiple Hands in* Blocksworld. In this example there are two hands: "x" and "y". Then concurrency is conceivable. Indeed, there would seem to be no purpose to having two hands but for using them concurrently. The intended task is to invert two towers each of height 2. The natural solution, using a single hand per tower-to-be-inverted, is permitted even under the conservative definition of concurrency. Of the 5 longest paths depicted in Figure 2.7, the first 3 are all duration-optimal variations upon the natural solution. The encoding is in Figure 2.5, with a duration-optimal solution in Figure 2.6.

[4] Formally, we may drop constant-valued fluents from states yet retain their locks.

```
(constants  a b c d table (Block {a, b, c, d}) (Place {a, b, c, d, table}) ((clear table) True)
            [x y (Hand {x, y})] )
(fluents    (clear b ∈ Block) ∈ 𝔹 (below b ∈ Block) ∈ Place)
(action     (move b ∈ Block y ∈ Place [h ∈ Hand] )[duration = 2]
            (let ((x (below b)) (v (= y table)))
            [(uses h) (uses b) (uses x) (uses y)]
            (not (= b y)) (= (clear b) True) (= (clear y) True)
            (:= (below b) y) (:= (clear x) True) (:= (clear y) v)))
(init  (:= (below a) table) (:= (below b) a)     (:= (below c) table) (:= (below d) c)
       (:= (clear a) False) (:= (clear b) True) (:= (clear c) False) (:= (clear d) True)))
(goal (= (below a) b)                             (= (below c) d))
```

Figure 2.5: A temporal encoding of Multi-Handed Blocksworld.

```
0: (move b table x)[2]   ; b, a, and x are all unavailable for 2 units,
0: (move d table y)[2]   ; d, c, and y are all unavailable for 2 units,[5]
    ;; wait until all objects are available once more,
    ;; arrive at the state with all blocks on the table,
2: (move a b x)[2]       ; a, b, and x are all unavailable for 2 units,
2: (move c d y)[2]       ; c, d, and y are all unavailable for 2 units,
    ;; wait until all objects are available once more, and finally
    ;; arrive at the state with the two towers inverted as desired, at time 4.
```

Figure 2.6: A solution to the Inverting Towers Problem of Figure 2.5.

Applying Sequential Planners Anyways. It is interesting to observe a potential 'mistake' that a less temporally-aware planner could make on this problem. If ignoring the possibility of concurrency altogether, a planner might choose to build a single alphabetical tower: that also satisfies the goal. That solution has the same 'cost' of 4 actions, but, its duration is 8 time units (*cf.* the fourth of the longest paths in Figure 2.7). Then it is dominated by the natural solution; but in the eyes of a sequential planner the two appear to be of the same quality. Whether or not the 'mistake' is made, then, would be a toss-up.

[5] It is easy, but cumbersome, to ensure a lack of mutual exclusions regarding the table.

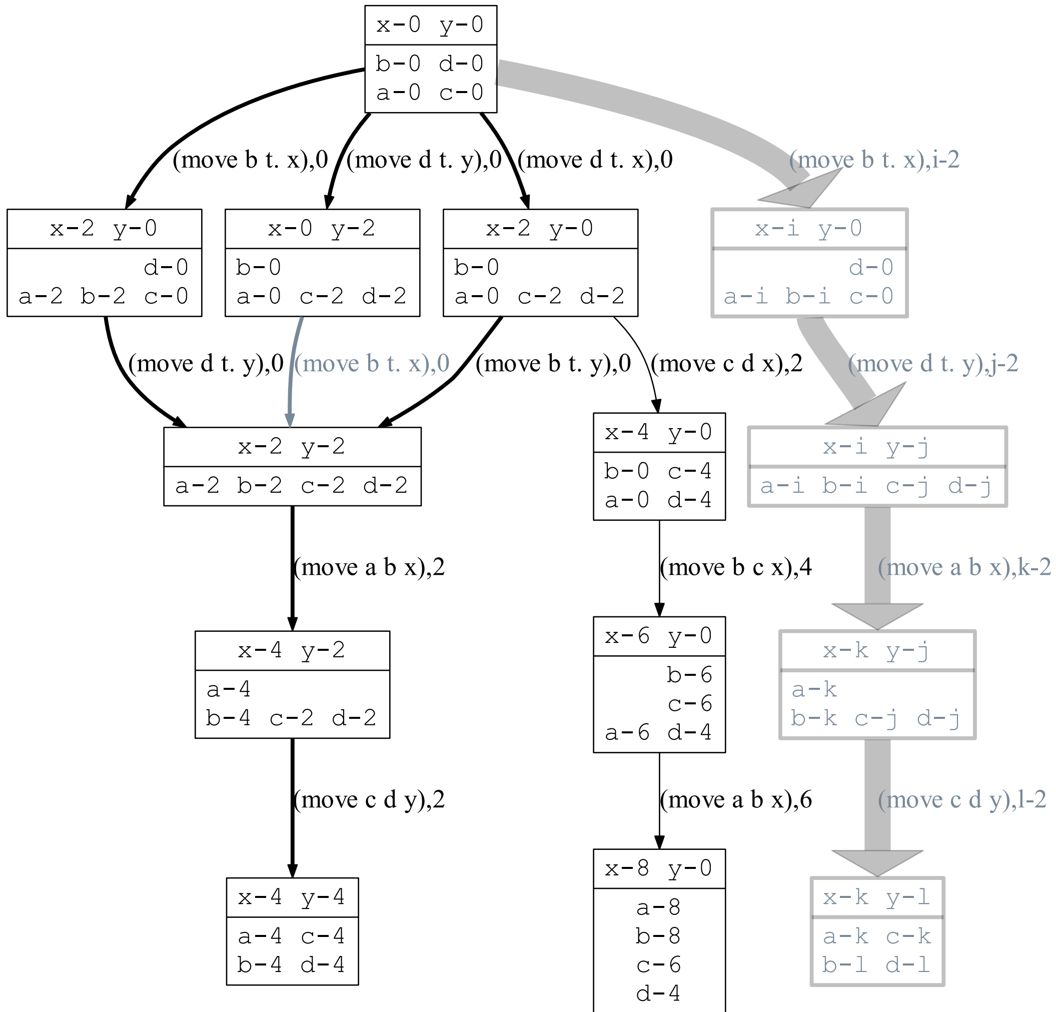

Figure 2.7: A fragment of the infinite CTP-situation space for the BLOCKSWORLD encoding of Figure 2.5. Vertex geometry represents the underlying state; the annotations are the earliest times that each object may be used.

Actually, though, a typical domain-independent sequential planner often prefers exploiting concurrency—under rescheduling—despite being unaware of concurrency in the midst of planning.[6] Especially LAMA is naturally biased in this manner [174]. An accurate technical explanation is neither simple nor compelling; clearly the supposed bias can be neither strong nor reliable. Perhaps it will suffice to imagine that heuristics are naturally more accurate concerning non-interfering actions. Regardless of 'why': That concurrency can be 'ignored' and yet successfully exploited nonetheless is a valuable observation in practice.

2.2.2 Machinery: Durations, Deadlines, Locks and Vault Transition Functions

The additional machinery to be employed for tracking whether or not specific instances of concurrency are to be permitted consists of locks. The protocol for manipulating locks is exceptionally—and deliberately—draconian. In part this is just because a temporal planner can—and is expected to—schedule actions so as to avoid all waiting/blocking. Informally:

Locking Protocol. There are two types of locks: read-locks, and write-locks. Read -locks can be shared, write-locks are exclusive. If some action is to be dispatched at some time, but other actions hold the locks needed to proceed, then the action waits until such time as it can acquire all the locks it needs. Actions acquire any locks they are waiting for (a.k.a., blocking on) as soon as those locks are released, in first-come-first-served order. Only once an action holds all of its locks does it commence its workload. When the workload is complete (the action's duration has elapsed), and no sooner, all locks held are released.

[6] Planners capable of creating looping plans—a rare breed—should, intuitively speaking, prefer to build fewer, and taller, towers. These ought to prefer, rather than avoid, the 'mistake'.

The remainder of the section formally defines the structure of conservative temporal planning problems and the associated machinery, locks and vault transition functions, that ultimately support the formal interpretation of concurrent execution.

2.2.2.1 Problem Structures: Durations, Deadlines, and Situations

Let $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ denote a **conservative temporal planning problem**, consisting of its fluent definitions *FluentDefs*, action definitions *ActionDefs*, initial situation *Initial*, and goal (expression) *Goal*. Except for those redefined in the following, $\mathcal{P}$ inherits all those definitions made for sequential planning problems. Conceptually there are just two changes to problem structure: we permit durations and deadlines.

The **fluent definitions** $\mathit{FluentDefs} := (\mathit{Values})_{\mathit{Fluents}}$ are precisely as in sequential planning: a fluent-indexed collection over the possible values of each.

The **action definitions** $\mathit{ActionDefs} := (\mathit{eff}, \mathit{dur})_{\mathit{Actions}}$ are an **action**-indexed collection mapping each to its effect and its duration. The **effect** eff_a of action a is precisely as in sequential planning, namely, a partial function on assignments: $\mathit{eff}_a^{\in \mathit{States}(\mathit{Depends}_a) \to \mathit{States}(\mathit{Writes}_a)}$. The **duration** dur_a of action a is, for simplicity, a positive Rational: $\mathit{dur}_a^{\in \mathbb{Q}^+}$.

Locks have yet to be defined, but we shall define situations here nonetheless. Say:[7] A **situation**, a.k.a. **fortune**, (*State*, *Vault*) consists of its state *State*, a fluent-indexed collection of values, and its vault *Vault*, a fluent-indexed collection of locks. Let $\mathit{Fortunes} := \mathit{States} \times \mathit{Vaults}$ denote the entirety of situations.

The **initial situation** *Initial*, a.k.a. **initial fortune**, is a fortune: $\mathit{Initial}^{\in \mathit{Fortunes}}$. Usually the **initial vault** $\mathit{Vault}_{\mathit{Initial}}$ is trivial. That is, assume a **beginning-of-action**

[7] 'Clearly' a collection of locks constitutes a vault; a vault containing value must be a fortune.

time t_0 (*i.e.*, 0 or 1) when all initial locks are released (all of which are write-locks). For convenience, also assume there is some canonical **beginning-of-time** $t_{-\infty} < t_0$ when all initial locks are acquired. (Non-uniform initial locks, which would be a limited variation on timed initial literals [59, 194], pose no difficulties—it is simply nice to have a default.)

The **goal (expression)** *Goal* is no more expressive than a negation-free boolean expression over deadline-goals. A **deadline-goal** $g = (f, v, t)$ consists of its fluent f, its value $v^{\in Values_f}$, and its **deadline** $t^{\in \mathbb{Q} \cup \{\infty\}}$. The intuitive interpretation is that the fluent-value equality $f = v$ is to be achieved no later than time t and remain that way *for the rest of time*. Note that one is permitted to ask that $f \neq v$ hold by some deadline (*i.e.*, as a disjunction over every other possible value of the fluent). Forbidding negations only serves to prevent negation of the implied temporal quantification. The restriction is to ensure that deadline-goal satisfaction can be evaluated with respect to just final situations (rather than whole executions).

Deadline Goal Satisfaction. Rather than prove that a general temporal logic is unnecessary we shall simply define so; then instead it will remain to prove that the definition captures the intuition. Let $g = (f, v, t)$ be any deadline-goal. By fiat declare that g takes value `True` with respect to an entire execution if and only if the final situation alone, say $(S, V)^{\in Fortunes}$, satisfies both:

$$S(f) = v \qquad \text{and,} \tag{2.9}$$

$$\text{Read-Time}(V(f)) \leq t \qquad \text{(defined shortly).} \tag{2.10}$$

As the underlying propositions are, by fiat, independent of all but the final situation, then so too is any expression over them. Then we may regard the full goal expression itself as merely a (non-arbitrary!) truth-function on situations: $\mathit{Goal}^{\in \mathit{Fortunes} \xrightarrow{\text{total}} \mathbb{B}}$.

2.2.2.2 *Interpretation Structures: Locks, Vaults, and Vault Transition Functions*

Situations now additionally include collections of locks. So we need the (i) the formal representation, and (ii) how actions effect change upon it.

Definition 2.5 (Locks and Vaults)**.** A **lock** $(\mathit{Acquired}, \mathit{Released}, \mathit{Readable})^{\in \mathit{Locks}}$ is given by its: **acquisition-time** $\mathit{Acquired}^{\in \mathbb{Q}}$, which begins a right-half-open interval, **release-time** $\mathit{Released}^{\in \mathbb{Q}}$, which ends the interval at a strictly greater time, and **lock-type** $\mathit{Readable}^{\in \mathbb{B}}$, which is `True` for **read-locks** and `False` for **write-locks**.

Define the **write-time** of a lock as its release-time. Define the **read-time** of a lock as its acquisition-time if readable, otherwise, as its release-time. So:

$$\text{Write-Time}(\ell) := \mathit{Released}_\ell, \qquad \text{and} \tag{2.11}$$

$$\text{Read-Time}(\ell) := \text{ if } \mathit{Readable}_\ell \text{ then } \mathit{Acquired}_\ell \text{ else } \mathit{Released}_\ell. \tag{2.12}$$

A **vault** $(\mathit{Acquired}, \mathit{Released}, \mathit{Readable})_{\mathit{Fluents}}$ is a fluent-indexed collection of locks. Denote the type by $\mathit{Vaults} := (\mathit{Fluents} \xrightarrow{\text{total}} \mathit{Locks})$.

The changes in locks from applying an action—its **vault transition function**— are to be given by: (1) acquiring read-locks at their read-times for all fluents only read from, (2) acquiring write-locks at their write-times for all fluents written to, (3) actually starting only after acquiring all locks, and finally (4) releasing all locks at precisely the actual finish-time. We sketch the corresponding notation before the definition proper. Consider any action and some current vault.

The **earliest start-time** is given by the maximum over the read-times of the fluents only read from, and the write-times of the fluents written to:

$$\text{EST} = \max \text{Read-Time}(\cdot) \cup \text{Write-Time}(\cdot).$$

The **earliest finish-time** merely adds in the duration:

$$\text{EFT} = \text{EST} + dur.$$

Suppose an **actual start-time** and **actual finish-time** for the action, say:

$$\text{AST} \geq \text{EST}, \qquad \text{and}$$
$$\text{AFT} = \text{AST} + dur.$$

The **acquired write-locks** are a collection of write-locks on the fluents written to, held from their former write-times to the actual finish-time:

$$\text{Acquired-Write-Locks} = (\text{Write-Time}, \text{AFT}, \texttt{False})_{Writes}.$$

The **acquired read-locks** are a collection of read-locks on the fluents read from, held from their former read-times to the latest actual finish-time of all of the actions sharing each. Compute those release-times by just incrementally taking maximums, with correctness to be shown shortly:

$$\text{Acquired-Read-Locks} = (\text{Read-Time}, \max(\mathit{Released}, \text{AFT}), \texttt{True})_{Reads}.$$

All of the **acquired-locks** are the union:

$$\text{Acquired-Locks} = \text{Acquired-Read-Locks} \cup \text{Acquired-Write-Locks}.$$

So the new vault consists of replacing the old locks with the new:

$$V' = V \oplus \text{Acquired-Locks}.$$

For correctness, consider that read-locks are supposed to be held from the earliest acquisition-time to the latest actual finish-time of the actions sharing the lock. So we need to verify that incrementally taking the maximum is the same as tracking the whole set and taking the maximum later.

(Base Case:) The base case is read-locking a presently write-locked fluent. By the exclusivity of write-locks, the action starts later than the current release-time: $\mathit{Released} \leq \text{AST}$. By positive durations, the action finishes yet later: $\text{AST} < \text{AFT}$. Hence $\mathit{Released} < \text{AFT}$. So taking the maximum is harmless, because it will always end up as just the actual finish-time of the sole action now read-locking the fluent:

$$\max(\mathit{Released}, \text{AFT}) = \max\{\text{AFT}\}, \qquad \text{because} \qquad \mathit{Released} < \text{AFT}.$$

(Inductive Case for Sharing Read Locks:) The inductive case is read-locking a presently read-locked fluent. By induction, the former release-time is equal to maximizing over the former set of read-locking actions. By associativity and commutativity of max, it is enough to set the new release-time by taking the maximum

of the former release-time and the action's actual finish-time:

$$\max(\textit{Released}, \text{AFT}) = \max\{\text{AFT}_a \mid a \text{ shares the lock}\}.$$

To make all of the dependencies explicit:

Definition 2.6 (Vault Transition Function)**.** Let vault V be arbitrary. The **vault transition function** $V'_{a,t} \in \textit{Vaults} \xrightarrow{\text{total}} \textit{Vaults}$ of a dispatch of action a for **requested start-time** t is given by the following.

$$\begin{aligned}\text{EST}_a(V) &:= \max \text{Read-Time}(V(\textit{Reads}_a)) \\ &\quad \cup \text{Write-Time}(V(\textit{Writes}_a)).\end{aligned} \tag{2.13}$$

$$\text{EFT}_a(V) := \text{EST}_a(V) + \textit{dur}_a. \tag{2.14}$$

$$\text{AST}_{a,t}(V) := \max(\text{EST}_a(V), t). \tag{2.15}$$

$$\text{AFT}_{a,t}(V) := \text{AST}_{a,t}(V) + \textit{dur}_a. \tag{2.16}$$

$$\text{Acquired-Read-Locks}_{a,t}(V) := \begin{pmatrix}\text{Read-Time}(V_f), \\ \max(\textit{Released}(V_f), \text{AFT}_{a,t}(V)), \\ \texttt{True}\end{pmatrix}_{f \in \textit{Reads}_a}. \tag{2.17}$$

$$\text{Acquired-Write-Locks}_{a,t}(V) := \begin{pmatrix}\text{Write-Time}(V_f), \\ \text{AFT}_{a,t}(V), \\ \texttt{False}\end{pmatrix}_{f \in \textit{Writes}_a}. \tag{2.18}$$

$$\begin{aligned}\text{Acquired-Locks}_{a,t}(V) &:= \text{Acquired-Read-Locks}_{a,t}(V) \\ &\quad \cup \text{Acquired-Write-Locks}_{a,t}(V).\end{aligned} \tag{2.19}$$

$$V'_{a,t}(V) := V \oplus \text{Acquired-Locks}_{a,t}(V). \tag{2.20}$$

Finally define the **earliest vault transition function** $V'_a \,^{\in Vaults \xrightarrow{\text{total}} Vaults}$ by forcing the actual start-time to the earliest start-time:

$$V'_a(V) \coloneqq V \oplus \text{Acquired-Locks}_{a,\text{EST}_a(V)}(V). \tag{2.21}$$

2.2.3 Plans, Executions, and Solutions

So, with machinery in place, we can define plans, executions, and solutions.

Definition 2.7 (Plan)**.** A **plan** is a schedule; a **schedule** $(a,t)_{[n]}\,^{\in(Actions\times\mathbb{Q})^*}$ of length n is a sequence of **dispatches**, *i.e.*, time-stamped actions. The time-stamps are the **requested dispatch-times**, a.k.a. **requested start-times**, of the corresponding actions. *The time-stamps may be unsorted.*

Definition 2.8 (Execution)**.** The **execution** of a schedule $X = (a,t)_{[n]}$ of length n from situation $F_0 = (S_0, V_0)$ is the sequence of situations $F = (S, V)_{[0,n]}$ given by applying the associated state and vault transition functions iteratively (and independently). The **underlying** state-sequence and vault-sequence are $(S)_{[0,n]}$ and $(V)_{[0,n]}$. For each index $i^{\in[n]}$, let $a = a_i$ be the current action, let $t = t_i$ be the current requested start-time, and let $X' = X\restriction_{[i-1]}$ be the schedule thus far. Define the **result** $\text{Result}(X', F_0) \coloneqq F_{i-1}$ of executing the plan thus far as just the current situation. In turn define the current situation by applying the associated transition functions:

$$S_i \coloneqq S'_a(S_{i-1}), \tag{2.22}$$

$$V_i \coloneqq V'_{a,t}(V_{i-1}). \tag{2.23}$$

When entirely defined, then say the schedule X is **executable** from situation F_0.

Definition 2.9 (Solution)**.** A **solution** is a schedule $X^{\in(Actions\times\mathbb{Q})^*}$, executable from *Initial*, such that the final result $\mathrm{Result}(X, Initial)$ of the execution satisfies the goal *Goal*. So schedule X is a solution precisely when $Goal(\mathrm{Result}(X, Initial))$ holds. Let $\mathrm{Solutions}(\mathcal{P})$ denote the set of all solutions:

$$\mathrm{Solutions}(\mathcal{P}) \coloneqq \{X \mid Goal(\mathrm{Result}(X, Initial))\}. \tag{2.24}$$

2.3 INTERLEAVED TEMPORAL PLANNING DEFINITIONS

This section presents the semantics for interleaved temporal planning, building upon both conservative temporal planning and sequential planning. The basic aim is to weaken the draconian restrictions of the conservative interpretation. We should want to, for example, be able to permit commutative actions to execute concurrently. (Or forbid: on a case-by-case basis.) The idea is to split up mutex actions into smaller intervals, thereby allowing us to reduce the temporal scope of the various mutual exclusions concerned. That in turn then allows a temporal planner to carefully weave the smaller intervals together without actually scheduling any two still-mutex intervals concurrently. Simply, we shall sharply distinguish actions and their effects:

- Interfering action *may* be concurrent.
- Interfering effects *may not* be concurrent.

Formally the semantics rest upon the notion that one is obliged, having begun an action, to carry out all of its parts (*i.e.*, effects). Situations then remember (in addition to a state and vault) also every outstanding obligation. Ultimately we have the desired result that executions are by composition of transition functions.

The remainder of the section consists of three parts. We demonstrate the significant relaxation of the interpretation of concurrency by way of a further psuedo -encoding of Multi-handed Blocksworld. Then we jump into the formal treatment, building up the machinery of debt transition functions. Finally we make the semantics precise:

- plans shall be either action-schedules or effect-schedules,

- executions shall be actual (state, vault, debt)-sequences, and

- solutions shall be debt-free deadline-goal-achieving executable plans.

2.3.1 Intuition

Let us begin by reexamining the potential for concurrency in Blocksworld. Our conservative model overlooks an interesting ability that the normal (4 action schema) version of Blocksworld would have if given two hands. Consider the problem of taking the bottom element of a 3-height tower and putting it on top:

(**init** (**and**	(:= (below a) b)	(:= (below b) c)	(:= (below c) table)
	(:= (clear a) `True`)	(:= (clear b) `False`)	(:= (clear c) `False`)))
(**goal**	(= (below a) b)		(= (below c) a))

Figure 2.8: Build the tower c-a-b, from the tower a-b-c.

In the hypothetical classical encoding of two hands we get to consider the plan where one hand holds up the top block long enough to clear the bottom block, thereby saving 2 actions:

```
(unstack a b x)   ; hold block a in the air with hand x,
(unstack b c y)   ; retrieve block b off of block c with the other hand y,
(putdown b y)     ; immediately place it on the table,
(pickup c y)      ; retrieve block c with hand y,
(stack a b x)     ; have hand x stack block a on block b, meanwhile
(stack c a y)     ; finish building the requested tower.
```

Figure 2.9: The smallest solution to the problem of Figure 2.8.

In contrast, a one-handed agent has to put down, and later pick up, block a: for a total of 8, rather than 6, actions. How does the two-handed plan translate into our abstraction? Considering that "move" is just a macro with respect to normal Blocksworld, it is straightforward to collect together the various pairs and infer what ought to be the equivalent schedule: see Figure 2.10.

0: (move a b x)[4]
1: (move b table y)[2]
3: (move c a y)[2]

Figure 2.10: A schedule equivalent to the plan in Figure 2.9?

But unfortunately the conservative interpretation of concurrency would reject this plan/schedule without a second thought: the (only) pair of concurrent actions are clearly interfering with one another. Picture hand x stacking block a on block b in mid-air. Will hand y really be able to handle setting both down at once? Still, given our physical understanding of Blocksworld, we already know that the concurrency, interpreted carefully, may be permitted. So: We seek a (i) domain encoding, and a (ii) definition of concurrency, such that the formal interpretation is as desired.

Example: Multi-handed Blocksworld *with Interleaved Concurrency.* As we shall see, going beyond the conservative case, interleaved concurrency gives us the desired capability (consider Figure 2.10 to be a solution to the problem of Figure 2.8). Our encoding of the dynamics is in Figure 2.11. Figure 2.12 loosely depicts the formal interpretation, which is as desired, of the schedule in Figure 2.10.

Discussion. The interleaved encoding is almost identical to the hypothetical extension to multiple hands of the classical 4-action-schema encoding of Blocksworld. (The comments in the temporal encoding are hopefully clear enough, *e.g.*: the beginning of a *movement* is either a *pickup* or an *unstack*.) The only difference is that the temporal version is able to express the constraint that a block cannot be held up indefinitely. That constraint is not too important: we presume blocks can be held up quite a while. So ignore the (quantitative aspects of the) temporal machin-

```
(constants  a b c table (Block {a, b, c}) (Place {a, b, c, table}) ((clear table) True)
            x y (Hand {x, y}))
(fluents    (clear b ∈ Block) ∈ 𝔹 (below b ∈ Block) ∈ Place)
(action     (move b ∈ Block y ∈ Place h ∈ Hand)[duration ∈ {2, . . . , 20}]
            (let ((x (below b)) (v (= y table)))
            (uses h) (uses b) (not (= b y)) (= (clear b) True)
            (:= (below b) y)
            (start [duration = 1] ; corresponds to the pickup/unstack part
              (uses x)
              (:= (clear x) True))
            ;; When y is the table, assume the following does nothing.
            (end [duration = 1] ; corresponds to the putdown/stack part
              (uses y)
              (= (clear y) True)
              (:= (clear y) v))))
```

Figure 2.11: An encoding of Blocksworld for interleaved action concurrency.

ery, thereby obtaining a fully classical version of the problem. Do the same for the conservative encoding of multiple hands.

Then we have two ways to think about multi-handed Blocksworld from a strictly classical perspective. One of them uses just full movement actions; the other breaks those up into two parts each.

For *single-handed* Blocksworld that distinction is computationally meaningless, because having retrieved a block there is nothing else to do but to complete its movement. With multiple hands though, the distinction is quite meaningful. To start with, the number of interleavings of movements is much larger than the number of sequences of movements. (Suppose n independent actions and compare $n!$ permutations with $\binom{2n}{n}n!$ interleavings: we can reasonably expect, *e.g.*, heuristic plateaus to be *far* larger from the latter perspective.) Moreover those extra plans can be useful.

Specifically plans such as that depicted in Figure 2.12, costing 3 movements, can only be found from the interleaved perspective. Whereas the best plan we may

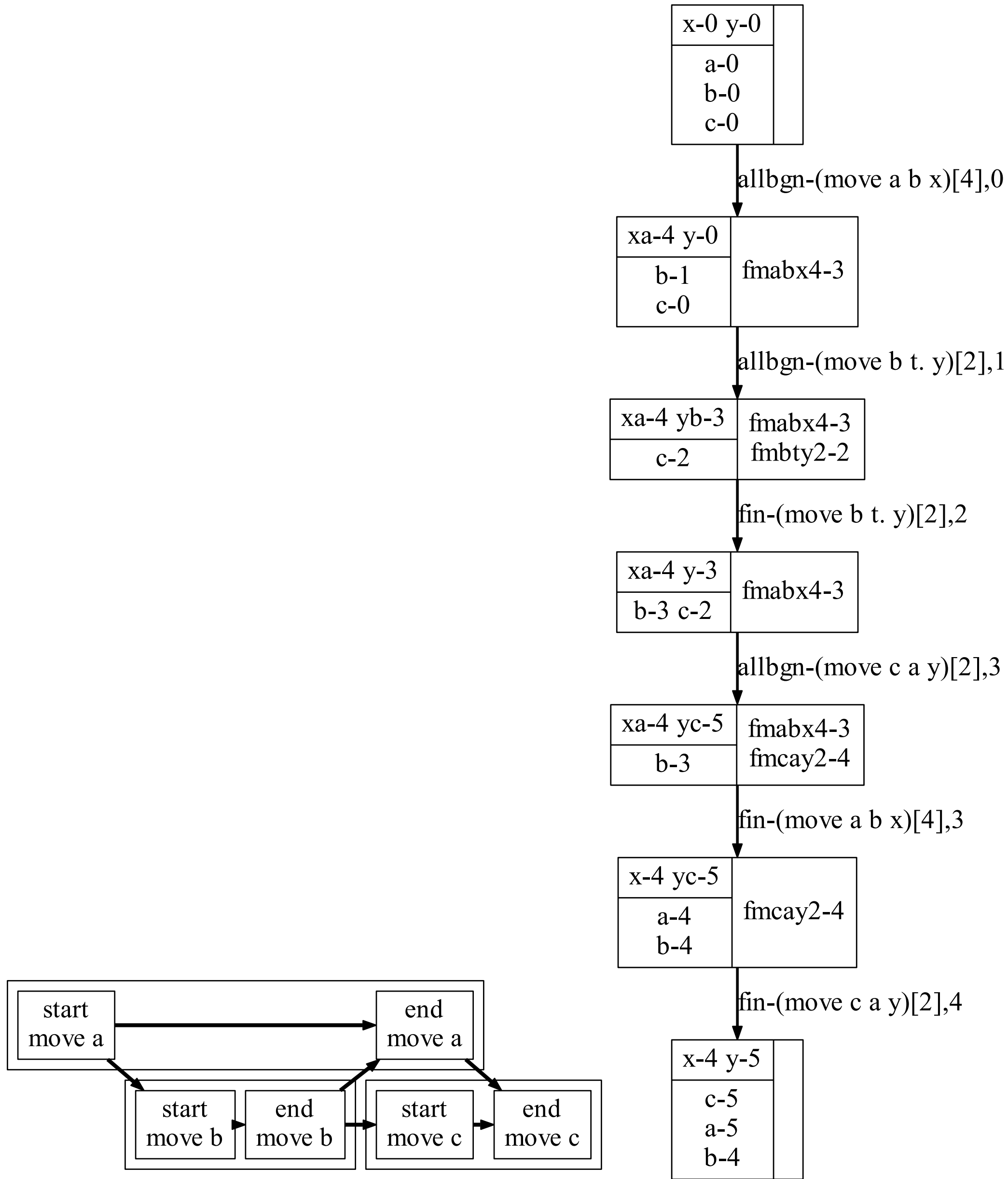

Figure 2.12: An ITP-plan and loosely its execution, solving the encoding of Figure 2.11. Compare with Figure 2.7. The extra structure for (i) vertices records when unfinished actions must finish, and for (ii) edges interleaves actions' parts.

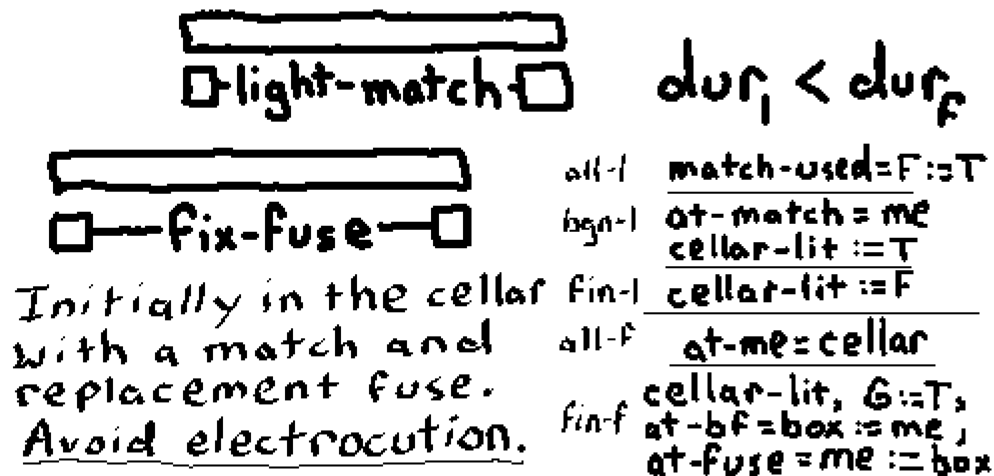

Figure 2.13: A variation on a motivating example for temporal PDDL.

find from the conservative perspective, as initially discussed, costs 4 movements instead of 3. So here the engineering tradeoff is between *quality* (of the solution plans) and *runtime* (of the planner).

It seems likely that the runtime advantage will be too large. That is, the conservative interpretation of concurrency seems to—in BLOCKSWORLD—have quite the upper hand. (Unless of course *optimality* is non-negotiable, as is sometimes the case.) We considered the quintessential classical planning domain towards motivation in order to faciliate discussion of the technical connections between the various forms of planning. In particular BLOCKSWORLD lends itself nicely to abstraction by pairing up the beginnings and endings of movements.

Naturally the better motivation towards favoring interleaved temporal planning is to step notably further beyond the understanding of classical planners (*i.e.*, further beyond than the conservative interpretation of concurrency already does). Consider the problem of repairing a broken fuse, as in Figure 2.13. Specifically note that, in an extreme case, the only source of light to work by may be the paltry light of a

match. Then *concurrency* is now much more important than merely plan quality. *Now* concurrency is *required*.

2.3.2 Machinery: Compound Actions, Obligations, Debt Transition Functions

In this section we precisely define the extensions in low-level semantics called for by the generalization to the interleaved interpretation of concurrency. The extensions are to compound actions, obligations, and the machinery tying them together. An obligation is a promise to enact every future part of a compound action on time. Specifically an obligation is a structure mapping each part of a compound action to the dispatch-time that each must begin execution at. A debt is just the collection of all (outstanding) obligations. The debt transition functions are responsible for actually implementing the formal semantics. Most importantly these deliberately fail to be executable if an obligation goes unmet.

Let $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ now denote an **interleaved temporal planning problem**: these consist of their **fluent definitions** *FluentDefs* as in Sequential Planning (Section 2.1), **primitive definitions** *ActionDefs* as in Conservative Temporal Planning (Section 2.2), **initial situation** *Initial*, a debt-free situation, and **goal (expression)** *Goal*, regarded as a truth-function on debt-free situations. Concerning action definitions: A general treatment would ascribe deeper structure in order to describe varying ways of composing compounds out of primitives. However, and perhaps not for the best, it is overwhelmingly the case that, in theory, only a single possibility is permitted.

An **(endpoint-interleaving) (compound) action** $\alpha^{\in \mathit{Compounds}}$, the only kind of compound permitted, permits effects over precisely three sub-intervals of its executions, namely: the whole interval, a starting interval, and an ending interval. We

use compound/action interchangeably. Let *Intervals* denote the set of names of the sub-intervals (of executions) of an action that effects are permitted to occupy, here, only: $\mathit{Intervals} \coloneqq \{\mathtt{all}, \mathtt{bgn}, \mathtt{fin}\}$. Also define the set of the (names of the) strict sub-intervals (of executions *etc.*), here, only: $\mathit{StrictIntervals} \coloneqq \{\mathtt{bgn}, \mathtt{fin}\}$. An **(endpoint-interleaving) (compound) action definition** is implicitly given by the definitions of its three primitive parts:

its **all-part**, formally named $(\alpha, \mathtt{all})$, written $\mathtt{all}\text{-}\alpha$,

its **start-part**, formally named $(\alpha, \mathtt{bgn})$, written $\mathtt{bgn}\text{-}\alpha$, and

its **end-part**, formally named $(\alpha, \mathtt{fin})$, written $\mathtt{fin}\text{-}\alpha$.

That is, consider the primitive names $\mathit{Primitives} \coloneqq \mathit{Compounds} \times \mathit{Intervals}$ to reveal the associated compound and sub-interval. The all-part is chiefly a notational convenience: only a **psuedo-part**. The others are **proper** parts.

The **primitive (action) definitions** $\mathit{ActionDefs} \coloneqq (\mathit{eff}, \mathit{dur})_{\mathit{Primitives}}$ are a **primitive**-indexed collection mapping each to an action definition of Conservative Temporal Planning. So each such definition $\mathit{ActionDefs}_a$ of a primitive consists of its **effect** eff_a, a partial function from/to partial states, and its **duration** dur_a, a positive Rational. We refer to parts/primitives/effects interchangeably, despite the subtle differences. Naturally: Both the **all-effect** $\mathit{eff}_{\mathtt{all}\text{-}\alpha}$ and the **start-effect** $\mathit{eff}_{\mathtt{bgn}\text{-}\alpha}$ are to start at the actual start-time of the action; Both the all-effect $\mathit{eff}_{\mathtt{all}\text{-}\alpha}$ and **end-effect** $\mathit{eff}_{\mathtt{fin}\text{-}\alpha}$ are to end at the actual finish-time of the action.

Definition 2.10 (Obligation and Debt)**.** An **obligation** is the collection of **promised start-times** for each still future part of an action execution. Let $\mathit{Obligations} \coloneqq \mathit{StrictIntervals} \to \mathbb{Q}$ denote the type. A **debt** is an action-indexed collection of obligations. The type is written $\mathit{Debts} \coloneqq \mathit{Compounds} \xrightarrow{\text{total}} \mathit{Obligations}$. The **trivial** debt is denoted $(\emptyset)_{\mathit{Compounds}}$.

A **situation** $B = (State, Vault, Debt)^{\in Balances}$ (to distinguish from other definitions say **balance**, *i.e.*, $Balances := States \times Vaults \times Debts$) consists of its state $State$, its vault $Vault$, and its debt $Debt$. A situation is **debt-free** when all obligations have been fulfilled: when its $Debt = (\emptyset)_{Compounds}$ is trivial. We assume, but it is of no consequence, that the initial debt $Debt_{Initial}$ is trivial. We *demand* that final situations be debt-free; define **(deadline-)goal-achievement** $Goal(B)$ to imply that the situation B is debt-free:

$$Goal(B) \Rightarrow \left(Debt_B = (\emptyset)_{Compounds}\right). \tag{2.25}$$

The **state transition function** of each primitive remains precisely as in Sequential Planning. Note that primitives here are, in formal terms, the actions of simpler forms of planning. (However, the truer relationship is that compounds here correspond to actions there.) So for cross-referencing, $Primitives = \mathrm{Dom}(ActionDefs)$ and $\mathrm{Dom}(ActionDefs) = Actions$ both hold: but we shall always avoid the last name.

The **vault transition function** for each pair of a primitive a and demanded start-time $t^{\in \mathbb{Q}}$ remains as in Conservative Temporal Planning: under mapping the demands to requests. The distinction is that we shall no longer automatically fix broken start-times. That is we shall insist that dispatches be **actual**, meaning that the demands are for times weakly later than earliest possible. The **restricted vault transition function** for each pair builds in that restriction. So, recalling that the full collection of (unrestricted) vault transition functions may be written $V' \in Primitives \times \mathbb{Q} \times Vaults$, we may write:

$$V'_{\text{restricted}} := V' \restriction_{\text{actual}}, \qquad\qquad \text{where}$$

$$\text{actual} := \{(a, t, V) \mid t \geq \text{EST}_a(V)\}.$$

The setup for debt transition functions is as follows. For our purposes there are three types of obligations,[8] respectively consisting of those promises that remain after executing each part on-time. The order is fixed: denote the cycle by Oblige = $\mathit{Obligation}_{\texttt{all}} \mapsto \mathit{Obligation}_{\texttt{bgn}} \mapsto \mathit{Obligation}_{\texttt{fin}} \mapsto$. The promises themselves are just to start the start-part immediately, $\text{PST}_{\texttt{bgn}} := \text{AST}$, and to start the end-part so as to finish with the action, $\text{PFT}_{\texttt{fin}} := \text{AFT}$, that is, $\text{PST}_{\texttt{fin}} := \text{AFT} - \mathit{dur}_{\texttt{fin}}$. Then, with respect to a particular execution (fix AST, AFT):

$$\begin{aligned}
\mathit{Obligation}_{\texttt{all}} &:= (\text{PST})_{\mathit{StrictIntervals}}, && (2.26)\\
&= \{\texttt{bgn} \mapsto \text{AST},\ \texttt{fin} \mapsto \text{AFT} - \mathit{dur}_{\texttt{fin}}\},\\
\mathit{Obligation}_{\texttt{bgn}} &:= (\text{PST})_{\mathit{StrictIntervals} \setminus \{\texttt{bgn}\}}, && (2.27)\\
&= \{\texttt{fin} \mapsto \text{AFT} - \mathit{dur}_{\texttt{fin}}\}, \text{and}\\
\mathit{Obligation}_{\texttt{fin}} &:= (\text{PST})_{\mathit{StrictIntervals} \setminus \{\texttt{bgn},\texttt{fin}\}}, && (2.28)\\
&= \emptyset;
\end{aligned}$$

$$\text{Oblige} := \left\{\begin{aligned}
&\mathit{Obligation}_{\texttt{all}} \mapsto \mathit{Obligation}_{\texttt{bgn}},\\
&\mathit{Obligation}_{\texttt{bgn}} \mapsto \mathit{Obligation}_{\texttt{fin}},\\
&\mathit{Obligation}_{\texttt{fin}} \mapsto \mathit{Obligation}_{\texttt{all}}.
\end{aligned}\right\} \qquad (2.29)$$

Definition 2.11 (Debt Transition Function)**.** The **debt transition function** of effect $a = (\alpha, x)$ at actual start-time t is denoted by $D'_{a,t}{}^{\in \mathit{Debts} \to \mathit{Debts}}$ and defined as follows.

[8]The obligation remaining after the all-part is ephemeral. In practice one may just alternate between recording and erasing $\text{PST}_{\texttt{fin}}$.

- For the all-part ($x = \texttt{all}$), let AST $= t$ and AFT $=$ AST $+ dur_a$ in order to: record the associated promises, *i.e.*, set $D_\alpha(y) := \text{PST}_y$ for $y^{\in StrictIntervals}$.
- For the start-part ($x = \texttt{bgn}$), recompute AST as AST $= t$ in order to: check+erase $D_\alpha(\texttt{bgn}) = t$.
- For the end-part ($x = \texttt{fin}$), recompute AFT as AFT $= t + dur_{\texttt{fin-}\alpha}$ in order to: check+erase $D_\alpha(\texttt{fin}) = t$.

In short:

$$D'_{(\alpha,x),t}(D) := D \oplus \{\alpha \mapsto Obligation_x\}, \qquad \text{but only when:} \tag{2.30}$$

$$D_\alpha = \text{Oblige}^{-1}(Obligation_x). \tag{2.31}$$

2.3.3 Plans, Executions, Solutions

Then (most of) the machinery is in place. Plans are normally given as action-schedules. It is convenient to define plans as effect-schedules, and later support action -schedules by another layer of interpretation. The technique, covered shortly, is to induce a corresponding effect-schedule to serve as the interpretation. First we define plans, executions, and solutions.

Definition 2.12 (Plan)**.** A **plan** is either an effect-schedule, or an action-schedule. An **effect-schedule** $(a,t)_{[m]}$ is an m-length sequence of **effect-dispatches**; each consists of its effect $a^{\in Primitives}$ and its **demanded start-time** $t^{\in\mathbb{Q}}$. Likewise an **action-schedule** $(\alpha,t)_{[n]}{}^{\in(Compounds\times\mathbb{Q})^*}$ is an n-length sequence of **action-dispatches**.

Definition 2.13 (Execution)**.** The **execution** of an effect-schedule $X = (a,t)_{[m]}$ of length m from situation B_0 is the sequence of situations $B = (S,V,D)_{[0,m]}$ given by

iteratively applying the appropriate transition functions. We demand that dispatch -times be actual: $t_i \geq \mathrm{EST}(a_i, V_{i-1})$ for $i^{\in[m]}$. When entirely defined, then say the effect-schedule X is **executable** from the situation B_0.

Drilling down, for each index $i^{\in[m]}$: let $a = a_i$ be the current effect, let $t = t_i$ be the current demanded start-time, let $X' = X{\restriction_{[i-1]}}$ be the effect-schedule thus far, and then firstly define the **result** $\mathrm{Result}(X', B_0) \coloneqq B_{i-1}$ of executing the plan thus far as the current situation. Secondly define the current situation $B_i \coloneqq (S_i, V_i, D_i)$ by applying the state, restricted vault, and debt transition functions, with actual $\coloneqq$ $\{(a, t, V) \mid t \geq \mathrm{EST}_a(V)\}$ supporting the restriction to actual dispatch-times:

$$S_i \coloneqq S'_a(S_{i-1}), \tag{2.32}$$

$$V_i \coloneqq V'{\restriction_{\text{actual}}}(a, t, V_{i-1}), \qquad \text{and} \tag{2.33}$$

$$D_i \coloneqq D'_{a,t}(D_{i-1}). \tag{2.34}$$

The definition forces actual start-times and demanded start-times of executable schedules to coincide. The reason is just to force the promised start-times to be identical, and hence actual, as well. Which could be enforced in other ways; this way the existences of the underlying sequences are independent of one another.

Definition 2.14 (Solution)**.** A **solution** is an effect-schedule $X^{\in(\mathit{Primitives}\times\mathbb{Q})^*}$, executable from *Initial*, such that the final result $\mathrm{Result}(X, \mathit{Initial})$ of the execution satisfies the goal *Goal*. So X is a solution precisely when $\mathit{Goal}(\mathrm{Result}(X, \mathit{Initial}))$ holds. Let $\mathrm{Solutions}(\mathcal{P})$ denote the set of all solutions:

$$\mathrm{Solutions}(\mathcal{P}) \coloneqq \{X \mid \mathit{Goal}(\mathrm{Result}(X, \mathit{Initial}))\}. \tag{2.35}$$

The requirement that final situations be debt-free, which is folded into the definition of goal-achievement, means that all actions must finish for a plan to be considered complete. This is not easily generalized away: future parts of an action may threaten satisfaction of the goal. These must be evaluated before checking the goal expression means what it is supposed to.

2.3.3.1 Inducing Effect-Schedules from Action-Schedules

Then we come to formally defining, by reduction to induced effect-schedules, the interpretation of action-schedules; any question about the meaning of an action-schedule is to be answered by considering its induced effect-schedule instead. Conceptually, to induce an effect-schedule from an action-schedule we replace the compounds by their parts, and then simply sort by dispatch-times. For notation:

Definition 2.15 (Expansion)**.** Let $Y = (\alpha, t)_{[n]}^{\in(Compounds \times \mathbb{Q})^*}$ be an action-schedule. Let the **simple expansion** $\hat{Y}$ of action-schedule Y be the sequence obtained by simply expanding each compound in place:

$$\hat{Y} := \begin{pmatrix} (\texttt{all-}\alpha_i,\ \hat{t}_{\texttt{all},i} = t_i), \\ (\texttt{bgn-}\alpha_i,\ \hat{t}_{\texttt{bgn},i} = t_i), \\ (\texttt{fin-}\alpha_i,\ \hat{t}_{\texttt{fin},i} = t_i + dur_{\texttt{all-}\alpha_i} - dur_{\texttt{fin-}\alpha_i}) \end{pmatrix}_{i \in [n]} . \tag{2.36}$$

Each such **action-expansion** ensures the effects receive the correct dispatch-times.

While the all-part is generally convenient, here it is inconvenient: it and the start -part have identical dispatch-time. There are any number of easy resolutions to the technicality. For example we could just flatten the simple expansion and invoke a

stable sort by dispatch-times. We could also just eliminate the all-parts by divvying up their work between the respective start-parts and end-parts.

Instead we <u>shuffle</u>; shuffling is a useful piece of machinery in general. The formalization captures and generalizes only the latter half of real-world shuffling of cards, namely, the process of merging together several piles of cards by interleaving their constituents. The property of interest is that the relative order of each such pile is preserved. So for our purpose we consider the total-orders, "Order the all-part before the start-part before the end-part.", over the effects of each action. To shuffle these total-orders is to interleave the effects of the actions in some potentially executable fashion. In general that is interesting. Our use here is, though, only to ensure the desired tie-breaking with respect to the almost-total order on start-times.

For any sequence X, define the **sequence-order** $<_X{}^{\in \mathrm{Rng}(X)\times\mathrm{Rng}(X)\overset{\text{total}}{\longrightarrow}\mathbb{B}}$ as the total-order on its elements induced by their indices: $X(i) <_X X(j)$ iff $i < j$. (Distinguish duplicates from one another.) Let Z be a collection of partial-orders over elements Z'. A **shuffle** $W^{\in[|Z'|]\overset{\text{total}}{\longrightarrow}Z'}$ of Z is a sequence with sequence-order extending every partial-order in Z: for each $z^{\in Z}$, with $w = (<_W)$, $w \supseteq z$.

Definition 2.16 (Induced Effect-Schedule)**.** Let $Y = (\alpha, t)_{[n]}$ be an arbitrary action-schedule with simple expansion $\hat{Y} = ((\mathtt{all}\text{-}\alpha, \hat{t}_{\mathtt{all}}), (\mathtt{bgn}\text{-}\alpha, \hat{t}_{\mathtt{bgn}}), (\mathtt{fin}\text{-}\alpha, \hat{t}_{\mathtt{fin}}))_{[n]}$. Let the action-expansion orders $\left(<_{\hat{Y}_i}\right)_{i\in[n]}$ of the simple expansion $\hat{Y}$ be the collection of sequence-orders corresponding to each of the action-expansions. Let the effect-dispatch order $\prec_{\hat{Y}}$ of the simple expansion $\hat{Y}$ (partially) order effect-dispatches by their start-times: $((\alpha_i, x), \hat{t}_{x,i}) \prec_{\hat{Y}} ((\alpha_{i'}, x'), \hat{t}_{x',i'})$ precisely when $\hat{t}_{x,i} < \hat{t}_{x',i'}$ (for any i and i' in $[n]$ and for any x and x' in *Intervals*).

The **induced effect-schedule** $(a, t')_{[3n]}{}^{\in(\mathit{Primitives}\times\mathbb{Q})^*}$, with respect to Y, is (any choice of) a shuffle of the effect-dispatch order $\prec_{\hat{Y}}$ together with the action-expan-

sion orders $(<_{\hat{Y}})_{[n]}$ of the simple expansion $\hat{Y}$ of Y: 'the' induced effect-schedule of Y is any shuffle of $\{<_{\hat{Y}}\} \cup \mathrm{Rng}((<_{\hat{Y}})_{[n]})$.

We *ought* to forcibly break the remaining ties in some canonical fashion, say, alphabetically by action name. That the remaining ambiguity (simultaneous effects of differing actions) is of no consequence will become clear in the next chapter. For the moment, though, let us rephrase. Let there be only one question permitted concerning the meaning of an action schedule, with answer: an action-schedule is a solution if and only if *all* of its induced effect-schedules are solutions.

Remark 2.1. It should already be intuitively clear that, to begin with, tie-breaking over the start-times *should* be insignificant. That we can and do formally prove so can be taken as a slight improvement upon the temporal semantics of PDDL. Specifically the relative order of a simultaneous all-part and start-part of differing actions is, here, of no consequence. Stated differently, for the purpose of search, our compounds may legitimately be treated as consisting of just two parts (hence the distinction between <u>proper</u> parts and the psuedo-part): see Proposition 5.18. The same is not true of PDDL, in theory: "(over all ...)" names a proper part. Meaning that, for the purpose of search, the specification requests (quite non-obviously) that we treat every compound as consisting of three (rather than two) parts. The practice rejects that particular nuance of the specification (*cf.* Figure 3.5 and Section 3.4.2).

For completeness, note that the reverse mapping is trivial. To be more interesting let us generalize to inferring the minimal completion of an effect-schedule:

Definition 2.17 (Induced Action-Schedule)**.** Let $X = ((\alpha, x), t)_{[m]}{}^{\in(\mathit{Primitives}\times\mathbb{Q})^*}$ be an effect-schedule with sequence-order $<_X$. Let $Z = (\mathit{Compounds} \times \{\texttt{all}\}) \times \mathbb{Q}$ be the

entirety of all-part dispatches. Let the filtered sequence $\hat{X} = ((\hat{\alpha}, \texttt{all}), \hat{t})_{[n]}$ remove all but all-part dispatches, *i.e.*, such that $(<_{\hat{X}}) = (<_X)\restriction_{Z \times Z}$ holds.

The **induced action-schedule** $(\hat{\alpha}, \hat{t})_{[n]}$, with respect to effect-schedule X, is given by transliteration from the filtered sequence, *i.e.*, by replacing all-parts with the actions themselves starting from the filtered sequence $\hat{X}$.

So doubly-inducing gives the **minimal completion(s)**, *i.e.*, doing so finishes (the interpretation of) all unfinished actions in the ‘obvious’ (time-sorted) way.

2.4 SUMMARY

We have precisely defined the semantics of three planning languages. These neatly nest within one another. Sequential Planning is a restriction of Conservative Temporal Planning. Conservative Temporal Planning is a restriction of Interleaved Temporal Planning. Then for reference, the following runs through through the basics of Interleaved Temporal Planning.

An interleaved temporal planning problem is given by defining fluents, primitives, an initial situation and a goal expression:

1. Fluents are defined by sets of possible values; an assignment to all fluents is a state.

   A fluent in the midst of change is locked and its value undefined.

2. Actions, *i.e.*, compound actions, are defined implicitly by their three primitive parts: the all-part, start-part, and end-part.

3. The primitives, *i.e.*, the parts/effects, are defined by their effect, their duration, and their type (all, start, or end).

4. The initial situation is assumed to be equivalent to an ordinary state.

   So no action is in the midst of execution nor is any fluent locked.

5. Situations satisfying the goal expression must also have no actions in the midst of execution.

   Deadline-goals are permitted only so long as a mere situation suffices to check satisfaction.

The high-level interpretation of a planning problem is given by:

1. A plan is a dispatch-sequence over either effects (*i.e.*, primitives) or actions (*i.e.*, compounds).

2. An execution fills in the missing information: the situations between every plan step.

3. A solution is a plan with goal-satisfying final situation.

The high-level interpretation ultimately grounds down to machinery equivalent to the following.

There are four atomic fields per fluent per situation:

1. its perhaps inaccessible value,

2. when it got that value (the acquisition-time of the lock),

3. when it may first change (the release-time of the lock), and

4. whether or not the value is accessible (the lock-type).

There are three[9] atomic fields per compound action per situation:

1. the promised start-time of its start-part,

2. the promised start-time of its end-part, and

3. the next part, with value cycling through: all-part, start-part, and end-part.

The all-part and start-part may be combined together for free, *i.e.*, with no negative computational consequences and an insignificantly more complicated implementation. (That reduces the representation to the promised start-time of the end-part

and whether or not the action is presently executing.) In practice it is more common to instead implement support for effects over arbitrary sub-intervals of action executions.

Finally the situation transition functions are the (disjoint) unions of three kinds of underlying transition functions:

1. The state transition function of each primitive is defined in Section 2.1. These update the values of fluents and verify classical executability of the primitive.

2. The (restricted) vault transition functions of each primitive, further indexed by demanded start-time, are a minor modification to the unrestricted case defined in Section 2.2. These update the acquisition-times, release-times, and accessibility-flags of the per-fluent locks and verify that demanded and actual start-times coincide.

3. The debt transition functions of each primitive, further indexed by demanded start-time, are defined in Section 2.3. These update the promised start-times of future effects and verify that demanded and promised start-times coincide.

[9]The formal definitions use a variable number of fields, because for example that generalizes more naturally to, say, promises consisting of non-trivial constraints rather than start-times outright.

Chapter 3

Foundational Theory for Temporal Planning

In this chapter we shall prove that:

Our temporal planning formalisms each meet certain minimum intuitions regarding rescheduling*,* reordering*, and* reduction*.*

(Rescheduling:) It should be possible to consider *(re)scheduling* events with significant leeway. That is, it should be possible to at least somewhat separate *planning* from *scheduling* considerations. For example, every precise interpretation of the start-time in “Bob drives to work starting around 8:00 o’clock.” ought to work out to roughly the same result.

(Reordering:) On the more computational side of intuition: Ideally there should not be a million different ways to formally say “the same thing”. But suppose there *are* large numbers of intuitively equivalent formal statements. (This happens chiefly when intuition is intolerably ambiguous/imprecise at the technical level.) We should be able to formally prove the correctness of an appropriate equivalence relation. Which ought to be, moreover, readily computable. If not, finding some other formalization is well worth the effort. There are *lots* of ways to formally model the same intuitions—with significantly differing computational consequences.[1]

In our case the issue is that the formal definitions demand that even independent activities be ordered; a schedule is, counter-intuitively, defined as a *sequence* of dispatches. So for example, formal plans demand that dispatches of “loading a package into a truck in New York” and “painting a wall red in Los Angeles”

be ordered. Intuitively speaking the order should be irrelevant, as the immediate consequences are independent of one another. Then we should be able to find a computational exploit. Towards that aim, in short, we demonstrate that: *reordering* independent activities defines/preserves an appropriate equivalence relation.

(Reduction:) From an engineering standpoint, we should seek to streamline the process of generalizing classical planning technique. Among other things that entails demonstrating the correctness of some most-trivial non-ludicrous approach to finding plans. Particularly we ought to, and do, *reduce* to *finite* state transition systems (a.k.a., discrete finite state automata, single-source cheapest-path problems in finite directed multi-graphs, ...). Which, for temporal planning, is much harder than it sounds: time is, intuitively, infinite. In particular the truly obvious sound and complete approach—just check every schedule individually—is computationally ludicruous. So even 'brute-force' requires intelligence.

Discussion of Novelty and Significance. We regard these intuitions as crucial litmus tests of whether a given formulation of temporal planning accurately describes the state-of-the-art. As our particular treatment is novel in certain regards, if only for form, we ought to reprove the foundational theorems. Beyond merely form though: the details do meaningfully differ. For example, the formalization of re-

[1]An interesting example from general Computer Science concerns sorting. If we formalize the notion based on providing a bunch of objects and an $O(1)$-testable total-order then sorting is $O(n \lg n)$. If instead we formalize the notion as based on $O(1)$-queryable rank functions, then sorting is $O(n)$. That is, *quick-sort*/*merge-sort* are fast—*radix-sort* is, when possible, faster. The same distinction (comparison versus ranking) becomes grossly more significant when comparing, say, nested-loop versus hashing methods for computing joins in relational databases. Note that an under -appreciated fact is that, properly speaking, each state in a PDDL problem *is* a (state of a) relational database. For example, computing the set of executable actions is precisely a join—iterating which is, loosely, the *right* way to compute a relaxed planning graph. Indeed, precisely understanding planner internals benefits considerably from at least a rough knowledge of relational database theory [104, 116].

scheduling for Conservative Temporal Planning ends up as a linear-time computation with respect to actions, whereas for Interleaved Temporal Planning the runtime lies between quadratic and cubic (*i.e.*, Single-Source Cheapest Path with Negative Weights) with respect to actions' effects. Then in that respect our efforts here are worthwhile.

However, a strong case against significance may be levied. Namely: Developing novel theory should be considered, by default, to be counterproductive. That is, whenever possible, it is better to work with a standard rather than against it. So, then, we should, and do, also prove the inadequacy of preexisting theory.

In a sense we may prove sufficient excuse in one fell swoop. Chapter 2 makes the claim that our definitions are descriptive of current practice (rather than proscriptive). If true, then in a sense that is motivation enough. So consider the following. The PDDL standard claims [71], and VAL verifies [119], but virtually all other implementations reject, that the following schedule is a solution to the abstract problem shown in Figure 3.1.

1: (A)[5]

1: (B)[5]

The semantic question here concerns mainly whether a simultaneous "at start" effect may achieve an "over all" condition of another action. In the example, both actions A and B require that they be contained by the other. For action A that is by requiring that "(doing B)" be true "over all", and conversely for action B. (Those fluents are only true while each action is in fact executing.) So in order to accomplish either/both actions it is necessary that they start and end simultaneously.

Whether dispatching the actions simultaneously actually suffices is an interesting question: the standard affirms, implementations deny. Even those implemen-

tations we regard as *faithful* still reject solvability, for one of the first two of the following three related reasons. (The unfaithful simplify to Conservative Temporal Planning, in which case unsolvability is immediate: temporary effects cannot be expressed, so achieving the conditions "(doing ·)" ends up as inconceivable.)

1. Effects are taken to be $\varepsilon > 0$ long despite their specification as instantaneous. Hence the effect "(at start (doing A))" of action A starting at time 1 can only establish a condition upon such at time $1 + \varepsilon > 1$ or later, and in particular the condition "(over all (doing A))" of action B also starting at time 1 fails.

2. The "all" interval is taken to contain the "start" and "end" intervals, whereas the specification states that the "all" interval contains neither.[2] If the "all" interval is taken to contain either endpoint-effect, then the problem is unsolvable: "(over all (doing ·))" will fail at time 1 or time 6 (or both).

3. The plan is clearly not robust to perturbation. Meaning: there does not exist $\varepsilon > 0$ such that all for all $\delta_1, \delta_2 \leq \varepsilon$ the following rescheduling of the plan is also executable and goal-achieving.

$1+\delta_1$: (A)[5]

$1+\delta_2$: (B)[5]

So note: our definitions insist that (1) effects are durative, and (2) "all" means *all*.

That our treatment concerns the *de facto*, and distinct, semantics of PDDL is reasonably compelling of itself. Nonetheless, we would not follow were everyone to "jump off a bridge". That is, it is surely conceivable that the specification made

[2]To be less inaccurate, the specification takes "all" to include the conditions of "end" yet exclude everything else. To be 100% accurate is not worth our trouble here, given there exist *zero* correct implementations, which claim is left as an exercise for the reader.

```
(define (domain Simul-Domain)
(:requirements :durative-actions)
(:constants A B)
(:predicates (done ?act) (doable ?act) (doing ?act))

(:durative-action A :parameters ()
:duration (= ?duration 5)
:condition (and
        (at start (doable A))
        (over all (doing B)) )
:effect (and
        (at start (doing A))
        (at start (not (doable A)))
        (at end (not (doing A)))
        (at end (done A)) ))

(:durative-action B :parameters ()
:duration (= ?duration 5)
:condition (and
        (at start (doable B))
        (over all (doing A)) )
:effect (and
        (at start (doing B))
        (at start (not (doable B)))
        (at end (not (doing B)))
        (at end (done B)) )) )

(define (problem Simul-Problem) (:domain Simul-Domain)
(:objects )
(:init  (doable A)
        (doable B) )
(:goal (and
        (done A)
        (done B) )) )
```

Figure 3.1: Implementations reject solvability of this PDDL-syntax.

better choices, in which case we would go on: with sad fact being that the insight eluded its readers.[3] On the other hand, much of the value of implementation and

[3]Indeed, concerning what we take as the aims of Long and Fox, their design choices do appear superior. What went unforeseen is the value to be had by best formalizing less ambitious aims.

empirical evaluation goes directly to discovering that which went unforeseen: certainly we expect that regularities of practice point to truth. In short we take the issue highlighted by Figure 3.1 as yet greater excuse for our efforts here. Namely, beyond more accurately describing the state-of-the-art, we also seek to justify those deviations: we also seek to distill the lessons learned.

For example: Should we have defined a schedule as a *set*, as a *multi-set*, or, which is the case, as a *sequence*? Our choice *seems* poor. Until, that is, we come to leveraging classical planning technique to the hilt. At that time (*cf.* Proposition 3.5): appreciate well the beauty of the formalization of rescheduling. Specifically, our mapping from the best schedules of Conservative Temporal Planning to the best plans of Classical Planning is trivial. A merit is that the treatment here leaves 'painfully obvious' that every classical planner 'already is' a decent, albeit imperfect, approach to an interesting form of temporal planning.

It *seems* similarly obvious that we ought to have temporal situations declare a *current time*, but we do not. The reason is that instituting such greatly weakens the utility of the reordering intuition. Perhaps the following mental picture will serve. Consider any given fluent in isolation. For example each might be the local state-space of an agent (and then we are about to consider some centralized multi-agent planning problem, *i.e.*, a multi-body problem). Go ahead and give it a current time. Plan for many such fluents—each as independently as possible. Then note that there is no compelling technical advantage, when taking the global perspective, to demanding synchronization everywhere. Indeed, it just inserts a pointless obstacle in the way of a rather compelling motivation: decentralizing/factoring/parallelizing the planning effort [65]. So we avoid the trap. Which has the consequence that

we *need* to prove that it suffices to consider only time-sorted plans/executions/*etc.*, Corollary 3.15.

Far more abstractly, in general, our treatment is odd but useful: we partially invert the role of definitions and theorems. Our definitions bend over backwards to the limited minds of the machines (the planners). Our theorems then, in part, serve to establish that the definitions actually make sense, *i.e.*, to the humans. At the same time they serve to establish (possible) optimizations to implementations. In contrast, normally one defines the problem so as to obviously agree with intuition, and then prove that correct+efficient implementation is possible.

The technical motivation for the abnormal perspective stems from the following. Consider that we could characterize Chapter 2 as: 'we wrote out the internal representations of temporal planning employed by the state-of-the-art', or in short, 'we formalized the right datastructures'.[4] The approach is a natural way to go about understanding, and learning from, *deliberate non-conformance* with the specification of temporal PDDL.

We shall close the chapter with a continuation of this discussion.

Technical Results and Organization. Section 3.1 first, briefly, presents the reduction of Sequential Planning to State Transition Systems. Later, in Chapter 4, we shall have use for deep analysis of reachability in state-space in service of optimizing temporal planning. As the lemma in question may be stated within Sequential Planning, next, we formulate it. More specifically the result is an interesting consequence of combining the notions of forward-checking and landmarks. Incredibly

[4]Of course the state of the art disagrees well before the level of datastructures! In truth our definitions make a complex trade between *mathematical convenience*, *technical accuracy*, and *computational promise*.

it seems the potential optimization has a slight chance of having been overlooked by even LAMA (classical planning's state-of-the-art in exploitation of landmark analysis) [174]. (So *perhaps* Lemma 3.3 should be counted as novel rather than foundational.) Its real promise, though, lies strictly beyond the classical context.

Section 3.2 proves the correctness of a reduction of Conservative Temporal Planning to the same state transition system as for Sequential Planning; the difference is only the notion of the quality of a plan/path. Two significant theorems serve as tools in the reduction: the Completeness Theorem for TGP, and likewise the Completeness Theorem for Totally Unambiguous Partially-Ordered Plans (equivalently, Backström's Deordering Theorem) [8, 153, 188].

Section 3.3 covers Interleaved Temporal Planning, which is where our largest deviations are (from similar existing formal treatments, discounting implementations). We prove three theorems. First we prove a weakened form of the Completeness Theorem for TGP. Second we prove the correctness of pruning based on Deordering. Third we present the astronomically large reduction of Interleaved Temporal Planning to State Transition Systems. The combinatorial explosion is due to distinguishing vertices by time values, of which there are nominally infinitely many possibilities, but in fact only finitely many need be considered. Keeping the result, and time in particular, finite entails proving the existence of a unit time [188], that STRIPS can 'handle' bounded arithmetic [24], and that the number of times an action can be executed concurrently with itself is bounded (in our case, by 1) [35, 63, 175].

Wrapping up the chapter is a lengthy discussion of its *raison d'être*, Section 4.5.

3.1 SEQUENTIAL PLANNING THEORY: STATE TRANSITION SYSTEMS AND LANDMARKS

We (briefly) carry out the exercise of reducing to state transition systems in order to better highlight the differences in the later reductions of variants of temporal planning—*i.e.*, only for completeness.

Subsequently we immediately jump all the way to the bleeding edge: landmark analysis [174]. In particular we develop an interesting pruning rule that should serve temporal planners well.

3.1.1 Formal Reduction to a State Transition System

Let $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ denote a sequential planning problem. Construct its corresponding state transition system as follows. Let:

$$V = \mathit{States} \text{ denote the set of vertices/states.} \tag{3.1}$$

$$E = \left\{\left(u^{\in V}, v^{\in V}, a^{\in \mathit{Actions}}\right) \mid v = S'_a(u)\right\} \text{ denote the transitions.} \tag{3.2}$$

$$\Sigma = \mathit{Actions} \text{ denote the set of labels/actions.} \tag{3.3}$$

$$R = (u, v)_E = \left\{(u, v, a)^{\in E} \mapsto (u, v)\right\} \text{ map edges to their endpoints.} \tag{3.4}$$

$$\ell = (a)_E = \left\{(u, v, a)^{\in E} \mapsto a\right\} \text{ map edges to their labels/actions.} \tag{3.5}$$

$$s_0 = \mathit{Initial} \text{ denote the initial vertex/state.} \tag{3.6}$$

$$T = \left\{v^{\in V} \mid \mathit{Goal}(v)\right\} \text{ denote the set of accepting vertices, a.k.a. goal states.} \tag{3.7}$$

- Define the **state-space of the problem** $\mathcal{P}$ as the graph $G \coloneqq (V, E, R)$.
- Define the **state transition system of the problem** $\mathcal{P}$ as the the extension of its state-space by (correctly) designating the labels, edge-labeling, initial vertex, and accepting vertices. In notation: $M \coloneqq (V, E, \Sigma, R, \ell, s_0, T)$.

Then observe:

Theorem 3.1. *The solutions to a sequential planning problem $\mathcal{P}$ are the words of its state transition system M:*

$$\text{Solutions}(\mathcal{P}) = L(M). \tag{3.8}$$

Moreover the words of M are isomorphic to the labeled walks of its state-space from the initial vertex to any accepting vertex.

Proof. For the moreover, the isomorphism is just by reading the edge labels rather than the edge names; by determinism and the fixed initial vertex the result follows.

Let $\text{Result}_{\mathcal{P}}$ denote the intermediate results of execution underyling the definition of $\text{Solutions}(\mathcal{P})$; let Result_M denote the hypothetically distinct function underlying the definition of $L(M)$. It suffices to show that the functions coincide, the inputs coincide, and the constraints on output coincide, because, by definition, $\text{Solutions}(\mathcal{P}) = L(M)$ iff for all X:

$$Goal(\text{Result}_{\mathcal{P}}(X, Initial)) = \text{Result}_M(X, s_0) \in T.$$

The inputs coincide: $s_0 = Initial$. Output constraints coincide: $T = \{S \mid Goal(S)\}$. Hence it remains only to show $\text{Result}_{\mathcal{P}} = \text{Result}_M$.

Observe that both Result_M and $\text{Result}_{\mathcal{P}}$ are fully determined by just the single-step transition functions R'_a and S'_a for all $a^{\in(\Sigma=Actions)}$. So we need only show that the individual transition functions $R'_a = S'_a$ coincide for each label/action a in turn.[5]

[5]Demonstrating this, because the state transition functions S'_a are indeed functions, incidentally addresses the technicality that Result_M has not been defined, by us, unless M is deterministic.

Fix such an arbitrary choice $x^{\in(\Sigma=Actions)}$. Then demonstrating $R'_x = S'_x$ suffices:

$$
\begin{aligned}
R'_x &= \mathrm{Rng}(R\restriction_{\ell^{-1}(x)}) && \text{(by definition)},\\
&= \mathrm{Rng}(\left\{e^{\in E} \mapsto R(e) \mid \ell(e) = x\right\}) && \text{(apply the domain restriction)},\\
&= \mathrm{Rng}(\left\{(u, v, x)^{\in E} \mapsto (u, v)\right\}) && \text{(ℓ and R just select parts of edges)},\\
&= \mathrm{Rng}(\{(u, v, x) \mapsto (u, v) \mid v = S'_x(u)\}) && \text{(E is the union over S'_a)},\\
&= \{(u, S'_x(u)) \mid u \in \mathrm{Dom}(S'_x)\} && \text{(simplify away v, apply Rng())},\\
&= S'_x && \text{(set/function correspondence)}. \square
\end{aligned}
$$

Discussion.. Naturally, all else being equal, shorter or cheaper plans are better. That is, in terms of theory, finding good/best sequential plans corresponds to the Shortest Path Problem for (Edge-Weighted, Directed, Multi-)Graphs. (Which is polytime.) However the reduction involves a combinatorial explosion: state-space is exponentially large. Nonetheless the reduction can be practical, when treated carefully. Indeed variations on state-space search are the dominant approach to every flavor of domain-independent planning [7, 60, 76, 82, 105, 141].

3.1.2 Landmark Analysis in Support of Forcing Compilations

Consider a *special* sort of action:

Suppose that an executable action is an h^m-detectable landmark separating the current situation from every action it is mutex with.

Setup. Two actions are **mutex** if either writes to fluents the other depends upon. The **mutex-order** of an action-sequence is the transitive closure of orienting each mutex as per the sequence. Two plans are **behavior-equivalent** if their mutex-orders

are identical. Take it for granted (for now) that behavior-equivalent plans are result -equivalent (*i.e.*, Result(P, *Initial*) = Result(Q, *Initial*) for result-equivalent P and Q), and hence also solution-equivalent.[6] So it suffices to prune all but one representative of every behavior-equivalence class. Take any such choice as identifying the **canonical** representatives of behavior-equivalence classes. Suppose the choice of representatives is consistent with a partial-order on actions in the following sense. Say an action a is **higher priority** than a non-mutex action b if whenever the two are adjacent in a canonical plan, a precedes b. Allow that notion to be conditioned on a given subset of plans (*i.e.*, subtrees of a search tree).

Suppose we have a plan P and are investigating its forward-chaining extensions X. Let A be the set of actions executable in the final state of an execution of P. Let $\hat{A}$ denote the *special* subset of A. Let $\hat{A}_a$ denote all those special actions higher priority than $a^{\in A}$. Take every unspecial action as lower priority with respect to extensions of P: if $b \notin \hat{A}$ then $\hat{A}_b = \hat{A}$. Let P_a denote the (immediate) extension of P by some executable action a, and X_a all further extensions thereof. Let Y_b denote the set of extensions of P such that b occurs at least once.

Proposition 3.2. *Consider the highest priority* special *action. Suppose this most special action is furthermore an h^m-detectable landmark separating the current situation from satisfaction of the goal. Then all canonical solutions immediately execute it—every other choice may be pruned without loss. For notation, with $a^* \in \hat{A}$ denoting said highest-priority special action h^m-necessary for goal achievement: retaining only X_{a^*} retains all canonical solutions extending P.*

[6]The assumption that behavior-equivalence implies result-equivalence is the same as assuming that POCL Planners are sound.

In general, every extension of a plan by some action either (i) never again executes the special *actions higher priority than the choice, or (ii) is non-canonical. With notation as above: all $X_a \cap \bigcup_{b \in \hat{A}_a} Y_b$ are non-canonical.*

Proof. The claims are a combination of the definition of a landmark—a necessary condition for reachability—and that it suffices to retain just the canonical representatives of behavior-equivalence classes.

Landmarks for the goal must be included in the plan, so there is at least one a^* in every extension to a solution. Landmarks for anything else that should happen to occur are similarly forced, so everything mutex with an a^* is preceded by an a^* in every extension. Therefore everything between the first such a^* and now is non-mutex. By assumption (or Theorem 3.14, or [8, 149, 153]), it would be equivalent to swap this first instance of an a^* backwards through any such extension until it occurs (just after) now. Then the first claim is shown. As far as implementation goes: Note that one may arbitrarily declare any such special action as highest priority (one need not settle on canonical representatives ahead of time).

For the generalization: nothing mutex with a special action can occur until it does, by the definition of special. Consider any special actions that do end up occuring in some extension. Then these could have occurred right now, instead of then, without altering behavior. Force such: define them as higher priority than all non-special actions with respect to all extensions of P. (Among themselves pick an arbitrary order.) Then any special action to ever occur must occur sooner than every lower priority choice in every canonical extension. For notation, with $b \in \hat{A}_a$, $X_a \cap Y_b$ are non-canonical. □

Being a landmark for everything one is mutex with is indeed quite special. Consider, though, that in the context of compilation: such would occur regularly. That is, when one is carrying out a compilation one forces a planner to carry out some amount of auxiliary computation/book-keeping. Such forcing is not necessarily immediately noticed by the planner—it can easily be that the forcing is in the form of an eventuality. Then the proposition is getting at a mechanism that one could add to planners to make them more effective when fed compiled models. Which is important enough to be called a Lemma:

Lemma 3.3. *Assume compilation is in the restricted form of merely altering definitions and adding virtual actions/fluents. If every virtual action, whenever truly reachable, is:*

- *the sole remaining action h^m-reachable, from the last unforced choice, belonging to*
- *the cut-set of a disjunctive action-landmark, h^m-detected in preprocessing, separating*
- *executability of itself, from*
- *executability of every mutually exclusive action, and from*
- *satisfiability of the goal, such that also*
- *the current situation falls on the wrong (non-goal) side of the separation, then:*

The compilation is forcing—with respect to any forward-chaining approach sophisticated enough to find those landmarks.

Proof. The meaning is just to demonstrate that even weakest forms of the hypothesis are strong enough to satisfy the definition of forcing directly. (Strongest forms of the hypothesis are not at all interesting; computing all h^*-detectable landmarks, for example, 'during preprocessing', is not an interesting definition for "preprocessing".) There is nominally very little to show. Proposition 3.2 already covers the details of how landmarks can lead one to infer that pruning all but one option preserves completeness. We do need to show, though, that subsequent choices can also be forced without further deep analysis of unreachability. Argue by induction.

Base Case: No virtual action is possible. So the choice is only over real actions. We assume there is nothing to show.

(*First Caveat:*) That is, define/assume that every distinct permutation of real actions is a canonical representative, so no pruning needs to occur and no heuristic evaluations need be suppressed in the base case. Alternatively we may assume that any analysis powerful enough to guarantee the hypothesis concerning virtual actions could find the same independence relationships among real actions. (Clearly the target of a compilation should be unaware of the virtual/real distinction—as otherwise it amounts to a direct approach.) The ultimate fall-back position is just to weaken the definition of forcing. After all, considering at most all executable permutations of real actions is still a far and away stronger result than considering also pointless ordering decisions regarding virtual actions.

Inductive Case: A virtual action is possible. By hypothesis, the hypothesis of Proposition 3.2 is met without recourse to further analysis of unreachability. Specifically, of the possible actions there exists one immediately identifiable as (i) necessary for everything mutex with it, and (ii) necessary for the goal. Then by the Proposition it suffices to prune every other choice. Should there be multiple

possible virtual actions it does not matter which is taken as highest priority. That is, define priority by whatever choice the target planner happens to make. (Which is legitimate because there are no relevant constraints upon the meaning of "priority".) Note that heuristic evaluation is unnecessary: the forcing is identifiable without it. (And it would be reasonable to take the forced child as precisely 'good' as its parent, *i.e.*, re-use the parent's heuristic evaluation.) A target planner clever enough to suppress heuristic evaluation until arriving at an unforced choice will be notably faster; *e.g.*, if the compilation doubles the apparent size of plans then suppressing heuristics along forced paths will be a speedup of about a factor of 2.

As the plan is made longer we can more or less assume that the distance to the nearest unforced choice has decreased. Then by induction every unforced choice is only over real actions, completeness is retained, and as a bonus one heuristic evaluation is suppressed per instance of forced virtual action. So we are done—unless the distance to the nearest unforced choice is not actually smaller.

(*Second Caveat:*) The only way for the induction measure to fail to decrease is if the measure was infinite to begin with. Which means that all future choices continue to be forced—infinitely. Infinite behaviors are not equal to finite behaviors. As the pruning is completeness-preserving with respect to behavior-equivalence: then no solution in the entire sub-tree before pruning was finite to begin with. To say that every solution is infinite is to say that no solutions exist. (Solutions must be finite by our definitions.) So the last unforced choice lead to a dead-end. Moreover the dead-end is *almost* detected by the landmark analysis: all but a single infinite path is pruned (without heuristic evaluation, to boot).

All in all the possibility of such infinite paths is highly unlikely in the first place, and anyways not that problematic if appropriate search algorithms are employed.

That is, let us say that such infinite paths count as well-formed/canonical for the purposes of what must be pruned. (*I.e.*, define away the requirement to prune the infinite path.) Alternatively invoke transfinite induction (in order to reach the infinitely far away unforced choice, again eliminating the need to prune). Or just assume that infinite paths simply don't exist. By whichever resolution (further discussion following): The result is shown. □

Regarding the possibility of infinities, for concreteness, the typical sort of example is:

- In this sub-tree, the only reachable (and always executable) action ever relevant to some numeric fluent halves its value.
- Elsewhere, finite plans for achieving 0 remain.

In our setup: If only linear operations are permitted upon the numeric fluent, the planner might have very well chosen to infer at the initial state that suppressing every decreasing action ensures the fluent monotonically grows. Hence it could know that the fluent never reaches 0 if positive with decreasing actions suppressed. If furthermore every decreasing action but specifically halving has been rendered h^m-unreachable at the last unforced choice,[7] then the planner could end up in this situation of realizing that halving is necessary, and doing so immediately is sufficient. Note that if we were to permit infinite plans, then the infinite plan really would be a solution: $\lim_{n\to\infty} 1/2^n = 0$. (Which is meaningful in practice if said action of halving also becomes multiplicatively faster each time.)

Naturally, it would also have been possible to infer that the only way to finitely reach 0 by multiplicative operations is to multiply by 0 specifically. Then the whole

[7] Initially, reaching 0 would have been a large disjunctive landmark over all decreasing actions.

sub-tree would be pruned by landmark analysis, instead of all but one path. But anyways pruning down to just one path is already far superior to leaving the whole sub-tree intact. Whether the remaining difference between an empty tree and an infinite forced path is at all significant depends upon the overall search strategy.

Lookahead. So long as the search algorithm is <u>fair</u> (meaning only that it visits every infinite path infinitely often), then the existence of an infinite path in the search tree is relatively harmless [48]. For reference the 'right' implementation of generating children is to: (1) compute possible actions, (2) prune them by permutation/landmark considerations, (3) if multiple remain continue normally (evaluate heuristics and submit the children to the overall search), (4) otherwise skip evaluating the heuristic for the sole remaining child, (4') *i.e.*, pretend it works out as small as possible consistent with the parent's heuristic value, finally (5) submit it to consideration by the search algorithm as normal (as if it had siblings). In particular if guaranteeing fairness involves imposing a depth penalty, then do so in (4'). Intuitively we would prefer to look ahead as far as possible to an unforced choice: (5') go back to (1). As long as such lookahead puts a bound upon how deep one looks before falling back to the principled approach of allowing the search algorithm to make exploration choices, then lookahead still preserves fairness and thus also completeness. (Moreover we still have our desired property: no heuristic need be computed during lookahead.) So: (5") go back to (1) at most N times, else (5).

In practice the issue is presumably moot (because infinite paths are presumably ruled out in some other fashion). But for a counter-example (naïve lookahead gone awry on even a finitely solvable problem): If someone hands you a 100-disk Towers of Hanoi problem, and even the algorithm for solving it as well, it is still better to

walk away from the puzzle than to even attempt to compute the exponentially long forced sequence solving it.

Discussion. All theoretical caveats aside—the interesting issue here is whether or not this style of pruning would be useful in practice. For classical planners, utility is unlikely without significant further generalization. Generalization itself is straightforward. Whether one can find an exploitable pattern that actually occurs in planning benchmarks is the rather harder part of such an endeavor.

As far as temporal planning goes: Lemma 3.3 is almost surely beneficial. Particularly so for PDDL-style temporal planning, as we shall see in Chapter 4 (specifically concerning Theorem 4.12).

3.2 CONSERVATIVE TEMPORAL PLANNING THEORY: RESCHEDULING, REORDERING, AND REDUCTION

In this section we reduce Conservative Temporal Planning to Graph Theory, by way of a remarkably efficient reduction to Sequential Planning. The key techniques are left-shifting and deordering. Left-shifting serves to render all quantitative aspects of time irrelevant by implementing a dominance reduction from temporal to classical plans. Deordering aims at extending the reduction to classical situations proper (rather than just plans). Unfortunately deadlines leave perfection beyond reach, formally: Conservative Temporal Planning reduces to the Multi-Objective Path Problem. In contrast, "Classical Planning" usually refers to the Single-Objective Path Problem.

Perspective. The brute-force approach to reducing to Graphs would be to use one vertex per (temporal) situation, as in Figure 2.7. This is a useful way to define the semantics, but a hopeless perspective in practice; *i.e.*, temporal situations can differ from one another in *just* values of time. Distinguishing these is both a pointless and hopeless explosion of the difficulty of the task the planner faces. For this variant of temporal planning the highly attractive, and perhaps obvious, reduction is possible—use but one vertex per (classical) state—chiefly by left-shifting. Indeed, extremely fortunately, left-shifting takes *linear-time*: we may successfully apply the state-of-the-art from Classical Planning with little to no thought required. Bear in mind that this is how the temporal planning competitions have been won every single time. That is, minor extension of classical planners with trivial scheduling techniques has been and continues to be the empirically dominant strategy. Then here we have the beginnings of an explanation.

At a technical level: We adapt, to our formal definitions, the Completeness Theorem for TGP [188], then, prove it. More specifically we prove that, under conservative semantics, dispatch-sequences efficiently reduce to action-sequences by way of First-Fit/left-shifting; to reverse the mapping is just to throw away dispatch -times. So we have an efficient mapping, in both directions, between the *plans* of Conservative Temporal Planning and Sequential Planning (and a theorem guaranteeing that solutions aren't lost by exploiting it). Then the search-trees, before any pruning, are identical. While a great result, a perfect reduction must also be able to survive subsequent pruning. Then the issue is that duplicate state elimination is widely employed by classical planners. That is, it is easily said that a perfect reduction ought to be an efficient mapping between *situations* rather than *plans*. This we cannot achieve.

In lieu of a perfect reduction we adapt and prove Backström's Deordering Theorem, or equivalently the Completeness Theorem for Totally Unambiguous Partial Orders [8, 153]. So deordering serves as a sufficiently weak replacement for duplicate state elimination (which prunes too much). Specifically we prove the 'definition' of non-mutex: the sequencing of non-mutex dispatches is irrelevant. The technical upshot here is an efficient pruning rule making even state-space planners consider such sequencing to be irrelevant. Note that avoiding pointless ordering decisions is the guiding principle of Partial-Order Planning. Then in short: Deordering allows us to fool Sequential Planners into performing a kind of Partial-Order Planning.

Motivation in Two-Handed BLOCKSWORLD. For the sake of concreteness, recall the example two-handed BLOCKSWORLD problem concerning inverting two towers each

of height two. Intuitively there is just one natural solution (one hand and two actions per tower). Formally though we could state infinitely many variations by inserting pointless delays. *Fortunately, by <u>left-shifting</u>, we can efficiently avoid any consideration of pointless delays.*

However even after left-shifting there still remain six ways of formally stating the natural solution (there are six ways to interleave the activities of the two hands). Six hardly inspires fear; so consider scaling up to two towers each of height ten. There remains just one natural solution, now consisting of twenty actions, ten per tower. But now there are nearly *two-hundred thousand* uselessly distinct formal variations upon it. To be precise, choose 10 out of 20 positions of the dispatch-sequence for the 'right' hand (with the remaining 10 positions going to the 'left' hand): there are $\binom{20}{10} = 184,756$ identically behaving shuffles of the twenty dispatches. *<u>Deordering</u>, in theory, gives our planners an efficient mechanism for considering just one.*

Organization. So the remainder proves two standard results concerning *rescheduling* and *reordering* in support of the *reduction* to Sequential Planning and then to Graph Theory:

1. Section 3.2.1 proves that <u>left-shifted</u> plans <u>dominate</u> [188].
2. Section 3.2.2 proves that <u>deordering</u> characterizes <u>behavior-equivalence</u> [8].
3. Section 3.2.3 applies the techniques to reduce to the Multi-Objective Path Problem.

Finally Section 3.2.4 summarizes the key high-level points.

3.2.1 Rescheduling: Left-Shifting, Executability, and Deadline-Goal Satisfaction

For Conservative Temporal Planning it is always best to start everything as soon as possible, because doing so (1) has no impact on executability, and (2) is superior in terms of achieving deadline-goals. Formally:

Theorem 3.4 (Left-Shifting/First-Fit/TGP is Complete[188])**.** *If any schedule in an action-sequence equivalence class is a solution, then the action-sequence equivalent left-shifted schedule is as well.*

So it is completeness-preserving, and thus also optimality-preserving,[8] to prune away all but left-shifted schedules. A left-shifted schedule is the result of running First-Fit: There is at most one left-shifted schedule per sequence of actions, and it is by definition computed in linear-time. It would be hard to overstate the practical utility of a linear-time reduction from schedules of actions to sequences of actions.

Unfortunately the theorem fails for Interleaved Temporal Planning. The failure is easy enough to see if we remember to check. It is also equally easy to just mistakenly assume that faster/sooner is always better. So in a sense it pays to be extra careful here. A sketch is straightforward:

(Proof Sketch:) The locking protocol immediately gives every action the locks it is waiting on as soon as those become available. Hence starting an action any later than necessary only results in further constraining the timeline. That is, by the protocol, delaying an action does not 'make room' for other actions to 'sneak' in. So it is best to always dispatch actions as early as possible. Then note that earliest vault transition functions are everywhere defined and determined solely by the

[8]For the sake of analysis directly express metrics and quality bounds as fluents and goals, *i.e.*, so that poor quality plans become non-solutions instead.

choice of action-sequence. Therefore, having picked an action-sequence: all else is determined. In particular if an action-sequence has an assignment of dispatch-times resulting in a solution, then that assignment produced by applying the earliest vault transition functions instead results in a (better) solution. 'QED'.

Even a detailed sketch is deceptive: the fault in applying the argument to Interleaved Temporal Planning is subtle. Next is a far more detailed sketch.

3.2.1.1 Sketch of Left-Shifted Dominance: Preserve Executability and Dominate Deadline-Goal Satisfaction

A **left-shifted schedule** starts every action at its earliest start-time. Two schedules are **action-sequence equivalent** if they dispatch the same actions in the same order, *i.e.*, two schedules differing only in dispatch-times are action-sequence equivalent. One schedule **dominates**, in terms of deadline-goal satisfaction, another when—*for every hypothetical goal at once*—the dominating schedule satisfies the goal if the dominated schedule does. Then the idea is to break the theorem down into three parts: show that left-shifting (0) preserves action-sequence equivalence (which is by definition), (1) preserves executability, and (2) dominates in terms of deadline-goal satisfaction.

To begin: Rescheduling has, counter-intuitively, no impact on executability. The reason is just that the machinery permits only requests. Then attempting to dispatch an action too early merely results in that action automatically waiting till its earliest start-time. So in particular preserving executability amounts only to preserving action-sequence equivalence:

Proposition 3.5 (Executability)**.** *A schedule is executable from a situation if and only if the underlying sequential plan is executable from the underlying state. Moreover the state-sequences of the two executions are identical.*

Proof. The scheduler, namely the collection $(V')_{Actions,\mathbb{Q}}$ of all vault transition functions is, by inspection, itself a total function: $V'^{\in Actions \times \mathbb{Q} \times Vaults \overset{\text{total}}{\longrightarrow} Vaults}$. That is, every individual vault transition function is total. Hence their compositions remain total: the entire vault-sequence of every hypothetical execution exists (even if the execution does not). Executions of schedules consist of nothing more than the underlying vault-sequences and state-sequences. Then we are done, for the state-sequence of a schedule simply *is* the execution of its underlying sequential plan. Well, technically, we wrote out the definition twice. There was no mistake. □

Then it remains to argue for dominance with respect to deadlines. Consider first equivalence with respect to satisfying deadlines. Say that two action-sequence equivalent schedules are moreover **execution-identical** (rather than literally identical) if merely their actual, rather than requested, start-times match:

Proposition 3.6. *Replacing a requested start-time with the actual start-time has no effect. That is, for any action a, vault V, and requested start-time s, with actual start-time $t = \mathrm{AST}_{a,s}(V)$, the results of attempting to dispatch the action at either the requested time s or the actual time t are identical:*

$$V'_{a,s}(V) = V'_{a,t}(V). \tag{3.9}$$

Moreover, if the requested start-time is not already actual, then the actual start-time is earliest:

$$V'_{a,s}(V) = V'_a(V) \qquad\qquad (\textit{if } s < \mathrm{EST}_a(V)). \tag{3.10}$$

Proof. Trivial: $\mathrm{AST}_{a,s}(V) = \max(\mathrm{EST}_a(V), s)$ holds by definition. For completeness: If $s = t$ there is nothing to show, and otherwise the only possibility is $s < t = \mathrm{EST}_a(V) = \mathrm{AST}_{a,s}(V)$. Then $V'_a(V) = V'_{a,t=\mathrm{EST}_a(V)}(V)$ and $V'_{a,s}(V) = V'_{a,t=\mathrm{AST}_{a,s}(V)}(V)$ hold by definition. Hence $V'_a(V) = V'_{a,t}(V) = V'_{a,s}(V)$ holds. □

Action-sequence equivalent schedules *not* execution-identical to one another execute in the same way but for the precise times that changes actually take place. In particular the final situations may differ—only in the final collection of locks—and then it is possible that one is a solution and the other is not. Deadlines behave intuitively: sooner is indeed better. Formally:

Proposition 3.7 (Deadline-Goal Satisfaction (part 1))**.** *The set of true primitive deadline-goals is increasing as the read-times of locks decrease.*

Proof. For notation: Let state S, vaults X and X', and primitive deadline-goal $g = (f, v, t)$ be arbitrary. Say situation $Y = (S, X)$ is the completion of X, situation $Y' = (S, X')$ is the completion of X', $g(Y)$ is the truth-value of g in Y, and $g(Y')$ is the truth-value of g in Y'.

Increasing for sets refers, of course, to the partial-order by set inclusion (and decreasing for times to the total-order by magnitude). Then it suffices to show: If we decrease the read-time of f, (*) $\text{Read-Time}(X'_f) \leq \text{Read-Time}(X_f)$, then any deadline-goal upon f can only be 'more true', (†) $g(Y) \Rightarrow g(Y')$.

By definition, for any (final) situation $B = (S, V)$: $g(B) = (S(f) = v)$ and (Read-Time$(V_f) \leq t$). Since the states of Y and Y' are identical then (†) simplifies to: (‡) (Read-Time$(X_f) \leq t$) $\Rightarrow$ (Read-Time$(X'_f) \leq t$). If the read-time in X is small enough (less than t), then certainly the yet smaller read-time of X' is also small enough: (*) implies (‡) and hence (†) as well. □

In short: Satisfaction of deadline-goals is monotone. Then we desire that satisfaction of full goal expressions be monotone as well. Goal expressions are restricted to be negation-free: hence satisfaction is—by design—monotone.

Proposition 3.8 (Deadline-Goal Satisfaction (part 2)). *Satisfaction of negation-free boolean expressions is monotone in increasing sets of true propositions.*

Proof. By "negation-free" we mean of course compositions of just "and" and "or". Compositions of monotone functions remain monotone in general. Then note that "and" and "or" are monotone, *i.e.*, flipping inputs from false to true make their results 'more true': (and x y ...) implies (and `True` y ...), and, trivially, (or x y ...) implies (or `True` y ...). □

Then one schedule can dominate another with respect to *all hypothetical goal expressions at once*, by simply accomplishing everything faster. Naturally there is no earlier than earliest: hence left-shifted plans dominate their action-sequence equivalence classes. So the sketch is complete.

Discussion. The sketch has a few small holes here and there. None are difficult to address. For example: What does it really mean to reschedule every action to its earliest start-time? What if two mutually exclusive actions could both start 'now'? Even just a cursory attempt to apply the machinery has no choice in answer. The

locking protocol forces the later-in-dispatch-sequence of the two to wait, regardless of requested start-times given. That is, whichever is currently scheduled to occur first must remain that way; the definitions permit nothing else.

Then let us reflect for a moment on what it means to prove the theorem. In Sequential Planning all effects are permanent. But in reality all effects are temporary (the sun sets; all things die; entropy increases without bound; *etc.*). Coordinating the lifetimes of multiple effects is the *usual* story. For a random example: Only a novice begins cooking mushrooms and potatoes at the same time. *So cooking a fine meal requires careful temporal coordination of concurrent processes.* Realistically, the proper event to be astonished and impressed by is when the paradigm of purely sequential decision making actually performs well on some real-world task. This is no longer, of course, in fact astonishing; success stories abound. But nonetheless the notion that it should suffice to think only about sequences of actions is still worth meeting with a healthy dose of skepticism. The proof may very well go through—but then the planning language can hardly be said to directly apply to a plethora of real world applications. Conversely, a planning language may very well be rather expressive—but then the proof must fail.

Either way the language may be of significant practical utility, insofar as the definitions themselves may successfully address 'half' of the technical challenges. In other words, there are two extremes to language design, characterized by the remaining technical challenge:

- sufficiently optimize the implementation of the system, and
- wrap the system in sufficient layers of expressiveness-enhancing techniques.

So the question here is not really about the truth of the theorem. Whether the theorem holds 'merely' distinguishes simpler from more general forms of temporal planning. As we are interested in both kinds, we are as interested in disproving the theorem as we are in proving it. Put another way, our definition of Conservative Temporal Planning is *designed* (by ruling out specifically *required concurrency*), to collapse to Sequential Planning. Provability is then a foregone conclusion. Likewise Interleaved Temporal Planning generalizes only far enough to break the theorem.

So the (more) interesting matter is the proof itself, rather than the status of the theorem. Specifically we should investigate proof as robust to generalization as possible. 'Failure' is inevitable: we can hardly prove the theorem when it is false. But the fantastic nature of the optimization the theorem legitimizes—conduct temporal planning by straight-up application of a classical planner instead—is far too attractive to give up without a fight.

Long story short, the aim behind the level of rigour of our formal proof is to better support the salvaging of useful lesser results subsequent to failure of the theorem itself. Understandably the full proof, given in Appendix B, may not be of immediate interest. Indeed, it is likely more valuable to independently prove the theorem; one of our meta-claims is that our definitions streamline well and truly taking care of every little detail.

So next we move on to *reordering* the dispatches, leaving the time-stamps alone, which is converse to our notion of rescheduling.

3.2.2 Reordering: Deordering and Behavior-Equivalence

In this section we reformulate, more intuitively, the notion of execution—say behavior versus execution—and show that the original reduces to it. The formal notion of execution reflects unfortunate realities of implementation difficult to efficiently address in general, but usually greatly mitigated by duplicate state elimination. Duplicate state elimination is only correct under certain assumptions—assumptions not met in application to Conservative Temporal Planning. So among other things the reduction serves to describe a correct alternative.

Formally, we prove that one of Backström's theorems continues to hold when adapted [8]. For reference, Backström's Deordering Theorem is that eliminating unnecessary ordering relationships in a partial-order plan is poly-time. In contrast his Reordering Theorem is that finding the minimum number of ordering relationships on the same set of actions is NP-complete.

For our purposes, we adapt the theorem so as to motivate and prove correct the following pruning rule.

Deordering-motivated Pruning. Ascribe any canonical total-ordering to all actions. When considering adding some dispatch of an action to the end of an executable plan, first collect together the suffix of its non-mutex ancestors. Prune the dispatch if it is canonically less than any element of that suffix.

The rule is dead simple, but, important. So, reiterating: Give every action an arbitrary ID. When generating a child, scan backwards till encountering a mutex action. Compare IDs along the way. Skip to generating the next child if the current ID isn't biggest.

The pruning preserves completeness (and optimality if one records plan quality as an explicit fluent). The proof is that at least one permutation of that suffix (the sorted one) is retained, and the behaviors of all of them are identical. Roughly:

Theorem. *Deordering preserves behavior.*

The treatment is closest to that of Backström [8], but we could equally well attribute: POCL-soundness, Completeness of Totally Unambiguous Partially-Ordered Plans, GraphPlan-completeness and soundness, and Completeness of Stratified Planning [8, 16, 30, 149, 153].

Organization. We begin with a longer account of the underlying motivations before moving on to the proof proper. The technical treatment starts by defining the alternative notion of execution (behavior and behavior-equivalence), continues by building tools (mutexes and reordering), and finishes with the promised equivalence reduction. We conclude with a longer discussion of the opportunities and issues surrounding deordering.

Duplicate State Elimination is Incomplete. Slow and steady wins the race. Exhausting resources to 'win' a half-race, a pointless endeavor, immediately loses the race itself—fastest solutions do not necessarily, *nor even usually*, consist of fastest solutions to sub-problems. Then consider, as has been suggested, indeed blindly handing temporal problems to a state-of-the-art classical planner (perhaps extended by First-Fit so it can at least evaluate the actual plan quality metric [*i.e.*, duration]). Figuratively, the planner might attempt a dead-sprint through the whole problem—and keel over dead half-way through. At a technical level: duplicate state elimination could, at least in theory, prevent us from finding a fast/fastest solution if it

happens to consist of locally non-fastest sequences (because duplicate state elimination prunes locally sub-optimal sequences whenever the sub-optimality becomes known). If deadlines are tight enough, then we might fail to solve the temporal problem.

The flaw is easily 'addressed' in practice. For example, we can always just disable duplicate state elimination—or just forbid the use of deadline-goals. Can we do better? At a minimum, we should be able to prove that it suffices to consider paths rather than walks; disabling duplicate state elimination shouldn't mean we cannot even replace it with mere cycle-checking. Just cycle-checking is, though, a far cry from the power of duplicate state elimination.

Applying Deordering to Logistics. Consider search-trees on up to 10 load actions (of different packages). Loading is pairwise non-mutex, so the order is irrelevant; that is, every way of ordering a given subset results in the same final state. Mere cycle-checking does not see that, and so considers the full search-tree: $10! = 3,628,800$ leaves, $\binom{10}{9}9!$ parents of leaves, $\binom{10}{8}8!$ grandparents, ..., and 1 root for the empty plan. (If also the futility of unloading goes unnoticed then $\sum_{i=0}^{10} 10^i$ is the right expression.) Duplicate state elimination perceives that only $2^{10} = 1024$ distinct states are possible (one per subset of 10 packages), and so prunes accordingly. The deordering-motivated pruning rule forces pairwise non-mutex sets of actions to occur in an arbitrary but fixed order. Then only one path per subset of the 10 load actions survives pruning. So again the search tree consists of 1024 vertices, for a subtly different reason.

Here the pruning rule is actually significantly superior to duplicate state elimination. Its operation is search-order independent and it needs only enough memory

at any given moment to represent the plan in question. Duplicate elimination, in contrast, requires exponential memory and anyways only prunes down to 1024 vertices as its best case; if suboptimal paths are found first, even duplicates may need to be re-expanded. Of course sets of pairwise non-mutex actions are the ideal case for deordering. In general duplicate elimination is a far stronger filter (in fact too strong for our purposes). Nonetheless, somewhat surprisingly, the two may be used in tandem even in the context of classical planning [30], which we attribute largely to the search-order dependence of duplicate state elimination.

3.2.2.1 Timelines, Behaviors, and Behavior-Equivalence

The natural notion of an execution (or execution trace) is that such gives every fluent a single value per time. That is, in notation, our temptation is to write $f(t) = v$ to denote the value v of fluent f at time t with respect to the execution of interest. When many are of interest we would write instead along the lines: $Z(f,t) = v$ denotes that $f = v$ holds at time t with respect to Z. As we have already defined executions as a rather different structure, to distinguish, let us say **behavior** denotes the natural notion; so behaviors are such that each behavior $Z^{\in \mathit{Fluents} \times \mathbb{Q} \to \mathit{Values}}$ is a function mapping fluents and times to values. Precisely:

Definition 3.1 (Fluent Timelines and Behaviors)**.** A **timeline** of a fluent f is written $Z_f{}^{\in \mathbb{Q} \to \mathit{Values}_f}$, and means that $Z_f(t)$, if defined, is the value of f at time t in the timeline Z_f. A **behavior** collects all timelines together and is written $\left(Z^{\in Q \to \mathit{Values}}\right)_{\mathit{Fluents}}$. So $Z_f(t) = Z(f,t)$, if defined, is the value of fluent f at time t in the behavior Z.

For mapping from formal executions, *i.e.*, from situation-sequences, to corresponding behaviors we first define the notion of what (temporal) assertions are made by any given (temporal) situation.

Definition 3.2 (Temporal Assertion)**.** A situation Y **asserts** that each fluent f has the value $\mathit{State}_Y(f)$ specified by its state State_Y for each time that its lock $\mathit{Vault}_Y(f)$ asserts definition for. When readable, the **interval of definition** of a lock is the closed interval from its acquisition time through its release time; when unreadable, its interval of definition is just the release time of the lock. For notation, write $t \mathrel{\tilde{\in}} \ell$

to denote that lock ℓ asserts definition for time t:

$$(\tilde{\in}) := \{(t,\ell) \mid t \in \text{if } T_\ell \text{ then } [A_\ell, R_\ell] \text{ else } [R_\ell, R_\ell]\}, \qquad \text{where} \tag{3.11}$$

$$(A_\ell, R_\ell, T_\ell) = (\mathit{Acquired}_\ell, \mathit{Released}_\ell, \mathit{Readable}_\ell).$$

Then to derive the corresponding behavior of an execution, just collect together all the assertions from each situation in turn. In principle such might contradict one another: say **temporally inconsistent** or that the behavior does not exist. For said principle we ought to define corresponding behaviors as relations (relating fluent and time pairs to all values asserted by the execution for that pair). However, the chief responsibility of the machinery of vault transition functions is to ensure lack of such inconsistency beyond a shadow of doubt. So let us define corresponding behaviors as functions:

Definition 3.3. The **behavior corresponding** to an execution is given by:

$$\text{Behavior}(X) := \{(f,t) \mapsto \mathit{State}_Y(f) \mid t \,\tilde{\in}\, \mathit{Vault}_Y(f) \text{ and } Y \in \text{Rng}(X)\}\,. \tag{3.12}$$

The behavior corresponding to executable schedules is that of their executions. Entities are **behavior-equivalent** when their corresponding behaviors are identical.

As a sanity check, where **time-sorted** means that time-stamps weakly increase:

Proposition 3.9. *The behaviors of time-sorted, executable, schedules exist.*

Proof. Argue by induction on the length, n, of the execution.

Base Case: $n = 0$. The behavior of the trivial execution exists trivially:

$$\text{Behavior}(\mathit{Initial}) = (\{t \mapsto \mathit{State}_{\mathit{Initial}}(f) \mid t = \mathit{Released}(\mathit{Vault}_{\mathit{Initial}}(f))\})_{f \in \mathit{Fluents}}.$$

Inductive Case: $n > 0$. Let a_n be the last action of the schedule in question, with actual start-time t_n, in turn yielding the last situation (S_n, V_n) of the execution. By induction a behavior, $Z' = Z_{n-1}$, exists for the execution omitting the last situation. It suffices to guarantee that the temporal assertions of the last situation do not contradict those already made. So consider, in turn, any arbitrary fluent f and its by-induction temporally consistent previous timeline $Z_{n-1}(f)$.

Case: The fluent is uneffected by a_n. Then so too is its timeline uneffected, and thus trivially remains temporally consistent: $Z_n(f) = Z_{n-1}(f)$.

Case: The fluent is read by a_n. So the former value and new value are identical: $S_n(f) = S_{n-1}(f)$. Inconsistency requires differing assertions, so, temporal consistency is trivial. At most the new timeline is defined at more times: $Z_n(f) \supseteq Z_{n-1}(f)$.

Case: The fluent is written to by a_n. All prior assertions concerning the fluent f are for times prior to (or at) the previously greatest release time $\mathit{Released}(V_{n-1}(f))$ by time-sortedness. The only new assertion, from the write-lock on f, is only for the now greatest release time $\mathit{Released}(V_n(f))$, which is larger by positivity of action durations. As inconsistency requires temporally intersecting assertions: the new timeline remains temporally consistent.

Elaborating, more than simply starting last, the action begins weakly after every other interacting action: $t_n \geq \mathit{Acquired}(V_n(f)) = \mathit{Released}(V_{n-1}(f))$ by the definition of acquired write-locks. So furthermore it finished strictly last: (*) $\mathit{Released}(V_n(f)) > t_n \geq \mathit{Released}(V_{n-1}(f))$ as action durations are strictly positive. As the lock in question is a write-lock it asserts a value for the fluent only at its release time (by the definition of $\tilde{\in}$), which by (*) is strictly larger than $\mathit{Released}(V_{n-1}(f))$. Hence the only new assertion is temporally disjoint from every prior assertion, which suffices. □

More usefully:

Proposition 3.10. *Behavior-equivalence implies result-equivalence.*

More specifically, behavior-equivalent executions share final state and vault.

Proof. Let Z be the identical, corresponding, behavior of two executions with final situations (A, C) and (B, D). So we must show $(A, C) = (B, D)$. Note that executions are finite, ergo so too are corresponding behaviors, meaning there are greatest times (and intervals) for which each timeline is defined. Specifically let t^*_f be the last time in the timeline Z_f for which the fluent f has a value. Then the final states are identical: $A_f = B_f = Z_f(t^*_f)$.

If the value $Z_f(t')$ is undefined for times t' arbitrarily close to t^*_f, then the fluent f must be write-locked in both of the final vaults C and D. In this case let $t^\dagger_f$ be the penultimate time for which fluent f has a value in behavior Z. So the penultimate time $t^\dagger_f$ is the acquisition time of those write-locks (and the ultimate time is the release time): $(t^\dagger_f, t^*_f, \texttt{False}) = C_f = D_f$.

Otherwise the final interval for which the value of fluent f is defined is proper (*i.e.*, not just $[t^*_f, t^*_f]$). The only possibility is that both final vaults C and D read-lock the fluent f. In this case let $t^\dagger_f$ be the beginning of said final interval over which fluent f has a value in behavior Z. So $t^\dagger_f$ is the acquisition time of those read-locks: $(t^\dagger_f, t^*_f, \texttt{True}) = C_f = D_f$.

Hence the final situations are identical. □

The proposition does *not* close under induction to the notion that behavior and execution mean the same thing. (Behavior-equivalence does *not* imply execution-equivalence.) Indeed, the whole point is that the former captures the smaller/efficient intuitive notion and the latter the larger/convenient formal notion.

3.2.2.2 (Not) Mutually Exclusive and Behavior-Preserving Reordering

The formal notion of execution is the way it is in order to bow to the needs of planners. Specifically planners need to structure the candidate space of plans into search-trees with usefully bounded branching factors. (Planning by reduction to some other combinatorial substrate does not really change that reality, *i.e.*: SAT-solvers still search.) Branching over individual actions is a useful enough bound; for contrast, branching over sets of actions is not. So formally it is most convenient for semantics to be defined by sequences.

However, we then have the unfortunate side-effect that pointless ordering of independent activities is formally significant. One way to mitigate that is to notice, *a posteriori*, that the results of executing two plans are identical. This is not terribly useful in temporal planning, because two situations can differ in just the precise values of times. The way we pursue here is to notice, *a priori*, that reordering of independent activities will lead to identical results.

In short, the heart of the theorem is to reorder non-mutex actions:

Lemma 3.11. *Swapping adjacent non-mutex dispatches preserves behavior.*

For notation, let $(a, b)^{\in Actions^2}$ *be a pair of* ***non-mutex*** *actions, meaning neither writes to fluents the other depends on:* $Depends_a \cap Writes_b = Depends_b \cap Writes_a = \emptyset$*. Let* $(r, s)^{\in \mathbb{Q}^2}$ *be a pair of start-times for the two. Let* $F_0 = (S_0, V_0)$ *be an arbitrary situation. Let* $X = (F_0, F_a, F_{ab})$ *and* $Y = (F_0, F_b, F_{ba})$ *be the executions of the schedules* $((a, r), (b, s))$ *and* $((b, s), (a, r))$ *from situation* F_0*. Suppose both start-times are actual with respect to* V_0*,* i.e.*, satisfying* $r \geq \mathrm{EST}_a(V_0)$ *and* $s \geq \mathrm{EST}_b(V_0)$*. Then the swap preserves behavior, actualness, and moreover final situations:*

$$\mathrm{Behavior}(X) = \mathrm{Behavior}(Y), \tag{3.13}$$

$$r \geq \mathrm{EST}_a(\mathit{Vault}(F_b)), \tag{3.14}$$

$$s \geq \mathrm{EST}_b(\mathit{Vault}(F_a)), \qquad \textit{and moreover} \tag{3.15}$$

$$F_{ab} = F_{ba}. \tag{3.16}$$

(Proof Sketch:) Intuitively speaking, swapping independent actions ought to have zero impact. Non-mutex, though, is not as strong as independence: the two actions could both read from the same fluent. If so, then neither writes to it. So both see the same value: it seems the result should follow easily. However, the concerned locks do not enjoy the same independence that the values do.

Proof. So long as every part a situation is altered at most once, *i.e.*, rather than twice, the result is trivial (as sketched). However, shared read locks are updated twice, complicating matters. First we setup sufficient notation, then easily capture order independence with respect to the values of fluents, and finally argue for their locks. There are 5 situations/executions of interest, let:

$$(S_0(f), V_0(f)) = F_0(f),$$

$$(S_a(f), V_a(f)) = F_a(f), \qquad (S_b(f), V_b(f)) = F_b(f),$$

$$(S_{ab}(f), V_{ab}(f)) = F_{ab}(f), \text{ and } \quad (S_{ba}(f), V_{ba}(f)) = F_{ba}(f).$$

Abbreviate with $v_\alpha = S_\alpha(f)$ denoting the value and $\ell_\alpha = V_\alpha(f)$ denoting the lock for fluent f with respect to each of the 5 executions α. Likewise abbreviate with $Z_\alpha(f) = \mathrm{Behavior}_\alpha(f)$ denoting the corresponding timelines of fluent f. So $Z_{ab}(f) = \mathrm{Behavior}(X)(f)$ and $Z_{ba}(f) = \mathrm{Behavior}(Y)(f)$.

Neither action writes to fluents the other depends on, by the definition of non-mutex. Therefore both see the same partial state when applied, *i.e.*, for the case of

action a:

$$S_0\restriction_{Depends_a} = S_b\restriction_{Depends_a}, \qquad \text{because}$$
$$S_b\restriction_{\overline{Writes_b}} = S_0\restriction_{\overline{Writes_b}}, \qquad \text{and}$$
$$Depends_a \subseteq \overline{Writes_b}.$$

So let $Q_a \coloneqq S_0\restriction_{Depends_a}$ and $Q_b \coloneqq S_0\restriction_{Depends_b}$ denote the partial states each action sees regardless of order. Then the assignments thus computed are likewise independent of order. So let (*a) $R_a \coloneqq \mathit{eff}_a(Q_a)$ be the assignment applied to either S_0 to arrive at $S_a = S_0 \oplus R_a$, or to S_b to arrive at $S_{ba} = S_b \oplus R_a$. Likewise let (*b) $R_b \coloneqq \mathit{eff}_b(Q_b)$ be the assignment applied to either S_0 to arrive at $S_b = S_0 \oplus R_b$, or to S_a to arrive at $S_{ab} = S_a \oplus R_b$.

Then argue for each fluent timeline as separately as possible, with cases for how the two actions jointly effect each such fluent.

Case $f \notin Depends_a$. Then applying a affects neither the value of f nor its lock, hence: $v_0 = v_a$, $\ell_0 = \ell_a$, $v_b = v_{ba}$, and $\ell_b = \ell_{ba}$. In particular whenever b is applied, the input lock is always the same. Therefore the output lock is as well: $\ell_b = \ell_{ba} = \ell_{ab}$. Then regardless of order, the final sets of temporal assertions remain the same, *i.e.*, the two timelines $Z_{ab}(f)$ and $Z_{ba}(f)$ are identical.

Elaborating: By the definition of vault transition functions, the dependencies are on the prior lock, the action, and the actual finish-time. The last also depends upon the prior lock as well as the action's duration and whether the requested start -time is actual. By the case, the prior lock is the same whether action a is applied first or second. Action durations are constant, in particular, the duration of action b is the same whenever applied. The requested start-time is actual with respect to

the initial vault $\mathit{Vault}(F_0)$ by hypothesis. Should action a be applied first then the requested start-time is conditionally actual: inasmuch as the computation of earliest start-time depends on this particular f then the condition $s \geq \mathrm{EST}_b(\mathit{Vault}(F_a))$ is preserved. (By which we mean that $\ell_0 = \ell_a$ holds for this f; if such holds for every $f \in \mathit{Depends}_b$ then its earliest start-time is uneffected by action a.) In short the dependencies of the output lock are all constant with respect to the order, so, the output lock is itself constant as well.

Recall that, by (*b), the new values are all the same: $v_b = v_{ba} = v_{ab} = R_b(f)$. Therefore, as expected, both final behaviors are oblivious to anything besides the (at most) one change wrought by b: at most the locks ℓ_0 and ℓ_b are distinct, regardless of order, so the timelines consist of the assertions from $t \mathrel{\tilde{\in}} \ell_0$ and $t \mathrel{\tilde{\in}} \ell_b$ whether a precedes b or not, and in particular $v_0 = Z_{ab}(f, t) = Z_{ba}(f, t)$ holds when $t \mathrel{\tilde{\in}} \ell_0$, while for $t \mathrel{\tilde{\in}} \ell_b$ then $v_b = Z_{ab}(f, t) = Z_{ba}(f, t)$.

Case $f \notin \mathit{Depends}_b$. Argue symmetrically with the prior case. *I.e.*, the new lock is the same whenever obtained (as b does not touch the lock), as is the new value (by (*a)), hence the final timelines are identical. Hiding in the details is the need to moreover verify that the start-time (*i.e.*, r) remains (conditionally) actual: with respect to this fluent f, the earliest start-time of action a is clearly unaltered whether action b precedes or not.

Then we come to the non-trivial case.

Case $f \in \mathit{Depends}_a \cap \mathit{Depends}_b$. As actions a and b are non-mutex the only possibility is that both read-lock: $f \in \mathit{Reads}_a \cap \mathit{Reads}_b$. Note that the intermediate timelines are not identical: $Z_a(f)$ and $Z_b(f)$ potentially differ. The reason is that read -locking a fluent extends the definition of its behavior further forward in time. So at least one of $Z_a(f)$ or $Z_b(f)$ is identical to the final behavior: whichever has the later

actual finish-time. (Alternatively all the timelines $Z_\alpha(f)$ are equal because the fluent f was already read-locked for longer than either of the actions a or b require.) For notation, by the definition of acquired read-locks, verifying which in detail establishes actualness (*i.e.*, $r = \text{AST}_a(V_0) = \text{AST}_a(V_b)$ and $s = \text{AST}_b(V_0) = \text{AST}_b(V_a)$ because $\text{EST}_a(V_0) = \text{EST}_a(V_b)$ and $\text{EST}_b(V_0) = \text{EST}_b(V_b)$), the two important locks are given by:

$$\ell_{ab} = (\text{Read-Time}(\ell_0), \max(\max(\textit{Released}(\ell_0), r + \textit{dur}_a), s + \textit{dur}_b), \texttt{True});$$
$$\ell_{ba} = (\text{Read-Time}(\ell_0), \max(\max(\textit{Released}(\ell_0), s + \textit{dur}_b), r + \textit{dur}_a), \texttt{True}).$$

As taking maximums is associative and commutative, they are equal:

$$\ell_{ba} = \ell_{ab}, \qquad \text{so trivially:}$$
$$(t \mathrel{\tilde{\in}} \ell_{ab}) \Leftrightarrow (t \mathrel{\tilde{\in}} \ell_{ba}).$$

Therefore the timelines are independent of the order of the actions, in particular:

$$(t \mathrel{\tilde{\in}} \ell_{ab}) \Rightarrow (v_0 = Z_{ab}(f, t) = Z_{ba}(f, t)).$$

The case analysis is exhaustive, so we are done but for the moreover. That the final situations are identical follows directly from demonstrating that the behaviors are identical, *i.e.*, finish by Proposition 3.10. □

Then we can prove the theorem by simply closing the lemma under induction, more or less. Some additional machinery is useful.

Definition 3.4 (Mutex-Order of a Plan)**.** The **mutex-order** of a dispatch-sequence is the partial-order given by orienting each mutual exclusion so as to be consistent with the sequence. So (a_i, t_i) is before (a_j, t_j) in the mutex-order of $(a, t)_{[n]}$ when a_i and a_j are mutex and $i < j$. More accurately, with $\prec_{\text{mutex-}X}$ denoting the mutex-order (of $X = (a, t)_{[n]}$), define:

$$(\prec_{\text{mutex-}X}) \coloneqq \text{Transitively-Close:} \left\{ \begin{aligned} &X(i),\ X(j) \mapsto i < j \mid (a_i, a_j) \text{ are mutex;} \\ &\qquad\quad \cdot\,,\cdot \mapsto \texttt{False} \end{aligned} \right\}. \tag{3.17}$$

Reordering, Permutations, Chains, etc.. Let $\sigma(X)$ denote the **reordering** of the schedule $X = (a, t)_{[n]}$ reached by permuting its dispatches in the manner dictated by the permutation σ (a permutation of $[n]$): $\sigma(X) \coloneqq \{i \mapsto X(\sigma(i))\}$. (A **permutation** is a bijection from and to its domain: $\text{Dom}(\sigma) = \text{Rng}(\sigma)$.) A (weak) **descent** in time-stamps is any index $k^{\in[n]}$ such that $t_{k-1} \geq t_k$; a (weak) **inversion** in time-stamps is any index $k^{\in[n]}$ such that $t_j \geq t_k$ for any prior index $1 \leq j < k$. A **run** in time-stamps is a subsequence, say from index k' to k, of increasing time-stamps: $t_{k'} < t_{k'+1} < \ldots < t_k$. Define a **chain** from k' to k as a run in time-stamps further satisfying that: $t_i + dur_{a_i} \leq t_{i+1}$ holds for each $i^{\in[k', k-1]}$. That is, a chain is a sequence of dispatches that would execute *sequentially* supposing actual and requested start-times coincide. In contrast define a **causal chain** as moreover satisfying that every adjacent pair of dispatches is mutex. Such surely execute sequentially:

Proposition 3.12. *The dispatch and execution orders of mutually exclusive dispatches agree. Meaning that mutually exclusive actions execute strictly sequentially, in the order dispatched.*

Corollary 3.13. *An inversion in time-stamps of an actual schedule consists of non -mutex dispatches.*

Proof. In short the locking protocol is, first of all, implemented correctly: (i) write-locks may never be concurrent with any other lock upon the same fluent. Secondly the protocol is enforced forwards through time: (ii) acquiring new locks always occurs in the future with respect to existing locks. Finally, for a pair of effects/actions/dispatches to be mutex, both must lock a common fluent, one of them employing a write-lock. Therefore the two must execute sequentially, by (i), and in the same order as dispatched, by (ii). The corollary is just the contrapositive (mutex implies agreement, so disagreement implies non-mutex).

For notation let dispatches $(a, s) = X(i)$ and $(b, t) = X(j)$ occur in that order ($i < j$) in some plan X where actions a and b are mutually exclusive: there exists a fluent f in $\mathit{Writes}_a \cap \mathit{Depends}_b \cup \mathit{Writes}_b \cap \mathit{Depends}_a$. Consider any vault-sequence $(V)_{[0,|X|]}$ of some execution of the plan. Then the task is demonstrate that b starts no sooner than a finishes: $\mathrm{AST}_{b,s}(V_{j-1}) \geq \mathrm{AFT}_{a,t}(V_{i-1})$. Always locks are held for at least the whole duration of an action execution: AST $\geq$ *Acquired* (with equality for a witness to AST = EST) and AFT $\leq$ *Released* (with equality for write-locks, for example). So it suffices to show $\mathrm{AST}_b \geq \mathit{Acquired}(\ell_b) \geq \mathit{Released}(\ell_a) \geq \mathrm{AFT}_a$ for any such appropriate pair of locks on f—let $\ell_a = V_i(f)$ and $\ell_b = V_j(f)$.

By persistence through actions not locking f it suffices to consider just actions locking f in between a and b. Then by induction through actions locking f, each of which is mutex with whichever of a and b is writing to f (perhaps both), the lock just prior to the one acquired by b, say $\ell'_a = V_{j-1}(f)$, is only more constraining (*i.e.*, its read-time and write-time are weakly greater) than ℓ_a is. (The base case

of the induction is $\ell'_a = \ell_a$ and there is nothing to show.) Precisely, say: (*w) Write-Time(ℓ'_a) $\geq$ Write-Time(ℓ_a) and (*r) Read-Time(ℓ'_a) $\geq$ Read-Time(ℓ_a).

Case $f \in Writes_b$. Then ℓ_b is a write-lock, thus by the definition of acquired write-locks: $Acquired(\ell_b)$ = Write-Time(ℓ'_a). Always Write-Time = *Released*, by definition, so: Write-Time(ℓ_a) = $Released(\ell_a)$ and Write-Time(ℓ'_a) = $Released(\ell'_a)$. Hence, which suffices, by (*w): $Acquired(\ell_b) \geq Released(\ell'_a) \geq Released(\ell_a)$.

Case $f \notin Writes_b$. So $f \in Writes_a$ is the only possibility, as the two are mutex by hypothesis. Then ℓ_a is a write-lock, hence: Read-Time(ℓ_a) = $Released(\ell_a)$. By the definition of acquired locks, acquisition times are always weakly greater than read times, so, regardless of how b locks f: $Acquired(\ell_b) \geq$ Read-Time(ℓ'_a). Therefore, which suffices, by (*r): $Acquired(\ell_b) \geq$ Read-Time(ℓ'_a) $\geq Released(\ell_a)$. □

3.2.2.3 Formal Proof of the Deordering Theorem

To **deorder** a plan is to reorder it, preserving its mutex-order. Our theorem is that deordering is an equivalence reduction; it suffices to consider at most every distinct mutex-order (which is fewer than all total-orders). Precisely:

Theorem 3.14 (Deordering)**.** *Deordering preserves corresponding behavior, actualness, and result. For notation, with* $X = (a, s)_{[n]}$ *a schedule executable from situation* F_0 *and* $Y = (b, t)_{[n]} = \sigma(X)$ *a reordering of* X *such that* $(\prec_{\textit{mutex-X}}) = (\prec_{\textit{mutex-Y}})$*:*

$$\text{Behavior}(X, F_0) = \text{Behavior}(Y, F_0), \tag{3.18}$$

$$(s)_{[n]} \geq (\text{EST}(a_i))_{i \in [n]} \iff (t)_{[n]} \geq (\text{EST}(b_i))_{i \in [n]}, \tag{3.19}$$

$$\text{Result}(X, F_0) = \text{Result}(Y, F_0). \tag{3.20}$$

Proof. Say an X-descent of schedule Y is an adjacent pair of dispatches $Y(i)$ and $Y(i+1)$ such that schedule X dispatches them in the other order: $Y(i+1) <_X Y(i)$. If such a descent were mutex, then the mutex-orders of X and Y would differ, for which there is nothing to show. So every X-descent of schedule Y is non-mutex. By Lemma 3.11, swapping any of them preserves behavior, actualness, and result (*i.e.*, final situation). Then by induction on X-inversions, the result follows. □

3.2.2.4 Significant Corollaries of Deordering/Behavior-Equivalence

What matters here is that we have decoupled search order from the flow of time. *Forward-chaining temporal planning still makes sense even if decisions are not wholly sorted in time.* In particular it suffices merely to explicitly consider only interfering actions in strictly ascending temporal order; non-interfering (*i.e.*, non-mutex) actions may be considered out of temporal order if desired. Then, for exam-

ple, we can, at any point during a forward-chaining search, slap on an independent problem and start planning for it from the initial time. (*I.e.*, we can do so without undermining the properties key to any sensible notion of "forward-chaining".) Which is *far* better than, say, merely ensuring that any given set of concurrent loading operations are considered just once. Not that the difference between $k!$ and 1 should be taken lightly; the promise of the theorem is simply far better, even, than that. That such follows is perhaps not immediately clear. Consider that we could prune down to just time-sorted schedules:

Corollary 3.15. *The behavior of a schedule is not effected by sorting the schedule in ascending order of actual start-times.*

Proof. Consider any descent in actual start-times: by Proposition 3.12, the pair are non-mutex. So by induction on the number of inversions, time-sorting preserves the mutex-order. Then the result follows by the theorem. □

Corollary 3.16. *Every schedule deorders into only causal chains. (Hence it is completeness-preserving to prune accordingly.)*

Proof. With respect to actual start-times, consider any (strict) **ascent**, the opposite of a weak descent. If non-mutex with its predecessor, then swap. Continue until impossible. First note that the process obtains a deordering. Second the result satisfies: (†) Every ascent is mutex with its predecessor.

Recall that, by Proposition 3.12, mutually exclusive dispatches execute sequentially. So concerning mutex ascents, the latter of the pair starts after the former finishes. In other words, schedules meeting (†) also, by the proposition, satisfy: Every run is a chain.

For a chain to be *causal* every adjacent pair need be mutex, which is had directly from (†). So every schedule meeting (†) also satisfies: (‡) Every chain is causal.

Then to recap: There exists a deordering of every schedule into one satisfying (‡). By the Theorem, it suffices to consider all and only such schedules. □

Specifically the Corollary justifies: It is completeness-preserving to demand that every dispatch be either (1) directly causally related to the prior dispatch, or (2) scheduled weakly *earlier* in time.

Generalizing, the Left-Shifting and Deordering Theorems together justify:

1. Every dispatch should be scheduled for *immediately* after its most recent mutex ancestor by left-shifting (Theorem 3.4).

2. Moreover, by deordering (Theorem 3.14), every dispatch should be of an action considered greater than all of its most recent non-mutex ancestors, *for any notion of "greater" whatsoever*.

Firstly this ensures we can branch over sequences of actions yet secretly be considering only plans differing in their mutex-orders. In other words we can fake something resembling Partial-Order Planning whilst forward-chaining [36, 124, 149]. Secondly all *totally-ordered* interpretations of "greatest" achieve the same pruning power. (Technically neither corollary above does sufficient tie-breaking: sets of simultaneous dispatches should be tie-broken.) As not all variations upon "greatest" are equally convenient to implement in any given context, it is worthwhile to recognize the flexibility.

3.2.3 Formal Reduction of Conservative Temporal Planning to the Multi-Objective Path Problem in State-Space

In this section we apply left-shifting and deordering in order to complete the formal reduction to the Multi-Objective Path Problem. In short: A solution is any path (or walk, but paths are always better), from the initial vertex to a goal vertex, further meeting the multi-dimensional bound (the deadlines) on the multi-dimensional costs (the current read-times of every fluent). Technically left-shifting takes care of the whole reduction; we discuss the significance of deordering following the proof.

Theorem 3.17. *Conservative Temporal Planning amounts to nothing more—and nothing less—than ascribing a (complex) notion of cost to Sequential Planning.*

For notation, let $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ *denote a Conservative Temporal Planning Problem,* $\hat{\mathcal{P}} = \left(\mathit{FluentDefs}, (\mathit{eff})_{\mathit{Actions}}, \mathit{State}_{\mathit{Initial}}, \widehat{\mathit{Goal}}\right)$ *its underlying Sequential Planning Problem (*i.e.*, throw away durations and deadlines), and* $\hat{M} = (V, E, \Sigma, R, \ell, s_0, T)$ *the state transition system of* $\hat{\mathcal{P}}$*. Then:*

$$\text{Left-Shifted}(\text{Solutions}(\mathcal{P})) = \text{First-Fit}(\text{Cost-Bounded}(L(\hat{M}))). \tag{3.21}$$

Where:

$$\text{Cost-Bounded}(L(\hat{M})) = \left\{ \ell(P = s_0, \ldots, e_n, s_n) \;\middle|\; \begin{array}{l} P \text{ } \mathit{is\ an}\ s_0\text{-}T\ \mathit{path} \text{ and} \\ \text{costs}(P) \leq \text{bounds.} \end{array} \right\},$$

$$\text{Left-Shifted}(\text{Solutions}(\mathcal{P})) = \left\{ P = (a, t)_{[n]} \;\middle|\; \begin{array}{l} \mathit{Goal}(\text{Result}(P, \mathit{Initial})) \text{ and} \\ t_i = \text{EST}(a_i) \end{array} \right\}.$$

$$V = \mathit{States}, \qquad E = \{(s, s', a) \mid s' = S'_a(s)\}, \quad R = \{(s, s', a) \mapsto (s, s')\},$$

$$\Sigma = \mathit{Actions}, \qquad \ell = \Big\{(s, s', a) \mapsto a\,;\; P = s_0, \ldots, e_n, s_n \mapsto (\ell(e_i))_{i\in[n]}\Big\}$$

$$s_0 = \mathit{State}_{\mathit{Initial}}, \quad T = \widehat{\mathit{Goal}}^{-1}(\texttt{True}),$$

$$\mathrm{costs}(P) = \text{Read-Time}(\mathit{Vault}(\mathrm{Result}(\ell(P), \mathit{Initial}))),$$

$$\mathrm{bounds} = \{f \mapsto d \mid d \textit{ is the deadline of } f \textit{ in Goal}\}.$$

Proof. Nominally there is precious little to show relative to Theorem 3.4 and Theorem 3.1. Firstly: it suffices to consider only left-shifted schedules, which are isomorphic to action-sequences, by Theorem 3.4. Secondly: action-sequences are the citizens of Sequential Planning. Hence the result is a special case of reducing such to state transition systems: Theorem 3.1. Specifically the result is special insofar as it ascribes a particular notion of cost to the plans of Sequential Planning. Naturally, said costs are just read-times with bounds as the corresponding deadlines (infinite if a fluent is unconstrained by the goal). The following greatly elaborates.

Intuitively speaking, edge-weights/edge-costs are just the durations of the actions in question. However, to make the details work out we need to also keep track of which fluents are actually effected by the action, and in what sense (read/write). So, actually, path costs are full-blown collections of locks, *i.e.*, vaults. In other words, we effect a powerful equivalence reduction on the obviously correct (but infinitely large) transition system consisting of situations as vertices by contracting together all situations with identical first component (states). Then vaults are relegated to second-class status, *i.e.*: vaults are converted into the quality/cost metric. The resulting graph remains obviously correct relative to Theorem 3.4; the information lost is easily recovered whenever desired by applying First-Fit. To make the

connection with the Multi-Objective Path Problem abundantly clear, consider the following.

Plan/Path Cost in General. Denote by $c(\cdot)$, or $g(\cdot)$ in the context of A^* [95], a **cost function** upon walks. There are precious few constraints upon cost functions in general. Usually one insists that they at least grow monotonically. So if Q extends P then $c(Q) \geq c(P)$. Sometimes it is enough that cost functions merely grow without bound: $\lim_{n\to+\infty} \min\{c(P) \mid |P| = n\} = +\infty$ [48]. The most important thing is that *minimum* be a well-defined notion; if there is some absolute lower bound on all walks, then we can ensure termination of appropriate path-finding algorithms. For our purposes we note that cost functions may be multi-dimensional, *i.e.*, write $c(P) = \left(c^{\in Costs}\right)_{\text{Dimensions}}$. In which case, comparison/minimality/*etc.* is by **Pareto-dominance**: $(a)_X \geq (b)_X \iff a_x \geq b_x$ for all $x^{\in X}$.

In practice one normally insists that if Q extends P by a single edge e, then there is some simple rule for calculating the cost $c(Q)$ of Q as an 'increase' upon the former cost $c(P)$ by a weight $w(e)$ associated with the edge e. The default is of course to sum: $c(P+e) = c(P)+w(e)$, also written $g(P+e) = g(P)+w(e)$. In general there are many possibilities besides summation, say $c(P + e) = c'(c(P), w(e))$ for some 'reasonably well-behaved' cost transition function c'.

Costs in the Reduction. For our purposes, at the most detailed level, let costs be the vaults resulting from executing the corresponding plans. More specifically say that costs increase by applying earliest vault transition functions. Then for notation, transliterate the computation of earliest start-times into the vocabulary of cost

functions as:

$$w(e = (s, s', a)) \coloneqq (Reads_a, Writes_a, dur_a), \tag{3.22}$$

$$c((e)_{[n]}) \coloneqq c'(c((e)_{[n-1]}), w(e_n)), \tag{3.23}$$

$$c'(c((e)_{[n-1]}), w(e_n)) = V'_{a(e_n)} \circ V'_{a(e_{n-1})} \circ \cdots \circ V'_{a(e_1)}((-\infty, 0, \texttt{False})_{Fluents}); \tag{3.24}$$

that is, with:

$$\begin{aligned} c = c((e)_{[n-1]}), \quad & w = w(e_n), \\ c_f = (x, y, T), \quad & w = (R, W, d), \text{ and} \\ \text{EFT} = d + \max \text{Read-Time}(c(R)) \cup & \text{ Write-Time}(c(W)), \end{aligned}$$

then:

$$c'(c, w) \coloneqq \left\{ \begin{array}{l} f \mapsto c_f \mid f \notin R \cup W; \\ f \mapsto (y, \text{EFT}, \texttt{False}) \mid f \in W; \\ f \mapsto (x, \max(y, \text{EFT}), \texttt{True}) \mid T \text{ and } f \in R; \\ f \mapsto (y, \text{EFT}, \texttt{True}) \mid \neg T \text{ and } f \in R \end{array} \right\}. \tag{3.25}$$

Remark 3.1. Which, we note, defines a monotone (multi-dimensional) cost function. That, however, is only the lower bar upon 'politeness'. It is important to realize that the cost function above is *far* better behaved than mere monotonicity. In particular it is useful to note that computing the cost of the extension of a path by an edge is constant-time: c', above, is constant-time. So for all that the notation may be imposing: the update rule is nonetheless properly regarded as *simple*.

Bounds in the Reduction and the Subtlety of Disjunctive Deadlines. Technically we allow goals to be disjunctive. Consider the goal "Have laundry done by 11am and rooms cleaned by noon, or *vice versa*."; so we cannot actually give a unique deadline to either task without first picking which of the two possibilities to satisfy. The standard manipulation suffices; so pretend the goal expression is limited to a simple conjunction of primitive deadline goals. Then denote the collection of the deadlines over each such primitive deadline goal by: $(\text{bounds})_{\mathit{Fluents}}$. (With $\text{bounds}_f = \infty$ if f lacks a deadline.) Likewise denote throwing away all the bookkeeping surrounding costs by: $\text{costs}(P) = \text{Read-Time}(c(P))$. So $\text{costs}(P) \leq \text{bounds}$ is another way of writing the definition of satisfaction of deadlines, Equation (2.10).

(The Standard Manipulation:) Compile testing of disjunctive deadlines into additional one-way paths of the graph in front of the goal vertices proper. In particular, explode the goal expression out into disjunctive normal form and setup a separate goal-test action/edge for each disjunct; the explosion is 'harmless' in context as we have (nominally) already paid the price of writing out every state. (In practice though disjunctive deadlines would surely go very far towards ruining the effectiveness of domain-independent heuristics.) Have these special goal-test actions subtract the relevant disjunct-specific deadlines from each acquisition time and release time of a fluent. Regarding the example: subtracting 11 hours from clean-laundry and 12 hours from clean-rooms is the effect of the weight upon one edge leaving the original set of goal vertices, the other subtracts 12 hours from clean-laundry and 11 hours from clean-rooms. Recall that negative weights are unproblematic if they can only be collected once, *i.e.*, when cycles are of no concern. Then we can legitimately assert a single/global multi-dimensional bound upon cost: all zeros. A similar manipulation addresses non-uniform initial locks.

Summarizing: There is a touch of subtlety to the details that would likely end up being practically significant, particularly if deadlines are set very tightly. Which is interesting. Details aside though, the reduction is merely the transliteration of Theorem 3.4 into Graph Theory. □

Significance of Deordering to the Reduction. The motivation to take the reduction rather more seriously (*i.e.*, as closely related to practical approaches to performing Conservative Temporal Planning) is, or is closely related to, deordering. That is, we would much prefer to reduce to the Single-Objective Path Problem if possible. There it suffices, for optimality/completeness, to track single best known paths to each state, rather than Pareto-sets of best known paths. Then results serving to more efficiently recognize dominated or equivalent possibilities can be seen as something like making progress on reducing to the simpler setting of a single objective.

In practice the significance has more to do with how much quality one expects to lose by applying an incomplete/suboptimal approach instead. For example, if we desire neither explicit deadlines, nor implicit deadlines in the form of demanding optimality, then a solution is a mere initial vertex to goal vertex path. In this case Conservative Temporal Planning does reduce perfectly to the view upon Sequential Planning taken by the state-of-the-art classical planners. Which is far from a random hypothetical:

- the benchmarks lack deadlines,
- optimality is optional by definition in the satisficing tracks, although making some attempt at quality is certainly worthwhile, and

- the optimal version of the temporal track has never in fact taken place—only CPT even tries [202].

Plainly, so far as the present benchmarks are concerned: One need not even turn off duplicate state elimination. (Albeit, it is not a foregone conclusion that leaving it on will always do better.)

Hence it is, at present, difficult to say how large the significance of deordering may be; the benchmarks do not force us to consider the one way in which Conservative Temporal Planning is legitimately different from Classical Planning. Then, at this point in the dissertation, it is perhaps best said that the significance of deordering is largely meta. (Later, we shall have greater use for deordering.) In particular, the more practically significant we take deordering to be—the further apart we imagine Conservative Temporal Planning and Classical Planning to be. Conversely, the greater we take deordering to be of strictly theoretical significance, then the closer the two forms of planning become. Whether we view these as same or different has all sorts of interesting real-world consequences. For example: Should there even be separate tracks for them in the International Planning Competition?

3.2.4 Discussion: Conservative Temporal Planning is Simple(st)

So what have we proven?

- The precise time-stamps are irrelevant, by left-shifting.
- The qualitative model of time remaining is no more expressive than the partial -orders of Partial-Order Planning, by deordering.
- Finding best solutions is computationally equivalent to an NP-complete problem in state-space, by the reduction.

With respect to "Classical Planning":

- Older interpretations of "Classical Planning" sometimes consider the NP-complete variations on path-finding.

  For example, finding best partially-ordered plans constitutes a multiobjective problem in state-space.

- Modern interpretations of "Classical Planning" always face the poly-time variants.

  Even merely sum of edge weights is sometimes considered beyond classical planners; sum of edges weights is enough to capture the "Net Benefit" fragment of "Partial-Satisfaction Planning".

It is difficult to take the computational complexity class result entirely seriously in any truly practical context. If we are really considering, individually, so many vertices of state-space that its computational properties are relevant, *we have already failed*, in any practical context, for its computational properties are already beyond atrocious.

Elaborating, if some application demands optimality (or anything else forcing exploration of fractions of state-space), then it must be actually possible to somehow find optimal answers for that application. Whatever the domain knowledge is that converts the miraculous into the feasible is, clearly, knowledge that cannot be ignored. Who can say what it will take to make good on such knowledge? Will state-space really be appropriate? Will domain-independent techniques such as planning graphs remain relevant—or be entirely overshadowed? These are difficult questions.

In contrast, an easy question: Is Time Important? If we are going to seriously discuss applications, then, clearly: the system as a whole will need to do *something* with temporal reasoning. If even the simplest kinds force us to consider the complexities of Multi-Objective Search, then that is simply the lower bar for most realistic applications. In which case pining for Single-Objective Search is simply wishful thinking. The real world requires complex tradeoffs in multiple dimensions of quality and that is really just the end of the story. (Or is it the beginning? In any case, taking the issue of quality seriously forces multiple objectives.)

So in other words, realistic side-by-side evaluations/comparisons of Conservative Temporal Planning and Classical Planning need to further specify, *at least*, how the latter is to be wrapped (or otherwise easily modified) so as to deal with time in some less-than-ludicrous fashion. That at least puts both equally close/far from real world problems.[9]

[9] Yet more realistic evaluations would further specify how both are to be wrapped by techniques for addressing uncertainty (*etc.*), but there are too many such dimensions of real-world significance. One hopes, *i.e.*, the basic premise of the research agenda supposes, that the technical issues are orthogonal 'enough' in general. In particular we hope that conclusions with regards to temporal reasoning wrapped inside of, say, a replanning approach will be similar enough with regards to temporal reasoning wrapped inside of, say, a limited contingency approach to dealing with uncertainty. Note that FFReplan won the first probabilistic planning competition.

It is hard to imagine any more simple-minded method of dealing with Classical Planning's lack of a realistic approach to time than applying First-Fit. One could do that as a one-shot post-processing step: winning the 2008 temporal track [105]. Or one could wrap it inside of some anytime approach to planning, post-processing multiple times: winning the 2006 and 2011 temporal tracks [29, 56]. One-upping an anytime approach is to throw some amount of temporal reasoning into the heuristic itself ('post'-processing per search node): perhaps, then again perhpaps not, besting every temporal track winner for the last decade [87].[10] The techniques continue on [17].

Is the problem ever changing? It seems all of these are but usefully distinct approaches to a common problem. Should we call it Classical Planning? Sequential Planning? Conservative Temporal Planning? Whatever we name it: The technical challenges remain the same. That is, the degree to which technical challenges are simpler in Classical Planning seems to us to be due only to ignoring them.

It should be said that such is an entirely valid approach to research. At the same time, practical applications cannot wish away issues. The lesson, then, is that meaningful empirical evaluation of so-called "domain-independent" planners needs to be especialy careful regarding time. Specifically, when faster is better we must realize that classical planners, despite appearances, 'already are' a reasonable enough approach. That is, some trivial modification really ought to be taken as a baseline. Otherwise the claim that we are making empirical progress holds no water.

[10]Albeit, that LPG is relatively fast concerns chiefly its use of local search techniques. Note that LPG's absence should not be interpreted as admission of defeat.

However, to avoid provoking too much argument, rather than simply say that one is just the other in disguise, or that the differences are but illusions, *etc.*, we shall summarize thusly.

Summary. We have proven that Conservative Temporal Planning formally captures what appears to be the computationally simplest kind of (nondegenerate) Temporal Planning conceivable. It is, at the very most, a small step past Classical Planning, but that is neither here nor there in the *grand* scheme of things. Nonetheless at the technical level the precise relationship matters greatly: leveraging the state-of -the-art in Classical Planning is a great strategy. Then insofar as future work is concerned, we have attempted to contribute in this section by considering: a very detailed account of what the precise relationship is, limited examples of what does and does not readily transfer, and, to a certain extent, why.

Concerning the rest of the dissertation, the key observation complementing the relative simplicity of Conservative Temporal Planning is just:

Conservative Temporal Planning, despite deadlines, does not ever nontrivially *require concurrency*.[11]

The thesis is that the relationship is causal: Conservative Temporal Planning enjoys relative computational simplicity *because* it rules out required concurrency.

[11] The only concurrency ever required in Conservative Temporal Planning is for the trivial reason of simply having deadlines. Deadlines are not trivial themselves, but the concurrency forced by deadlines efficiently reduces to reasoning about mere action-sequences.

3.3 INTERLEAVED TEMPORAL PLANNING: SLACKLESS RESCHEDULING, BEHAVIOR EQUIVALENCE BY DEORDERING, AND REDUCTION TO STATE TRANSITION SYSTEMS

Here:

1. We demonstrate that much of the rescheduling flexibility of Conservative Temporal Planning is retained, *i.e.*, we adapt the Left-Shifting Theorem (Theorem 3.4). So, as desired, *time* remains largely qualitative.

2. We borrow/generalize the Deordering Theorem (Theorem 3.14) for free. The result is *crucial* to the thesis. Specifically, the formalization of 'trivial' versus 'nontrivial' reasons for requiring concurrency appeal to the theorem.

3. We brute-force reduce Interleaved Temporal Planning to Sequential Planning. The reduction illustrates that the technical challenge, relative to classical planning, is what it is supposed to be: reasoning about time.

3.3.1 Rescheduling: Dominance of Slackless Schedules

The intuition here is that we should still be able to exploit the independence of effects from the precise times at which they take place. (If not, we may as well allow the dependency!) So it remains the case that effects may be rescheduled relatively freely. The new constraint is that the multiple effects of an action all must occur at fixed offsets from one another in time. To take care of this constraint we generalize to Simple Temporal Networks [47].

So to begin: The new reason that some effect may be unable to start earlier is that some future effect of the same action is more directly unable to do so. Then we have two notions of earliest: earliest with respect to just the current situation (the

direct reason), and earliest with respect to the plan as a whole (the indirect reason). Say an effect is slackless if it is impossible to make it start any sooner. An action is slackless if any of its effects are. Formally:

Setup. Let $X = (a)_{[n]}$, the 'plan', be a sequence of effects, *Initial* be an initial situation, and consider all ways of scheduling X:

$$\mathcal{Z} = \left\{ Y = (a, t)_{[n]} \mid Y \text{ is executable from } \textit{Initial} \right\}. \tag{3.26}$$

Then say that the **earliest actual start-time** of effect i is the least start-time t_i it could receive in any possible scheduling of the plan:

$$\text{EAST}_{\mathcal{Z}}(i) \coloneqq \min \{ t_i \mid (a_i, t_i) \in Y \in \mathcal{Z} \}. \tag{3.27}$$

Define the **global slack** of dispatch i in an effect-schedule $Y = (a, t)_{[n]}{}^{\in \mathcal{Z}}$ as the difference between the given start-time and the earliest possible:

$$\text{Slack}_Y(i) \coloneqq t_i - \text{EAST}_{\mathcal{Z}}(i) \qquad \text{(Global).} \tag{3.28}$$

Let $\mathcal{Z}(i)$ denote the schedules globally slackless at dispatch i:

$$\mathcal{Z}(i) = \left\{ Y^{\in \mathcal{Z}} \mid \text{Slack}_Y(i) = 0 \right\}. \tag{3.29}$$

Achieving an entirely slackless schedule in the global sense should be 'incredible'. In contrast, the **local slack** of an effect is the difference between its given start-time and the earliest that the locking protocol would have locally permitted:

$$\text{Slack}_Y(i) := t_i - \text{EST}_{a_i}(V_{i-1}) \qquad \text{(Local).} \tag{3.30}$$

That is, the intuition is to make sure that at least one effect of every action has a local slack of 0. (Achievability of which should be at least plausible, if not obvious.) For notation, with $a_i = (\alpha, \texttt{all})$ serving to stand for the whole, then minimize:

$$\text{Slack}_{Y,\alpha}(i) := \min \left\{ \text{Slack}_Y(j) \mid a_j = (\alpha, x) \text{ with minimal } j \geq i \right\}. \tag{3.31}$$

Ensuring that every action is locally slackless would be a local maxima of rescheduling. Intuitively speaking there is just one maxima. Technically the local notion permits 'bootstrapping'; we could have sets of actions reschedulable earlier *en masse*, but no single one can be rescheduled earlier without breaking executability. As the global notion exists and is easily computed we shall typically ignore such. So:

Theorem 3.18 (Eliminating Slack is Complete)**.** *Let $\mathcal{Z}$ be the set of all executable schedulings of an effect-sequence and let $\mathcal{Z}(i)$ denote the subset slackless at dispatch i. Assume the effect-sequence is executable under some scheduling: $\mathcal{Z} \neq \emptyset$. There exists an entirely slackless member:*

$$\bigcap_i \mathcal{Z}(i) \neq \emptyset. \tag{3.32}$$

The result readily reduces to a property of Simple Temporal Networks [47]. We setup the STN, reiterate from first principles the proof of the corresponding property, and finally restate and prove the theorem.

Definition 3.5. The Simple Temporal Network of an effect-sequence consists of:

- two vertices per effect, one for its start-time and one for its finish-time,
- pairs of weighted edges modeling the corresponding duration constraints, and
- 0-weighted edges modeling the mutex-order of the effect-sequence.

The following, up to the lemma, elaborates the definition in great detail. Let $X = (a)_{[n]}$ denote the sequence of effects in question. Let $start_i$ and end_i be two vertices per index; say $\mathcal{S} \cup \mathcal{E}$ denotes all the vertices. Let index 0 denote the initial situation. Assume the initial situation finishes at time 0 (force $t_0 = 0$).[12] (Likewise take dur_{a_0} to be $-t_{-\infty}$.)

Optionally consider deadline goals by modeling them as special effects. Say there are m of them; let a_{-i}, for $i \in [m]$, stand for each of the deadline goals. Then for each, *i.e.*, with (f_i, v_i, t_i) and $i^{\in[m]}$, let: $W_{-i} := \emptyset$, $R_{-i} := \left\{ end_{j \in [0,n]} \mid f_i \in Writes_{a_j} \right\}$, and $dur_{a_{-i}} = t_i$. Ensure the last, the deadlines, are interpreted correctly by identifying $start_0 = start_{-1} = \cdots = start_{-m}$.

Rather than say there are 6 vertices per (execution of an) action identify those required to be simultaneous. *I.e.*, identify $start_i = start_j$ for any α, i, and least $j > i$ satisfying $a_i = \texttt{all-}\alpha$ and $a_j = \texttt{bgn-}\alpha$. (Presumably the start-part occurred

[12] Compile non-uniform initial situations into several actions implementing the initialization [72]. Also, normalize the earliest such release time to value 0 by adding/subtracting some constant. Here normalization is easy because action definitions cannot access the absolute value of time; in more general contexts normalization involves surgery on action definitions to dynamically determine the denormalized value.

immediately: $j = i + 1$.) Likewise identify $end_i = end_j$ for any α, i, and least $j > i$ satisfying $a_i = \texttt{all-}\alpha$ and $a_j = \texttt{fin-}\alpha$. Reducing to one vertex per action is also possible, as all parts are at deterministic offsets from one another. This way is just to simplify the presentation of the constraints.

Then let $\mathcal{D}$ denote the duration constraints in the form of pairs of directed weighted edges, *i.e.*, as pairs of inequalities enforcing the desired equalities:

$$\mathcal{D} \coloneqq \left\{ \begin{aligned} &(start_i,\ end_i,\ -dur_{a_i}) \mid i \in [-m, n]; \\ &(end_i,\ start_i,\ +dur_{a_i}) \mid i \in [-m, n] \end{aligned} \right\}.$$

Moving on to the precedence constraints, let $\mathcal{P}$ model the mutex-order of the plan as directed 0-weighted edges, with R_i and W_i being the finish-times in question:

$$\mathcal{P} \coloneqq \Big\{(end_j,\ start_i,\ 0) \mid end_j \in R_i \cup W_i \text{ and } i \in [-m, n]\Big\}.$$

Specifically, and somewhat more accurately, compute the vertices corresponding to the finish-times delaying action i's aquisition of locks as:

$$\begin{aligned} R_i &\coloneqq \Big\{end_j \mid Reads_{a_i} \cap Writes_{a_j} \neq \emptyset \text{ and } (i < 0 \text{ or } j < i)\Big\}, \\ W_i &\coloneqq \Big\{end_j \mid Writes_{a_i} \cap Depends_{a_j} \neq \emptyset \text{ and } (i < 0 \text{ or } j < i)\Big\}. \end{aligned}$$

Then define the **corresponding Simple Temporal Network** as:

$$\begin{aligned} &\mathrm{STN}(X) \coloneqq (V = \mathcal{S} \cup \mathcal{E}, E = \mathcal{D} \cup \mathcal{P}, R, w), \\ &\quad \text{with } R = \{(x, y, \cdot) \mapsto (x, y)\}, \\ &\quad \text{and } w = \{(\cdot, \cdot, x) \mapsto x\} \text{ merely serving to formally pull apart edges.} \end{aligned}$$

The meaning of an STN is in terms of what assignments of times to its vertices are considered to satisfy it. Say $\tau^{\in \mathcal{S} \cup \mathcal{E} \to \mathbb{Q}}$ is such a candidate assignment of times. (Force $\tau(start_0) = 0$, *i.e.*, pin the initial situation to the right time.) The meaning of a directed weighted edge $e = (u, v, w)^{\in \mathcal{D} \cup \mathcal{P}}$ is that the constraint $\tau(u) - \tau(v) \leq w$ should hold. If so, then the candidate **satisfies** that edge. If the candidate satisfies every edge, then it satisfies the whole network. The 'intended' usage is that edges point to successors in time and negated weights denote minimum separations: $\tau(v) - \tau(u) \geq -w$. Also an edge may point to a predecessor in time and the weight then denotes a maximum separation (which is how the notation reads). Then:

Lemma 3.19. *The set of executable schedules of a given sequence of effects is all*[13] *and only the schedulings permitted by its corresponding Simple Temporal Network.*

Proof. Continue with the notation from above, excluding consideration of deadline goals ($m = 0$), because such have nothing to do with executability. Consider the scheduling $Y = (a_i, t_i = \tau(start_i))_{i \in [n]}$ of a given effect-sequence X by a candidate solution τ to its corresponding Simple Temporal Network STN(X). Then the claim is that executability of Y is equivalent to satisfaction of STN(X) by τ. It suffices to demonstrate that each of the underlying sequences of an execution are all defined (iff the STN is solvable). Only the underlying debt-sequence and vault-sequence are relevant. That is, counter-intuitively, because scheduling does not change the order that effects are interpreted in, it is harmless to assume:

(S) The underlying state-sequence is defined, in either direction.

(Claim) The choice of $\mathcal{D}$, and the identification of those vertices required to be simultaneous, correctly encodes verification of durations and obligations. That is,

[13] Ignore sequences of effects never executable, *i.e.*, due to failing the state-sequence constraints.

firstly, start-times and finish-times mean what they are supposed to: (0) To satisfy $\mathcal{D}$ means that $\tau(start_i) + dur_{a_i} = \tau(end_i)$ holds for every i, so effect durations are encoded correctly. With that prerequisite fulfilled, then note that the three constraints verified by the machinery of debt-sequences are likewise fulfilled, as follows. (1) The ordering of parts has not changed, so still the parts of every action occur in the right order. (2) The construction identified $start_i = start_j$ for i and j the indices of the all-part and start-part of (the execution of any given) action, so all-parts and start -parts start at the same time as required. (3) The construction identified $end_i = end_j$ for i and j the indices of the all-part and end-part (*etc.*), so all-parts and end-parts finish at the same time as required. Incidentally, we need (0) most specifically in order to ensure that finish-times translate correctly in either direction, *i.e.*, so that the details of (3) hold. That is, the literal constraint of the STN implied by the identification of the vertices $end_i = end_j$, and the then four relevant edges/inequalities, is: $\tau(start_i) + dur_{\texttt{all-}\alpha} = \tau(start_j) + dur_{\texttt{fin-}\alpha}$ (with $\texttt{all-}\alpha = a_i$ and $\texttt{fin-}\alpha = a_j$). From the machinery side the equivalent constraint is more literally written (*i.e.*, the calculations are of both sides of): $t_i + dur_{\texttt{all-}\alpha} - dur_{\texttt{fin-}\alpha} = t_j$, where equivalence follows by recalling $t_i = \tau(start_i)$ defines the mapping between executable schedules and solutions to the STN. So:

(D) The duration constraints are satisfied iff the debt-sequence exists.

(Claim) The constraint that dispatch-times be <u>actual</u> is met by the choice of $\mathcal{P}$. That is, the construction of $\mathcal{P}$ guarantees that any dispatch i required to follow dispatch j in time does so. Specifically, by the construction of R_i and $\mathcal{P}$, the constraint $\tau(end_j) \leq \tau(start_i)$ exists for every dispatch j writing to a fluent read from by some later dispatch i; in fact only the last such j is relevant, and this last is the *establisher* of that precondition at i being fulfilled, but the definition is correct (shown

below) whether one maximizes j or not. (Not maximizing j is analogous to Chapman's white knight condition [28].) Likewise, by the construction of W_i and $\mathcal{P}$, the constraint $\tau(end_j) \leq \tau(start_i)$ exists for every dispatch j being clobbered/threatened by dispatch i. Here the whole set W_i is relevent, at least, that subset which concern read-locks is. That is because the last read-lock to expire can change under rescheduling (*i.e.*, the prior dispatch most delaying dispatch i is unclear from the sequence alone). So in short, the precedence constraints faithfully encode the definition of earliest start-times.

For notation, satisfaction of:

$$\begin{aligned} t_i &\geq \mathrm{EST}_{V_{i-1}}(a_i), \\ &\geq \max \text{Read-Time}(V_{i-1}(Reads_{a_i})) \\ &\quad \cup \text{Write-Time}(V_{i-1}(Writes_{a_i})), \end{aligned} \qquad \text{and} \qquad \text{(a)}$$

$$0 \geq \tau(end_j) - \tau(start_i) \text{ for all } end_j \in R_i \cup W_i, \qquad \text{are equivalent;} \qquad \text{(b)}$$

Because (b) rearranges to:

$$\begin{aligned} t_i &= \tau(start_i), \\ &\geq \max_{end_j \in R_i \cup W_i} \tau(end_j), \end{aligned}$$

which expands to (a) by the definition of acquired locks:

$$\begin{aligned} \max_{end_j \in R_i \cup W_i} \tau(end_j) = \ & \max \{\text{Read-Time}(V_{i-1}(f)) \mid f \in Reads_{a_i}\} \\ & \cup \{\text{Write-Time}(V_{i-1}(f)) \mid f \in Writes_{a_i}\}. \end{aligned}$$

Hence satisfaction of $\mathcal{P}$ implies that start-times are actual:

$$\tau(start_i) \geq \mathrm{EST}_{V_{i-1}}(a_i).$$

For yet greater detail regarding the appeal to acquired locks, consider:

$$\begin{aligned}
t_w &:= \max \text{Write-Time}(V_{i-1}(\mathit{Writes}_{a_i})), \\
&= \max \mathit{Released}(V_{i-1}(\mathit{Writes}_{a_i})) \qquad \text{by definition,} \\
&= \max \left\{ \mathrm{AFT}_{V_{j-1}}(a_j) \;\middle|\; \begin{array}{l} j < i \text{ and} \\ f \in \mathit{Writes}_{a_i} \cap \mathit{Depends}_{a_j} \text{ with} \\ k \leq j \text{ max subject to } f \in \mathit{Writes}_{a_i} \cap \mathit{Writes}_{a_k} \end{array} \right\},
\end{aligned}$$

by induction on the definition of acquired locks. Base case: The last write-lock on f. Induction: Through changes to the lock. By the same induction, we can skip explicitly finding the greatest such k, *i.e.*, the last finish-time in the sequence is necessarily greatest, so:

$$\begin{aligned}
&= \max \left\{ \mathrm{AFT}_{V_{j-1}}(a_j) \mid j < i \text{ and } f \in \mathit{Writes}_{a_i} \cap \mathit{Depends}_{a_j} \right\}, \\
&= \max \left\{ \tau(end_j) \mid j \in W_i \right\} \qquad \text{by the definition of } W_i. \qquad \text{(w)}
\end{aligned}$$

Similarly for read-locks:

$$\begin{aligned}
t_r &:= \max \text{Read-Time}(V_{i-1}(\mathit{Reads}_{a_i})), \\
&= \max \left\{ \text{if } \mathit{Readable}(\ell) \text{ then } \mathit{Acquired}(\ell) \text{ else } \mathit{Released}(\ell) \mid \ell = V_{i-1}(f^{\in \mathit{Reads}_{a_i}}) \right\},
\end{aligned}$$

$$= \max \left\{ \mathit{Released}(V_j(f^{\in Reads_{a_i}})) \;\middle|\; \begin{array}{l} j < i \text{ max subject to} \\ \mathit{Readable}(V_j(f)) = \texttt{False} \end{array} \right\},$$

because (*i.e.*, by a similar easy induction argument) the acquisition-time of a read-lock is just the release-time of the last write-lock preceding it. Hence:

$$= \max \left\{ \mathrm{AFT}_{V_{j-1}}(a_j) \mid j < i \text{ and } f \in \mathit{Reads}_{a_i} \cap \mathit{Writes}_{a_j} \right\},$$

$$= \max \left\{ \tau(end_j) \mid j \in R_i \right\}. \qquad \text{(r)}$$

Therefore, as (a) is just $\max(t_r, t_w)$ and that in turn rewrites to (b) by (r) and (w), the claim is shown. Namely, all the dispatch-times are actual. As that is the only constraint upon the vault-sequences of executions that need be satisfied:

(P) The vault-sequence exists iff the precedence constraints are satisfied.

Then finally, as $\mathcal{D}$ and $\mathcal{P}$ are the entirety of the edges of the corresponding Simple Temporal Network, by (S), (D), and (P): Assume executability and conclude satisfaction, or assume satisfaction and conclude executability. □

So the theorem reduces to a well-known result about solving Simple Temporal Networks—they are, indeed, simple. Precisely, solving a Simple Temporal Network comes down merely to the negatively weighted variation on the Shortest-Path Problem. In detail:

Lemma 3.20. *Satisfaction of a Simple Temporal Network is equivalent to checking for absence of negative-weight cycles. Moreover, the optimal assignment is that computed by setting the time of every vertex to the duration of its critical path.*

Proof. Say $\mathcal{T}$ denotes all τ satisfying the constraints of a Simple Temporal Network with initial vertex s.

Observe transitivity: if e_1 and e_2 connect x to z through y, then for $\tau \in \mathcal{T}$ to hold means that $\tau(x) - \tau(y) \leq w(e_1)$ and $\tau(y) - \tau(z) \leq w(e_2)$ do, so furthermore their sum $\tau(x) - \tau(z) \leq w(e_1) + w(e_2)$ holds. Then by induction the same holds of every walk: $\tau(x) - \tau(y) \leq \sum_{e \in Z} w(e)$ for any walk Z.

For a cycle the constraint reduces to $0 \leq \sum w$. So negative-weight cycles guarantee unsatisfiability. Then only the the satisfiable half of the claim remains, *i.e.*, that absence of negative-weight cycles guarantees satisfiability. Specifically, indeed assuming their absence, it suffices to demonstrate the moreover.

Let $w_x = \min \sum_{e \in\, s\text{-}x \text{ walk}} w(e)$ denote the minimum weight of a walk to x from the initial vertex. Note that x cannot possibly be scheduled sooner than $-w_x$ by transitivity (recall $\tau(s) = 0$): $\tau(s) - \tau(x) \leq w_x$. Further note, by the absence of negative cycles: such walks are paths. Hence the witnesses are called **critical paths**. In general it is egregiously optimistic to simultaneously assign all variables their greatest lower-bounds. (The Moreover:) For STNs it *does* suffice to take $\tau^*(x) = -w_x$.

For a contradiction suppose otherwise. That is, suppose there exists an edge $e = (u, v, w)$ such that $\tau^*(u) - \tau^*(v) \leq w$ is false. Then $\tau^*(u) - \tau^*(v) > w$, which can be written: (†) $-\tau^*(u) + w < -\tau^*(v)$. Recall that the lightest walk from s to u has weight $-\tau^*(u) = w_u$, and that the lightest walk from s to v has weight $-\tau^*(v) = w_v$. Then the weight of one particular walk—*i.e.*, through u—from s to v is $w_u + w = -\tau^*(u) + w$. Plugging into (†) yields the desired contradiction, $w_u + w < w_v$, *i.e.*, there cannot be a lighter-than-lightest walk to v. Then by virtue of contradiction: τ^* is a solution. □

So finally:

Theorem (Eliminating Slack is Complete)**.** *Let $\mathcal{Z}(i)$ denote the set of executable schedules, slackless at dispatch i, of some effect-sequence. A slackless executable schedule exists: $\bigcap_i \mathcal{Z}(i) \neq \emptyset$. (Assume $\mathcal{Z} \neq \emptyset$.)*

Proof of Theorem 3.18. Let $X = (a)_{[n]}$ be the given effect-sequence and STN(X) the corresponding STN, which is solvable by assumption. By Lemma 3.19, the executable schedules of X are all and only the solutions to STN(X). By the moreover of Lemma 3.20, the optimal solution to STN(X) gives every dispatch its earliest conceivable time. (*I.e.*, $(a_i, \tau^*(start_i))_{i\in[n]} \in \bigcap_i \mathcal{Z}(i)$ is said optimal solution.) In other words, setting the dispatch-time of every effect to the duration of its greatest-duration critical path yields the (globally) slackless scheduling of any given effect-sequence. □

Discussion. The result is not, of course, surprising. It is interesting to note how fragile it is, though. Consider rescheduling of commutative but nonetheless mutually exclusive effects: scheduling of classroom usage, for example. Then it is certainly impossible to dispatch every effect at its earliest conceivable time. (In such a setting one would surely define slack by the local notion, as the global notion would be useless.) At the technical level the scheduling problem then generalizes to Disjunctive Temporal Networks instead. Which are not simple.

It is also interesting to note that deadline goals are largely irrelevant to scheduling considerations. Including them could render an STN unsolvable; but besides that they have no impact. This is because the optimal solution to the STN is the same whether the deadlines are included or not. If it were possible to request that goals be accomplished by a certain time, but held true only for some finite interval after that, then including such ("temporally extended goals") could certainly impact

scheduling considerations. That is, then it would be possible for goals to participate in critical paths. So, oddly enough, deadlines are most relevant to planning-level considerations. Which is because unsolvability of the corresponding STN is certainly grounds for pruning; adding further vertices and edges will never change that the STN is unsolvable.

Moving on, we come to formalizing reordering for Interleaved Temporal Planning, which, in contrast with rescheduling, amounts to little more than borrowing from Conservative Temporal Planning.

3.3.2 Reordering: Deordering and Behavior-Equivalence

The generalization considered by Interleaved Temporal Planning has precious little impact upon deordering dispatch-sequences. That is because all reordering specifically leaves dispatch-times alone (converse to rescheduling leaving the order alone). As the 'only' new constraint concerns those, then the theorem transfers more or less directly from Conservative Temporal Planning. We sketch the proof.

The various definitions remain essentially the same, keeping in mind that effects/primitives here are the actions of simpler forms of planning.

Definition 3.6 (Mutex-Order/Deordering of a Plan)**.** Two effect-dispatches are **mutually exclusive** (mutex for short) if the underlying effects write to the other's dependencies. The **mutex-order** of an effect-schedule is the partial-order given by orienting each mutual exclusion so as to be consistent with the sequence in which the effects are dispatched. So (a_i, t_i) is before (a_j, t_j) in the mutex-order of $(a, t)_{[n]}$ if a_i and a_j are mutex and $i < j$. For notation let $\prec_{\text{mutex-}X}$ denote the mutex-order (of $X = (a, t)_{[n]}$), defined by:

$$(\prec_{\text{mutex-}X}) \coloneqq \text{ Transitively-Close: } \left\{ \begin{array}{c} X(i),\ X(j) \mapsto i < j \mid (a_i, a_j) \text{ are mutex;} \\ \cdot\,,\cdot \mapsto \texttt{False} \end{array} \right\}.$$

Define the **deorder** $\prec_{\text{deorder-}X}$ by additionally stipulating that effects of the same action be ordered correctly. With $\alpha(a) = \alpha'$ when $a = (\alpha', \cdot)$ denoting the action owning the effect a:

$$(\prec_{\text{deorder-}X}) \coloneqq \text{ Transitively-Close:}$$

$$\left\{\begin{array}{c} X(i), X(j) \mapsto i < j \mid \alpha(a_i) = \alpha(a_j) \text{ or } (a_i, a_j) \text{ are mutex;} \\ \cdot\,,\cdot \mapsto \texttt{False} \end{array}\right\}. \quad (3.33)$$

Recall:

Definition (Behavior)**.** An execution's **corresponding behavior** is given by:

$$\text{Behavior}(X) := \{(f,t) \mapsto \mathit{State}_Y(f) \mid t \mathrel{\tilde{\in}} \mathit{Vault}_Y(f) \text{ and } Y \in \text{Rng}(X)\}.$$

With:

$$(\tilde{\in}) := \{(t,\ell) \mid t \in \text{if } \mathit{Readable}_\ell \text{ then } [\mathit{Acquired}_\ell, \mathit{Released}_\ell] \text{ else } \{\mathit{Released}_\ell\}\}.$$

Theorem 3.21 (Deordering)**.** *Deordering preserves corresponding behaviors. Moreover, deordering preserves result-equivalence.*

Proof Sketch. Argue by reduction to the proof of Theorem 3.14, Page 128. There are two key changes from Conservative Temporal Planning.

The first change is to restrict vault transition functions to be defined only for actual dispatch-times. That is irrelevant, as we may freely restrict Theorem 3.14 to actual schedules.

The second change is to add debts to the structure of situations. That directly impacts the notions of result and execution. It also indirectly impacts the notion of corresponding behavior, but only in the trivial sense of altering the structure of its domain. In other words, note: the mapping rule (*i.e.*, $\tilde{\in}$) remains the same. So Proposition 3.10 holds in the sense that behavior-equivalence nets us final state and

final vault equivalence. Then it remains to net final debt equivalence and to reduce the notion of execution.

Final debt equivalence is straightforward, as follows. Note that debt transition functions operate only on a per-compound basis: each updates only the structure associated with its compound. Deordering does not add, remove, or reorder the parts of each compound, so, the final debt remains the same.

The debt-sequences underlying executions serve to check two kinds of constraints: (i) the various parts of actions must occur in order, and (ii) the dispatch -times of the various parts of an action must be at fixed offsets from one another. Reordering leaves dispatch-times alone, so (ii) is irrelevant.

So only (i) is an obstacle. There are two directions to take care of. If we were to swap the order in which two parts of the same action are dispatched, then immediately the underlying debt-sequence fails to be defined due to violating (i), and will remain so until said swap is undone. Conversely, if we never swap the order in which parts occur, then (i) remains satisfied. As (i) and (ii) are all and only the constraints that need be met: the underlying debt-sequence remains defined iff the relative order of the parts of actions is maintained. So the qualification added in ($\prec_{\text{deorder}}$), *i.e.*, above and beyond the definition of ($\prec_{\text{mutex}}$), takes care of the only relevant difference with Theorem 3.14 (regarding its dependency on the notion of execution). That is, we redefine **deordering** so as to preserve ($\prec_{\text{deorder}}$) rather than the mutex-order; that doing so has no bearing on the proof of Theorem 3.14 is simply by regarding all parts of an action as pairwise mutex with regard to what may be swapped.

So finish by the proof of Theorem 3.14. □

Discussion. Depending on ones taste for formality, the above perhaps counts as a proof. In any case it should be clear that the theorem is true. Which, so far as the thesis goes, is enough. For that matter, even truth/proof of this specific notion of equivalence is relatively unimportant. (However, 'every' remaining formal argument appeals to this specific theorem, so in a formal sense it is crucial to understand and believe.) All that we truly desire are equivalence/dominance reductions powerful enough to notice that a given probem is failing to exercise the facilities of Interleaved Temporal Planning. In other words, we just want a notion powerful enough to reasonably ground whether a problem *really* belongs to just Conservative Temporal Planning (or anything more general).

Then, getting ahead of ourselves for a bit, consider the following. What if we could always reorder (preserving completeness and optimality of course) to sequences that immediately carry out every part of an action? Well, then it would lose 'nothing' to abstract back to the perspective that actions lack parts. Such would be a *win*: getting away with abstraction for free is always a win. Then to understand the significance of Theorem 3.21: We say that a problem (causally/nontrivially) *requires concurrency* when such abstraction is non-free.

Returning to the present purpose: so we come to formalizing <u>reduction</u> to finite state transition systems.

3.3.3 Formal Reduction of Interleaved Temporal Planning to Transition Systems

Simply taking a state transition system view upon Interleaved Temporal Planning is entirely straightforward. What is tricky, due to the infinity of time, is to keep the result finite. Particularly values of time make reducing to a state space, even for expanded notions of state, awkward at best.

Organization. First we present the naïve reduction, which is unacceptable by virtue of being infinite. Second we present a reasonable take on a brute-force reduction, by exploiting the fact that time is effectively discrete. Third and fourth we examine more deeply two particular technical obstacles overcome by the latter reduction.

The Naïve Reduction. As far as clarifying semantics is concerned it may be useful to consider the straight-up reduction to state transition systems. However, the reduction has a number of difficulties, most importantly, each neighborhood of a vertex is infinite. For reference, the naïve reduction $M = (V, E, \Sigma, R, \ell, s_0, T)$ is:

$$
\begin{aligned}
V &= \mathit{Balances}, & E &= \left\{\left((s, v, d), (S'_a(s), V'_{a,t}(v), D'_{a,t}(d)), (a, t)\right) \,\middle|\, (a, t) \in \Sigma\right\}, \\
\Sigma &= \mathit{Primitives} \times \mathbb{Q}, & R &= \{(b, b', \cdot) \mapsto (b, b')\}, \quad \ell = \{(\cdot, \cdot, x) \mapsto x\}, \\
s_0 &= \mathit{Initial}, & T &= \{b \mid \mathit{Goal}(b)\}.
\end{aligned}
$$

This is far from acceptable for our purposes, as we aim to compare planning languages to one another by way of reduction to a common setting. Comparing the finite to the infinite is apples to oranges.

To ensure comparability we take the perspective of reducing *via* sequential planning.[14] That imposes a number of syntactic and semantic constraints; two are especially relevant. The significant obstacle is that infinities, *i.e.*, time, cannot be represented (*i.e.*, solving the reduction must fit in PSPACE). The other (minor) obstacle concerns the relationship between compounds and primitives.

Theorem 3.22. *There exists a (brute-force) reduction to State Transition Systems of Interleaved Temporal Planning Problems* via *Sequential Planning. More specifically the mapped-to graph is at worst exponentially large with polynomially large neighborhoods. For notation, let:*

- $\mathcal{P} = (\mathit{FluentDefs}, \mathit{ActionDefs}, \mathit{Initial}, \mathit{Goal})$ *denote the problem,*
- $\mu = \gcd\{\mathit{dur}_a \mid a \in \mathit{Primitives}\}$ *denote its natural unit of time,*
- $\mathit{Times} := [0, 2^m - 1]$ *denote all of the relevant temporal coefficients of* μ*,*
- σ^{-1} *denote the decoding of words into plans, and*
- M *denote its* ***unit-time reduced state transition system*** *as follows.*

Loosely, define:

$$M := (V, E, \Sigma, R, \ell, s_0, T),$$

$$V := \mathit{States} \times \mathit{Vaults} \times \mathit{Debts} \times \mathit{Times},$$

$$E := \left\{ \begin{array}{ll} \Big((s,v,d,k), (S'_a(s), V'_{a,t}(v), D'_{a,t}(d), 0), a\Big) & \Big|\; a \in \mathit{Primitives} \text{ and } t = k\mu; \\ \Big((s,v,d,k), (s,v,d,2k), \qquad\qquad\qquad a\Big) & \Big|\; a = (\text{push-0 k}); \\ \Big((s,v,d,k), (s,v,d,2k+1), \qquad\qquad a\Big) & \Big|\; a = (\text{push-1 k}) \end{array} \right\},$$

[14] We omit real explanation for how the brute-force reduction comes through Sequential Planning. The reader could invert Theorem 3.1 in application to Theorem 3.22 to obtain the representation.

$$\Sigma := \textit{Primitives} \cup \{\text{(push-0 k)}, \text{(push-1 k)}\},$$

$$R := \{(b, b', \cdot) \mapsto (b, b')\}, \qquad \ell := \{(\cdot, \cdot, a) \mapsto a\},$$

$$s_0 := \left(\textit{State}_{\textit{Initial}},\ \textit{Vault}_{\textit{Initial}},\ \textit{Debt}_{\textit{Initial}},\ 0\right), \quad T := \{(s, v, d, k) \mid \textit{Goal}((s, v, d))\};$$

$$\sigma^{-1} := \left\{ k_1, \ldots, k_{m'}, a \mapsto (a, k\mu) \;\middle|\; k = \sum_{i \in [m']} 2^{m'-i}(k_i = \text{(push-1 k)}) \right\}.$$

Then, the language encodes all slackless[15] *solutions, and encodes only solutions:*

$$\text{Slackless}(\text{Solutions}(\mathcal{P})) \subseteq \sigma^{-1}(L(M)) \subseteq \text{Solutions}(\mathcal{P}). \tag{3.34}$$

Proof Sketch. To be more formal we should write along the lines of $V'_{a,k\mu}(v\mu)/\mu$ to denote multiplying in and dividing out by the unit-time. Likewise we should write something like $V = \textit{States} \times (\textit{Vaults}/\mu) \times (\textit{Debts}/\mu) \times \textit{Times}$ to denote that the number of vertices is bounded by using at most the coefficients of μ to encode values of time inside the vaults and debts. In short the result amounts chiefly to applying the observation that time is effectively discrete, Corollary 3.23.

The additional fine point, also discussed below, is to encode (in big-endian binary) the selection of dispatch-times using the virtual actions/transitions "(push-0 k)" and "(push-1 k)". This serves to structure the exponentially many edges leaving a vertex of the naïve reduction into poly-many choices (the bits of the dispatch -times) over poly-many options (number of ground actions plus two). □

The following usefully elaborates, but we do not in fact formally complete the exercise of verifying the theorem.

[15] For reference, roughly $\text{Slackless}(X) := \left\{(a, t)_{[n]} \in X \;\middle|\; (a, t')_{[n]} \in X \Rightarrow (t')_{[n]} \geq (t)_{[n]}\right\}$.

3.3.3.1 Time is Discrete

Time values occur in both the representation of situations and plans, *i.e.*, as the acquisition-times and release-times of locks, and as the dispatch-times of plan steps. For encoding values of time we need to somehow bound the number of relevant values to at most exponentially many; then one boolean fluent per bit will encode time using polynomially many fluents. That alone is not enough to address dispatch -times; an exponentially large neighborhood (*i.e.*, branching factor) at a vertex is little better than an infinite neighborhood. We cannot actually get rid of any of those choices—a perfect reduction does need to preserve all plans, and each such choice of a dispatch-time is legitimate. We can, though, *hide* the exponential by restructuring the space.

Bounding to Exponentially Many Values of Time. While Theorem 3.18 (slackless schedules dominate) is not nearly so strong as Theorem 3.4 (left-shifted schedules dominate), it is still strong enough to support a mildly useful corollary:

Corollary 3.23 (of Theorem 3.18)**.** *Taking the greatest common denominator of all effect durations as the unit of time is completeness-preserving.*

Proof. Let μ denote the greatest common denominator of all effect durations.[16] In a slackless effect-schedule, by the proof of Theorem 3.18, every effect of every action starts, and finishes, at a time that is a sum of edge weights of the corresponding Simple Temporal Network. These are all either 0, an effect duration, or the negation of an effect duration. Sums of such are, then, multiples of μ. Hence restricting to multiples of μ as start-times retains all slackless schedules. So the corollary follows by the statement of Theorem 3.18. □

In other words, the effective domain of time is Natural numbers. Then it remains only to impose an upper bound. It is possible to prove the existence of a bound [175]. Also, a mere 64 bits counts out a half millenia in nanoseconds; for that matter, a century in seconds fits comfortably in 32 bits. Automated planning is in zero danger of supporting such scale any time soon. Anyways essentially any implementation will just use a hardware type for bounded arithmetic; 'only' a SCHEME programmer might accidentally default to unbounded arithmetic.

Then let us take it as true, for whichever reason seems best (proof or fact), that certainly polynomially (if not simply constantly) many bits suffice. Note that, in more general settings, specifically when durations of primitives are variable, it is not so simple to derive a maximum necessary precision [25]. It is then likewise not so simple to derive a realistically bounded encoding of relevant values of time. That here the semantics remain simple enough to readily do so is useful/convenient (and of course quite deliberate).

Encoding Numeric Fluents and Arithmetic. We assume that planners can directly represent numeric fluents, bounded to exponentially many possible values, and perform basic arithmetic operations upon them. In practice one extends the code to literally do so [12, 37, 103, 110, 115, 137, 154, 169, 174].

Formally the assumption is justified by polynomially-space-bounded Turing-completeness [24]. Technically this means one can make classical planners simulate every feasible computation, indeed, a good bit more than that. However, in

[16] As elsewhere, for the sake of argument assume the initial situation and goal are temporally uniform; in practice, use compilation tricks to ensure that quantities such as the effective unit of time are defined correctly. In this particular context, by Theorem 3.18, it is unnecessary to address deadlines; deadlines can be rounded down to the nearest multiple of the g.c.d. of every other duration. For more general forms of temporal logic it is necessary to take into account the goal expression.

practice, (a) it is extremely awkward to implement algorithms in a planning language, and (b) planners are anyways highly ineffective virtual machines, in the sense that moreover achieving effective performance typically requires deep knowledge of the internals of the planner to be applied. (So, usually, it is simplest and best to just modify the code directly.) Still it is worth keeping in mind that, with deep knowledge and great care, it is sometimes possible to take the PSPACE-completeness result at face value [11, 74, 75, 164]. Less atypically the utility in this sort of formal exercise lies in determining why the planner would fail, using such insight to guide direct extension of its implementation.

For an example of taking PSPACE-completeness literally, note that even STRIPS -style planning formally permits such operations as multiplication by 2: see Figure 3.2 for an ADL description. (Recall that conditional effects may be compiled down to STRIPS in a polynomial fashion [158], indeed, much as we are about to do for dispatch-times.) Of course, multiplication by 2 is one of the easy cases. Also one can implement non-linear operations, see Figure 3.3. Naturally, simulating always hardware-available operations should be taken strictly figuratively.

Reducing Numeric Parameters to Numeric Fluents. To address numeric parameters (the dispatch-times), encode them as numeric fluents under complete control of the planner. For concreteness let us say we encode the planner's ability to pick a dispatch-time as follows; many other schemes are possible. Let (push-1 f) and (push-0 f) be couplets of book-keeping primitives (per f) for manipulating the encoding of some numeric parameter as a numeric fluent f. Respectively these set the value of f by the effects: $f := 2f + 1$ and $f := 2f$. More formally, set $\mathit{eff}_{(\text{push-1 } f)} := \left\{\left\{f \mapsto x^{\in[0,2^m-1]}\right\} \mapsto \left\{f \mapsto (2x+1)^{\in[0,2^m-1]}\right\}\right\}$ for some large enough

```
;;;; How NOT to make classical planners do arithmetic.
;;;; Problem file insertions.
(:init (next b0 b1) (next b1 b2) ... (next b30 b31)
 ...)
;;;; Domain file insertions.
;; uint32: unsigned 32-bit integer; bit: one of the 32 bits.
(:types uint32 bit ...)
(:constants b0 b1 b2 b3 b4 ... b31 - bit ...)
;; value: true->1, false->0.
(:predicates (value ?f - uint32 ?i - bit) ...)
;; Double the value of f, encoded as an array of bits:
(:action push-0 :parameters (?f - uint32)
 :precondition (and (not (value ?f b31))) ; forbid overflow
 :effect (and (forall (?i ?j - bit) (when (next ?i ?j)
  (and ; so j=i+1
    ;; conceptually, for each i, set f[i+1] = f[i]:
    (when (value ?f ?i)       (value ?f ?j))
    (when (not (value ?f ?i)) (not (value ?f ?j))))))
  (not (value ?f b0)))) ; set f[0]=0
```

Figure 3.2: How to implement a left-shift by 1 bit in the ADL-fragment of PDDL.

bound 2^m; likewise define pushing 0 into the least significant bit. Then for example the sequence (push-1 k), (push-0 k), and (push-0 k) leaves the (book-keeping) fluent named “k” with the value 4.

Let “k” be special in that it stands for dispatch times specifically; have every compiled representation of a real effect reset “k” (set all of its bits to 0). Then the compiled plans are a rather faithful transliteration of effect-schedules, *i.e.*, with dispatch-times coded in big-endian. That has a certain appeal in theoretical terms, for example, it does not increase the worst-case number of possibilities.

It is also interesting to consider dedicating separate virtual actions to each bit (*i.e.*, “(set-bit-1 f i)” and “(set-bit-0 f i)”), as that would behave better under subsequent relaxation of the problem. A downside is that there would be many ways of naming the same dispatch time. A further possibility is to write out a number

```
(:init (larger b31 b30) (larger b31 b29) ... (larger b1 b0)
   ...)
(:predicates (larger ?j ?i - bit) ...)

(:action max :parameters (?a ?b ?c - uint32)
 :precondition (and )
 :effect (and
  ;; if a is larger than b then set c to a:
  (when (exists (?i - bit) (and
     (value ?a ?i) (not (value ?b ?i))
     (forall (?j - bit) (implies (larger ?j ?i) (and
      (implies (value ?a ?j) (value ?b ?j))
      (implies (value ?b ?j) (value ?a ?j)))))))
   (forall (?i - bit) (and
    (when (value ?a ?i)       (value ?c ?i))
    (when (not (value ?a ?i)) (not (value ?c ?i))))))

  ;; if b is larger than a then set c to b:
  (when (exists (?i - bit) (and
     (not (value ?a ?i)) (value ?b ?i)
     (forall (?j - bit) (implies (larger ?j ?i) (and
      (implies (value ?a ?j) (value ?b ?j))
      (implies (value ?b ?j) (value ?a ?j)))))))
   (forall (?i - bit) (and
    (when (value ?b ?i)       (value ?c ?i))
    (when (not (value ?b ?i)) (not (value ?c ?i))))))

  ;; if a and b are equal then set c to b:
  (when (forall (?j - bit) (and
     (implies (value ?a ?j) (value ?b ?j))
     (implies (value ?b ?j) (value ?a ?j))))
   (forall (?i - bit) (and
    (when (value ?b ?i)       (value ?c ?i))
    (when (not (value ?b ?i)) (not (value ?c ?i))))))))
```

Figure 3.3: How to implement $c := \max(a, b)$, see Figure 3.2.

of operators for adding by 2^0, 2^1, 2^2, and so forth [175]. That is likely a superior option in a language allowing one access to addition as a primitive operation. For one thing, the heuristics will likely have been designed with that operation in mind [37, 115].

All three are extremely similar both in what formal constraint is satisfied and how so: they all ensure that plan length increases only polynomially, likewise for branching factor, in order to represent selection of dispatch-times. None do so in a particularly clever manner. That is much of the challenge of developing an effective temporal planner, relative to classical planning: cleverly structure the choice of dispatch-times. So it is somewhat interesting to approach the issue from the reduction angle as seriously as Figures 3.2 and 3.3 begin to. However, it is more or less strictly superior to approach the technical issue directly. At least, all state-of-the-art strategies for choosing numeric parameters in general and values of time in particular seem to hail from a direct perspective [6, 93, 137].

Long story short: Time is, effectively, finite. So the greatest obstacle to reduction to transition systems via classical planning is eliminated. There is one other matter that is significant if generalizing the treatment here.

3.3.3.2 PDDL*-style Decomposition Is Simple*

It is easy to imagine generalizing to compund actions that decompose into other compounds [187], and very quickly it becomes ‘impossible’ (but see [74, 75]) to reduce to classical planning [63]. The highly restricted case of decomposition here though is easily reduced. The limitation that must be worked around is that creation and destruction of fluents is not permitted: variable-size structures cannot be reduced entirely literally.

The only portions of a situation that are effected by the limitation are obligations. These record the promised start-times of parts not yet dispatched, so, vary in size from empty up to the number of proper parts.

In general one would need to take the varying size seriously. Meaning we would be discussing the reuse of storage for differing purposes at differing times. However, for our purposes, implementing full-blown memory management within the model would be overkill. Doing so, incidentally, is how one shows that even complex forms of HTN-planning reduce to classical planning under the relatively harmless assumption of a bound on maximum stack depth [74, 75].

Here we can just preallocate storage for the largest case, using extra bits to encode what portions are presently in use. We make one very small optimization. As the parts of a compound are totally-ordered, it suffices to count how many remain in order to know which ones remain. In the partially-ordered case, one would associate a dedicated flag to each part instead.

Proposition 3.24 (Encoding Obligations)**.** *Map an obligation O into the signature $\hat{O} = \left(k^{\in[0,2]}, \mathtt{bgn}^{\in\mathbb{Q}}, \mathtt{fin}^{\in\mathbb{Q}}\right)$ by the rule:*

$$\hat{O} := (|\mathrm{Dom}(O)|,\ O_{\mathtt{bgn}} \text{ if defined else } \textit{garbage},\ O_{\mathtt{fin}} \text{ if defined else } \textit{garbage}).$$

Reachable obligations may be recovered by the inverse rule:

$$O = \left\{ \begin{aligned} &\mathtt{bgn} \mapsto \hat{O}_{\mathtt{bgn}} \mid \hat{O}_k = 2; \\ &\mathtt{fin} \mapsto \hat{O}_{\mathtt{fin}} \mid \hat{O}_k \geq 1 \end{aligned} \right\}.$$

Proof. The assertion holds by induction on the definition of debt transition functions. That is, the form $O = \{\mathtt{bgn} \mapsto \cdot\}$ is never reachable, and hence the case $\hat{O}_k = 1$ is not, in fact, ambiguous. $\square$

Significance of Fixed Size Representation to Temporal Planners. Note the direct relevence of the proposition to differing ways of implementing representation of temporal situations within temporal planners proper. That is, some literal reading of the definitions might make the 'mistake' of using complex memory management just to track promised start-times. For more general forms of temporal planning one does need such mechanisms. It is, for example, not so simple to usefully bound the maximum size of an event-queue [54], not even theoretically speaking [175]. Here we have stipulated that actions may not execute concurrently with themselves—precisely so that full-blown event-queues become unnecessary. This restriction upon semantics is virtually 'free', in practice. There is *one*, mild, downside: see [35]. Albeit even for such domains it is possible to argue that forcing domain modelers to employ the workaround is actually *better* for everyone involved.

3.4 DISCUSSION OF NOVELTY AND SIGNIFICANCE

So we have reinvented several wheels: Left-Shifted, Deordered, Slackless, STNs, Unit-Time, and several reductions to State Transition Systems [8, 47, 188]. The first two results are so basic we consider them to be litmus tests of the correctness of definitions purporting to capture the spirit of Conservative Temporal Planning. Likewise, we take the sufficiency of Simple Temporal Networks for capturing the induced scheduling problems of Interleaved Temporal Planning as, more or less, a defining characteristic. There is no dearth of formal treatments and published proofs that we could have—nominally—simply reused.

- For example, we have claimed that the Completeness Theorem for Totally Unambiguous Partially Ordered Plans is equivalent enough to Backström's Deordering Theorem [8, 153].

- In a sense, even merely the notion that POCL-planning is at all sound is equivalent enough to the reordering insight [28, 149].

- As far as the use of First-Fit in temporal planning goes, we could just cite the greedy post-processing of SAPA [53].

- Indeed, it is surely easy to find a great many citations for the use of First-Fit in generalizing to temporal planning. It *is* the obvious first step to take.

- . . . and so forth and so on [2, 4, 25, 30, 70, 71, 109, 112, 151].[17]

That others reinvent is a poor excuse. The question is begged:

Why reinvent?

[17]Dubiously, we could even take code itself to be 'theory'.

Answering the question is interesting, but hardly crucial. For example, lack of novelty is anyways at most a weak criticism: independent verification is the hallmark of good science. In fact—though we shall offer arguably little evidence—the work through Chapter 3 does possess a certain degree of technical novelty (and significance). The short of the matter is that we differ technically from the closely related work in (at least) two core issues:

- Regarding chiefly Conservative Temporal Planning: our definition of *mutual exclusion* differs subtly, in useful fashion, from that of TGP [188]. For reference, we assert that effects are mutex when one writes to a dependency of the other; TGP 'declares' instead that effects are mutex when one contradicts either a precondition or a postcondition of the other. The difference is that TGP assumes that whenever the value of a fluent before and after an action executes are surely the same, then the action only read-locked the fluent. We however distinguish between only reading from a fluent and subsequently writing back the value it already has.

- Regarding chiefly Interleaved Temporal Planning: our treatment of *change* more accurately reflects the practice of so-called PDDL-planning [71]. For reference, we permit only durative effects and durative conditions; PDDL nominally permits only instantaneous effects, instantaneous conditions, and durative conditions. So for example, faithful syntax for ITP problems permits expressions such as "(over [start,start+3) ⟨foo⟩)", but never expressions like "(at start ⟨foo⟩)".

The following lengthy discussion attempts to 'prove' that these differences are, as claimed, both (i) meaningful, and (ii) for the better. That is, to appreciate the fol-

lowing, we must take it as rebuttal to an undeserved criticism: one could hypothetically take the work here to lack novelty and/or significance. Lacking such doubt, we should instead just continue along the main line, to Chapter 4, Page 227.

Organization. We discuss in depth two particular issues distinguishing our treatment here, respectively concerning novelty of CTP and ITP. To be (arguably) quite unfair to the related work, we paraphrase as:

- (Section 3.4.1) *Parallelism* and *Concurrency* are distinct, hence:
  - TGP is suboptimal, incomplete, unsound, and so forth for CTP [188].
  - Partial-Order Planners typically just improve runtime in SP [8, 28, 149].
  - Petri-Nets likewise relate better to Sequential Planning [112, 113].
  - Even HTN-planners presumably fail to make the distinction [63, 155, 156].
- (Section 3.4.2) *Discrete* and *Discontinuous* are distinct, hence:
  - VAL corroborates, and so exacerbates, the difficulties of PDDL [71, 119].
  - Hybrid Automata only *weakly* relate to discrete temporal planning [111].
  - ZENO and friends fail to subsume ITP [12, 70, 137, 151, 154, 168].

We could, but will not, also examine several easily mustered generic excuses:

- Duplication of effort is at most a small crime. Mitigating which, we have sequestered all that could even be acccused of a lack of novelty.
- Anyways, to non-experts, a self-contained reference is a contribution.

- Albeit the treatment here arguably misses that ineffable sweet spot of both simplicity and generality. Still it seems reasonable to claim progress towards a textbook-level understanding of the issues.

- Coming up with proofs from several angles is important. In that respect, the strong bias here from forward-chaining is a novel enough take on the insights of Partial-Order Planning. *E.g.*, there are 'but two' references to give for tricking forward-chaining into performing partial-order planning [30, 36].[18]

  - The connection of the former reference to temporal planning (and perhaps even to partial-order planning) is surely unclear.

  - The latter directly concerns temporal planning (and to a lesser extent partial-order planning). We can surely claim that some readers find that treatment notably denser than ours (equally surely the converse holds).

---

[18]Regarding connections between differing search paradigms we should also consider at least Kambhampati's theory of Refinement Planning [124], and McAllester and Rosenblitt's planner: SNLP [149]. The *adjacency* constraint, $a * b$, from Kambhampati is an interesting way to fold both forward-chaining and backward-chaining perspectives together. It is far more constraining than it need be though (it was not designed for computational promise, mind): the deorder relation, $\prec_{\text{deorder}}$, we build here is computationally far superior.

The insight of SNLP was that partial-order planning is 'really' about effecting an equivalence reduction on totally-ordered plans. (For a contrary, we feel mistaken, view upon partial-order planning we refer to Chapman's TWEAK [28], which takes optimizing partially-ordered plans as an end unto itself; specifically Backström's distinction between parallelism $\neg(a \prec b \vee b \prec a)$ and concurrency $\neg(a \ \# \ b)$, which we also discuss, is a strong rebuttal [8].) We work from the converse direction: <u>deordering</u> totally-ordered plans goes towards a different implementation of the same notion. Specifically we work from the *canonical representative* side of the coin; SNLP focuses on direct representation of the *equivalence classes*.

The converse direction has an interesting advantage. In particular we may achieve the same gross benefit with respect to reducing the size of the search space, at the same time, we may also easily connect up the classes/representatives *via* forward-chaining. Forward-chaining is nice, because it lends itself quite well to techniques such as <u>state-based reachability heuristics</u>. So we take our work here as akin to, if not an improvement upon, the same-in-spirit achievement of Nguyen and Kambhampati's RePOP [160], which improves upon SNLP precisely by leveraging such heuristics.

To be accurate the SNLP constraint ($a \xrightarrow{p} b \wedge a \xrightarrow{\neg p} b$) defines a different reduction than our constraint (*i.e.*, $\prec_{\text{deorder}}$). Its notion is 'better' in that it is coarser/weaker, *i.e.*, leads to fewer equivalence

classes; the relationship is the same as that between the SNLP-space and the space of Totally Unambiguous Plans [153]. Of course the SNLP view upon equivalence is not in fact better than ours: specifically it is too coarse for Temporal Planning. That fact may be had by analogy with duplicate state elimination, or closer to home, TGP's too weak notion of mutex.

We mention all this work in classical planning only because many of the notions are tightly linked at the technical level. It is worth keeping in mind though that the higher level contexts differ, as do the low-level details. Then for as much as we may point to similarities, the work here cannot, for example, be held to lack novelty due to its relationship with SNLP.

3.4.1 Parallelism versus Concurrency, Or: TGP is Suboptimal

We differ in our approach, from Smith and Weld in their development of Temporal GraphPlan (TGP) [188], to defining the meaning of *mutual exclusion*. Note that the precise formal meaning of the Left-Shifting Theorem hinges upon that definition (as does everything else). So while we attribute their work, our theories certainly differ meaningfully (at a sufficiently technical level): their proof does not, formally, establish our theorem. The philosophical distinction to be made is between *commutativity/parallelism* and *nonmutex/concurrency* [8].

We begin with a high level demonstration of our claimed superiority. Next we recap the two formal definitions of mutex for the special case of STRIPS-style. Then we demonstrate how the wrong definition leads TGP to suboptimality in an entirely fleshed out Blocksworld (counter-)example. Finally we touch upon a great many technical and philosophical points and counterpoints.

3.4.1.1 Intuition: Lecture Scheduling

To partially demonstrate our claim, at a high level: Consider scheduling classroom usage. Surely it matters relatively little whether Physics is taught from 9:00am–9:45am, with Biology following at 9:55am, or *vice versa*. That is, presumably, qualitatively speaking, both orders are feasible and leave us in the same final situation. Even clearer: scheduling them both for the same time and place will fail.

Now, given numerous constraints on who teaches what, who (wishes to) attend what, availability of rooms, cost of electricity, *etc.*, we might very well find it interesting to automate such scheduling of classroom usage. For our purpose, the key point is that, while leaving the ordering of lectures in each room up to the scheduler, we specifically wish to explicitly rule out scheduling two classes for the same time

and place. Such is half of the distinction between *parallel* and *concurrent*: two lectures for the same room are *parallelizable*—either order has identical result—but not *concurrently executable*.[19] So our question is: How temporally expressive must our planners be in order to allow us to model the distinction?

Given the theory developed, it would *seem* that Conservative Temporal Planners are inadequate: these more or less lack the power to distinguish between more than every sequence of actions (Theorem 3.4). More abstractly, *temporary effects* are what lie beyond their understanding. So for contrast, note that an interleaved model of classroom usage is to state that *availability* of the classroom is false for the duration of the lecture.

However, this particular kind of temporary effect does lie within the power of CTP. That is because the interaction is strictly 'negative'. Indeed, the ability to model temporary negative interactions is all and only what our definition of mutex is concerned with. Specifically, to model temporary negative interactions, we need to have the concerned activities (1) write-lock some common fluent (*i.e.*, declare exclusive access to a shared resource), but actually (2) leave it alone (*i.e.*, write back the value already there).

Our preferred psuedosyntax resembles "uses r" for some model r of a shared resource (*i.e.*, r = room). Which is largely *because* the syntax dodges the semantic question. For the nuts and bolts: the first runner-up for modeling shared resources resembles "$r := r$", which (a state-dependent assignment) is within the power of ADL. The second runner-up further compiles that down into state-independent effects (so resembles "$r = v, r := v$" for any/each—ideally unique—legal

[19]Neither *parallelism* nor *concurrency* imply each other. *Required concurrency* is where concurrency succeeds despite failure of every sequence. The section discusses the reverse non-implication.

value $v \in \mathit{Values}_r$). In STRIPS-style, with u denoting the proposition "$r = v$", to model the usage one extends the action description as in $(P \cup \{u\}, D, A \cup \{u\})$.

Then let us consider TGP. The technical issue is that TGP regards, we claim erroneously, as 'mutex' *only* those primitives asserting distinct values of some common fluent (either before or after they execute). (In STRIPS-style, TGP says that a and b are mutex iff $\{x, y\} = \{a, b\}$ are such that x deletes either a precondition or add of y.) That *would* make sense if formalizing *parallelism* and/or *commutativity* of primitives. Reflect though upon (two instances of) "$r = v, r := v$", our model for a shared resource (within STRIPS-context, which TGP is limited to). In particular note that all copies are TGP-nonmutex: the TGP notion of mutual exclusion fails to recognize our intent. Meaning, when all is said and done, that TGP can be taken as *unsound*:

TGP *will schedule lectures for the same time and place!*

3.4.1.2 Definitions

Towards a formal demonstration, recall our definition of mutual exclusion:

Definition. Two primitives are **mutex** when either writes to a dependency of the other.

Restricted to STRIPS primitives (P_a, D_a, A_b) and (P_b, D_b, A_b), we say a and b are mutex precisely when either:

$$\begin{aligned} &(D_a \cup A_a) \cap (P_b \cup D_b \cup A_b) \neq \emptyset, \qquad \text{or } (\textit{vice versa}) \\ &(D_b \cup A_b) \cap (P_a \cup D_a \cup A_a) \neq \emptyset. \end{aligned} \tag{3.35}$$

It is not in fact clear what mutex truly means to TGP in anything besides a procedural sense [188].[20] The notion from GraphPlan, though, is entirely clear. So let us pretend that TGP mutex means what it ought to given generalization from GraphPlan.

Definition 3.7. Two primitives are **GraphPlan-mutex** when either (i) one contradicts a precondition of the other, or (ii) one contradicts a postcondition of the other.

For STRIPS primitives (P_a, D_a, A_b) and (P_b, D_b, A_b), GraphPlan asserts that a and b are mutex precisely when either:

$$\begin{aligned} &D_a \cap (P_b \cup A_b) \neq \emptyset, \qquad \text{or (\textit{vice versa})} \\ &D_b \cap (P_a \cup A_a) \neq \emptyset. \end{aligned} \tag{3.36}$$

So in short we are contrasting:

$$\begin{array}{cccc} \Big((D_a \cup A_a) \cap (P_b \cup D_b \cup A_b)\Big) & \cup & \Big((D_b \cup A_b) \cap (P_a \cup D_a \cup A_a)\Big) & \neq \emptyset, \quad \text{versus:} \\ \Big((D_a) \cap (P_b \cup A_b)\Big) & \cup & \Big((D_b) \cap (P_a \cup A_a)\Big) & \neq \emptyset; \end{array}$$

For futher comparison, drawing from SNLP yields approximately:[21]

$$\Big((D_a \cup A_a) \cap (P_b \cup A_b)\Big) \quad \cup \quad \Big((D_b \cup A_b) \cap (P_a \cup A_a)\Big) \quad \neq \emptyset.$$

Which definition is right? Well, that depends entirely on context. For temporal planning, we use mutex to control whether primitives are allowed to be *concurrent*. Due to that the right definition is ours (the most constraining one), which point we

[20] Smith and Weld disagree under detailed interrogation: *ambiguity* is entirely fair.

[21] POCL planners define *threats*, not *mutexes*. Ignoring the distinction is natural, but conflates the interesting and related difference between SNLP-plans and Totally Unambiguous Plans [153].

(**init**	(:= (below a) d)	(:= (below d) table)
	(:= (clear a) `True`)	(:= (clear d) `False`)
	(:= (below c) b)	(:= (below b) table)
	(:= (clear c) `True`)	(:= (clear b) `False`))
(**goal**	(= (below a) b)	(= (below c) d))

Figure 3.4: Swap block a and block c.

are about to elaborate upon. For partial-order planning, the notion is to use mu-tex to control whether primitives are allowed to be *parallel*: the least constraining definition is, there, philosophically correct. (The SNLP definition, the middle road, has interesting computational merits, *i.e.*, towards its namesake: *systematicity*.)

3.4.1.3 A Counterexample in BLOCKSWORLD *to Optimality of* TGP

To drive the point home: Consider the BLOCKSWORLD model on Page 51. Observe that in TGP syntax the model of the hand fits the pattern considered non-mutex by TGP: a precondition upon, and add, of “(empty hand)”. TGP *continues to make the mistake a sequential planner would, despite our attempt to fix the model for the temporal context.* For further detail, consider specifically the problem depicted in Figure 3.4 (and add one block to the domain). The problem is to take the top blocks of two towers and have them switch places, using one hand.

The in-truth duration-optimal solution, also size-optimal, takes 6 time units and 3 movements. Specifically, use the table as temporary storage for one of the two blocks. So one of the two optimal plans is to move:

1. block a from block d to the table,

2. block c from block b to block d, and finally

3. block a from the table to block b.

In contrast, TGP will select a duration-suboptimal solution. Specifically TGP will move both blocks to the table, and then move both blocks to their destinations. Doing so with one hand takes 8 time units and 4 movements. There are six concrete variations, one is to move:

1. block a from block d to the table,
2. block c from block b to the table,
3. block a from the table to block b, and
4. block c from the table to block d.

The reason that TGP will choose a duration sub-optimal option is that, in its mind, those options are faster. Specifically, TGP mistakenly imagines such plans to require 4 time units: 2 parallel steps of duration 2 each. (In GraphPlan parlance, the 4-action plans consist of 2 parallel steps, and in particular are optimal with respect to minimizing parallel length.) Formally:

Proposition 3.25. *Consider Single-Handed* Blocksworld. *Any pair, say* (a, b)*, of movements of distinct blocks to and from distinct places, with "the table" understood as an infinite set of distinct places, are* GraphPlan*-nonmutex. For notation:*

$$\emptyset = \Big((D_a) \cap (P_b \cup A_b)\Big) \cup \Big((D_b) \cap (P_a \cup A_a)\Big). \tag{3.37}$$

Proof. The only fluent common to any movements of disjoint blocks to and from disjoint-or-table places is "(empty hand)": $\{(\text{empty hand})\} = (P_a \cup D_a \cup A_a) \cap (P_b \cup D_b \cup A_b)$ for such (a, b). The fluent is never set to false: $(\text{empty hand}) \notin D_a \cup D_b$. So the actions are GraphPlan-nonmutex: $D_a \cap (P_b \cup A_b) = \emptyset$ and *vice versa*. □

In contrast:

Proposition 3.26. *Every pair, say* (a, b)*, of movements in Single-Handed* BLOCKSWORLD *are mutex, by virtue of writing to "(empty hand)". For notation:*

$$\begin{aligned}\emptyset \neq \quad & \Big((D_a \cup A_a) \cap (P_b \cup D_b \cup A_b)\Big) \\ & \cup \Big((D_b \cup A_b) \cap (P_a \cup D_a \cup A_a)\Big);\end{aligned} \tag{3.38}$$

more specifically,

(empty hand) $\in (Writes_a \cap Depends_b)$, *and symmetrically*

(empty hand) $\in (Writes_b \cap Depends_a)$.

Proof. The fluent "(empty hand)" is common to every pair of movements. Indeed, every movement writes to it: (empty hand) $\in$ *Writes*. Hence all are mutex. □

Then, despite Smith and Weld's duration-optimality theorem for TGP [188]:

Theorem 3.27 (TGP-suboptimality)**.** GRAPHPLAN *is certainly a duration-suboptimal temporal planner; presumably* TGP *is also.*

In general any so-called temporal planner failing to distinguish parallelism from concurrency may be called suboptimal, incomplete, unsound, and so forth.

Corollary 3.28. *Partial-Order Planners 'never' address the unit-duration special-case of anything rightly called "Temporal Planning".*

Proof. See the preceding counterexample for the initial claim. The following discussion should help clarify the sweeping generalizations. □

From suboptimality: hence TGP is incomplete (*i.e.*, if we impose deadlines). Indeed, we *could* even call TGP unsound: its literal output would schedule the first pair of movements concurrently (likewise for the second pair). As such is physically impossible, attempting to execute the schedule must indeed encounter an error eventually.

Presumably, though, the severity of that error would turn out to be mild. For example, the definitions we give for Conservative Temporal Planning themselves fold in an automated validation and rescheduling of any given set of requests. (Because such is easy to do.) Feeding that machinery the broken output of TGP would automatically result in a physically plausible/possible set of actual dispatch-times. Specifically, the machinery would automatically reschedule the 4 movements so as to take place sequentially. So among all the theoretical charges we may at least somewhat reasonably level: we prefer to call TGP suboptimal.

The real issue to remember though is the root cause. Namely, TGP (likely) has the wrong definition of mutex.[22] That is, (definitely) the natural generalization from GraphPlan's notion is wrong: *parallelizable* and *concurrently executable* are, while quite similar, nonetheless meaningfully distinct.

Observe, that, having seen one example, it is easy to construct *many* more. For example, with 2 pipes to a liquid container, one may concurrently fill and empty. With only 1 access pipe, such activities are (perhaps) commutative but nonetheless (certainly) mutually exclusive.

[22] Note that from an erroneous definition we may derive 'endless' theoretical complaints.

3.4.1.4 Discussion: Pros/Cons

The over-arching point is that, in contrast, our formal treatment leads to the right conclusion in 'every' case. That is, our notion for mutual exclusion in temporal planning, which we draw from Backström, Fox, and Long [8, 71], seems best 'in general'. Unfortunately, nothing is ever clear-cut. At a fairly technical level:

– Our notion has the unfortunate consequence that turning on distinct lights must be scheduled sequentially. That is because we forbid concurrently updating some common fluent encoding whether the room is lit. Of course there are unnatural models that bypass the issue.

+ A similar issue, but this time in our favor, concerns concurrent attempts to turn on the *same* light. Perhaps it seems that such ought to succeed in general? In fact the kitchen light of my childhood home is such that (i) concurrency is possible, but (ii) does not have the desired result. Specifically, flipping both switches in an attempt to turn the light on instead results in it toggling on and then off again (very quickly).

  Humans, of course, respond robustly to such failure: only one of the two parties proceeds to remedy the situation. In fact, usually it is understood ahead of time who 'owns' the right/responsibility to manipulate the light. So the failure rarely occurs in the first place. Our whole apparatus of locks, *etc.*, is towards getting automated systems to behave likewise (as is desirable).

– Technically one can exploit a quirk of STRIPS semantics in order to get TGP to do the right thing. We view such as an egregious hack, discussed next. For reference though, we could make every action precondition upon, add, *and* delete availability of the hand. The quirk is that the add effect takes

precedence, which means nothing to STRIPS: omitting the delete means, to STRIPS, the same thing. To GraphPlan, the distinction is that, with the 'shadowed' delete in place, any pair of movements are, as desired, considered mutually exclusive.

+ First of all the domain modeling hack does not address the point that TGP behaves erroneously on an entirely reasonable model.

  To go further, suppose we slightly rework STRIPS semantics as follows (which is quite natural). Let (P, E, F) stand for precondition, effect, and *frame*. The meaning is that the deletes are to be had by a closed 'world' assumption with respect to the frame. That is, say that the deletes are frame minus effects $D = F \setminus E$, and the adds are just the effects $A = E$. (Then state transitions may be written: $P \subseteq S \Rightarrow S' = S \setminus (F \setminus E) \cup E$.) The point is to render contradictory postconditions nonexpressible. In other words we finesse a tricky semantic question: $D \cap A = \emptyset$ is forced.

  Consider that there are at least three reasonable ways to define what happens when we say that a proposition is both deleted and added ($D \cap A \neq \emptyset$). We could have the add take precedence (which abstracts a *use*), or the delete could take precedence (which abstracts its opposite, a *lend*), or the entire operation might be regarded as self-contradictory (so, never executable). (Alternatively we might regard the result as nondeterministic rather than outright self-contradictory, which difference is, here, moot.)

  The last interpretation is sometimes convenient when considering *conditional effects* [139, 166]. Specifically consider an action stating both "when p then set r true" and "when q then set r false". Such simplifies to asserting that

fluent r will be made either/both true/false, in the case that "p and q" holds before. Taking such as 'too ambiguous to live' is natural.

Meaning we have another piece of evidence in support of the view that permitting $D \cap A \neq \emptyset$ so as to permit neatly fixing TGP's behavior in BLOCKSWORLD is a *kludge*.

– A *principled* fix begins by observing a tendency. When TGP's definition of concurrency is erroneous we may often observe the existence of a shareable —but only unit capacity—resource at a more detailed level of understanding of the planning domain. We can, in general, fix such models by breaking the task up into an acquisition-piece and a use+release-piece.

  E.g., in BLOCKSWORLD, the normal 4 action schema encoding generalized to TGP definitions works just fine; then it is possible to delete availability of the hand during the now separately modeled halves of a movement.

+ Note the relationship of the suggested manipulation to Interleaved Temporal Planning. Specifically two points are (somewhat) in our favor: (i) our approach ultimately permits a slightly better version of the manipulation (we can bound how long a shareable resource is exclusively held), and (ii) the principled fix for TGP has significant negative computational consequences (unlike strengthening the notion of mutex).

– Concerning (i), as TGP *is* duration-optimal so long as we respect its semantics: that unabstracting loses the ability to bound the amount of time shared resources are held is interesting, but, relatively unimportant. That is, the constraint that a block cannot be held up indefinitely is less important when the planner is anyways going to minimize such intervals automatically.

– TGP's more permissive notion is of itself something of a computational advantage. (The associated equivalence relation is coarser, *i.e.*, 'better'.) It is interesting to compare with classical planning, where similar computational advantages are easily sought (as the semantic differences are irrelevant) [114, 149, 196, 203].

  Furthermore, with several hands, the weaker interpretation of mutual exclusion lends itself towards modeling all the hands as a simple counter [189]. (Such is a related, but far more significant, computational boon.) In contrast, lacking any notion of a *shareable write-lock*, more or less each hand would need to be modeled as a full fledged entity in its own right (*cf.* [35]). Insofar as such fails to support useful abstraction, then the point is counterpoint to (ii).

+ There is, of course, no impediment to generalizing our treatment to shareable write-locks. Note that starting from entirely exclusive write-locks, the simpler case, is the 'right' way.

  Indeed, in general, the advantages for TGP can instead be sought as explicit generalizations from a simpler workable case. Our treatment is such.

For philosophy instead: *Even, perhaps especially, shades of meaning matter.*

- From a user/domain-modeler perspective, 'errors' are significant regardless of the engineering-difficulty of remedying the code. At a minimum—if an external workaround is possible/reasonable—such errors are "annoying": "annoying" is, for example, commercially significant.

- To be more realistic, which only increases significance, planning technology, particularly domain-independent planning technology, cannot really claim

application status: planners are, at present, only typically useful as *component* technology. So note that differences in formal meaning greatly hamper treating software as components of a larger system. In other words, tools in service of machines must be far more robust than tools in service of humans (because machines are, as of yet, not nearly so intelligent as humans).

- Furthermore, to call this difference in the definition of mutex an "error" goes too far. Both definitions are 'right': for subtly different purpose. So it would be 'wrong' to pretend that the distinction in meaning is just some 'bug' (and hence better left swept under the rug).

- In particular recall (from Brooks [122]) that the nastiest of bugs to find—in a sense the most significant bugs of all—are those arising from perfectly reasonable disagreement across the two sides of an interface. (Generally speaking, two sides of an interface are easily consistent/reasonable in isolation, but inconsistent when brought together.) In other words—a mere corollary of the seminal software engineering theory of Brooks—*shades* of meaning are the leading cause of project failures/delays.

- Then to be specific about our contribution here, by contrast with a 'miscontribution' of Smith and Weld, consider the following. To say that "TGP is duration-optimal" begs would-be users (humans or programmers) to substitute *concurrency* for what in truth is a perfectly functional (generalization to nonuniform durations of GraphPlan's) formalization of *parallelism*. Such is

a significant meta-problem, because the two notions are easily confused, and the subtlety does in fact manifest on real-world[23] problems.

- To boldy state that prior art is "miscontribution" requires elaboration. Particularly it is—extremely—unfair to evaluate algorithms designed for specified formal semantics against differing semantics. To be more specific, first note there are many ways to disambiguate concepts as vague as "parallelism" and "concurrency". Such disambiguation is crucial, because the various interpretations are, naturally, logically contradictory. Then to criticize TGP for implementing some interpretation differing from our own is—in form—an empty criticism: our work (or any work whatsoever) may be, automatically, criticized in symmetric/converse fashion.

  In this particular case the criticism is at least halfway legitimate, and we can prove so. The defense against this form of criticism is to be precise (and internally consistent): merely formalizing semantics goes a very long way towards nullifying philosophical dispute. It is, then, a crucial piece of evidence that the semantics of TGP are ambiguous. That point is especially telling when the question is as foundational as the one asked here, which may be stated as "Are effects mutex with the conditions they establish?". The authors themselves disagreed upon the answer to that question: for at least a span of hours, in person, with full recourse to whiteboards (*etc.*). Indeed, perhaps they still disagree. We could, of course, determine easily enough the behavior of the implementation. However, such point does nothing towards defense, in fact: we can think of no cleaner way to prove that TGP-semantics may be legiti-

[23]BLOCKSWORLD is not a "real-world" application, but it has proven itself to be a good *benchmark*: issues apparent in BLOCKSWORLD are, often enough, real-world issues.

mately criticized as *procedural* (rather than *declarative*). In short there can be no case in support of unambiguity of TGP-semantics.

Is our treatment any better? Naturally, we say yes. To evaluate that: realize the significance of proving the correctness of the naïve reductions to Graph Theory (Theorems 3.1, 3.17, and 3.22). The proofs are trivial—the theorems obvious—and *that* is why we may legitimately criticize TGP by evaluating it against our philosophy/axioms/definitions: its are demonstrably ambiguous, and ours are demonstrably unambiguous (that is, unambiguous relative to sufficient background in Graph Theory).

- We may also, finally and arguably least significantly, point out significant computational implications of the distinction between parallelism and concurrency. Such tend to command, or at least receive, attention; however, in the grand scheme of things, such tendency is slightly unfortunate. (The disproof of direct significance of computational issues is irrefutable: human computation is far and away more expensive than machine computation.) So, with slight hesitation, we remark that *parallelism* versus *concurrency* here correspond closely enough to: (1) *reordering* versus *deordering* of Backström [8], and (2) *Disjunctive Temporal Networks* versus *Simple Temporal Networks* [47]; in particular the computational difference in each case is the same as (3) NP-*complete* versus *polytime-solvable* [125].

  The superiority here is not face-value. We are 'better' here only in the sense of *simpler*, for a reasonably objective definition of such (which is nice).

  Simpler is better, though, only when sufficient. Hence the technical discussion above of *shareable write-locks*. Specifically, for problems featuring con-

current modification of fluents (sometimes called "true concurrency"), our treatment is more or less insufficient. In contrast, TGP can—in extremely limited fashion—model shared write-locks (*i.e.*, TGP permits shared write-locks for the special case when all writers write the same value). Then, for such problems, TGP is better by default: the treatment here is too simplistic to compete.

Can we sum up? Well, perhaps the balance favors us. Still we cannot in good faith claim that our Conservative Temporal Planning is *hands-down* better than TGP -semantics. It is, at least, meaningfully different: better in some ways, worse in others. Which is excuse enough to revisit the basic theory.

Then to conclude discussion of the relationship to TGP: The formalization of "Conservative Temporal Planning" developed here is, to a certain degree, novel in its particular approach to defining the precise relations between interference, mutual exclusion, parallelism, and concurrency. We have opted for clarity/simplicity over flexibility/generality. Such suits our purpose better in several ways: the two greatest advantages follow. Our stance is better suited to a domain-independent perspective (*i.e.*, it 'does the right thing' without domain-modeling 'hacks'). Furthermore it better serves subsequent generalization (*i.e.*, to the interleaved interpretation, which shoulders responsibility for taking a less simplistic view upon action interference).

TGP aside, *i.e.*, in general: It is worth our while to appreciate the distinction between *parallelism* and *concurrency*.

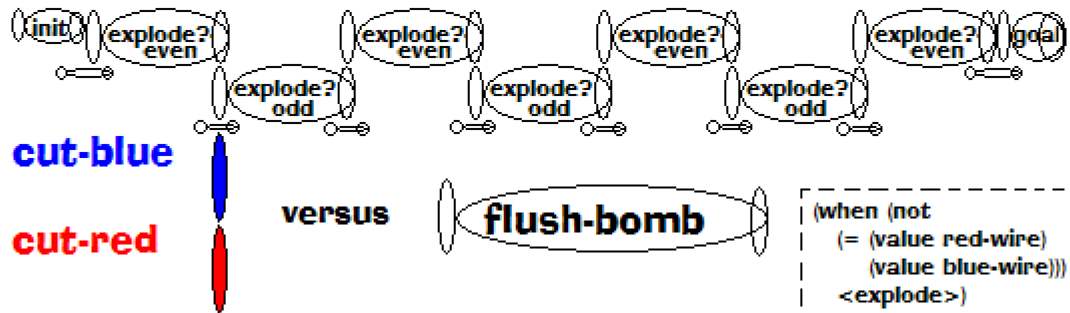

Figure 3.5: The bomb explodes whenever the two wires differ.
Enforcing that is awkward but possible in PDDL. In PDDL *theory*, simultaneously cutting works. In practice, planners agree that some other method must be sought.

3.4.2 Discrete versus Discontinuous, Or: PDDL is Unsound

To set the stage, consider the numerous puzzles in selecting temporal semantics.

Time Type Is time discrete or continuous?

Real-time Guarantees How is time measured? Are delays bounded [109, 154]?

Instantaneous Change Does reality obey a speed limit?

Divided Moment What happens while a fluent changes [2, 4, 71]?

Achilles and the Tortoise How do we prevent infinite action in finite time [145]?

Calculus *Must* we really insist upon sparseness of change?

Buridan's Ass How do we prevent infinitely precise action? [136]?

Soup Bowl *Must* we really insist that agents have bounded precision [182]?

There tend to be quite a few reasonable answers to each, all worth investigating. A tricky thing is to develop reasonable joint anwers (because the questions are hardly logically independent of one another). It can also be tricky just to reasonably

formalize what appears straightforward at the level of philosophy. At least, we claim that the temporal semantics of PDDL possess such flaws (which our formal treatment remedies).

(Jumping ahead:) Somewhat fairer is to claim only that most so-called implementations of temporal PDDL are flawed. Accordingly, temporal PDDL itself is only meta-flawed: meaning we improve by picking easier to implement semantics.

Given the following level of hair-splitting, there is a fine point about our own theory we should clarify up front. First we shall discuss the basic philosophy underlying the design of our formal languages. Then we shall look deeper at the only strikingly dubious aspect of the formal treatment not elsewhere discussed. Namely, the definition of corresponding behaviors is easily taken as flawed with respect to the machinery of vault transition functions.

Following that we discuss temporal PDDL from top to bottom (in relation to our ITP). First we review the key philosophical points underlying temporal PDDL specifically. Secondly we discuss the yet more abstract purpose—mixed discrete-continuous planning—that temporal PDDL truly aims for. That sets the right context for, third, poking a few holes in the specifics of the philosophy as applied to the special case of discrete temporal planning. Fourth we ground some of those criticisms into a specific counterexample (having to do with nuetralization of bombs) of soundness of VAL. Later work, by that group and others, resolves (procedurally) the issue, namely: "Bounded Precision". We have built into our Interleaved Temporal Planning formalism what we take to be the equivalent declarative solution: fifth, we elaborate upon the nature of this resolution both abstractly and with respect to the example problem of nuetralizing bombs.

Finally, when all is said and done, the lesson (or at least our claim) is just that our ITP is a better way of naming, and reasoning about, the class of discrete temporal planners that, for lack of better label, are termed temporal PDDL-planners. Our specific recommendation is that official formal syntax be altered to better reflect the implemented semantics.

1. Permit the use of "(over all . . .)" within ":effect", *i.e.*, permit *durative discrete change*.

2. Forbid the use of "(at start . . .)", "(at end . . .)", and in general everything purporting to be instantaneous.

The length of exposition far exceeds apparent significance. Indeed, the point we are making here *is* relatively insignificant. We must desire to carry out truly formal proof, equivalently to develop implementation, for the details being discussed to matter. If otherwise, then continue to Page 227.

If so: for formal intents, the issues being discussed are rather non-obvious, and thus, at that point, quite important. That is, it is easy to waste a great deal of effort in general working with internally inconsistent definitions, and specifically the nature of time and change are easily formalized in inconsistent fashion. Particularly for an upper bound on temporal expressiveness equal to the intuition "discrete temporal planning": the points we are making here are quite worthwhile. Albeit, our informal discussion in support is perhaps no better than simply baldly claiming authority; for greater formality, the reader might prefer to investigate the stage initially set.

3.4.2.1 *Our Philosophy*

We take reality to be *Fundamentally Continuous*. However, we take formal models thereof to be *Strictly Discrete*. This is not a contradiction.

Let Z denote the (sole) <u>behavior</u> of a plan in the mind of a (deterministic) discrete temporal planner, roughly: $Z^{\in \mathit{Fluents} \overset{\text{total}}{\longrightarrow} \mathbb{Q} \to \mathit{Values}.}$ with each timeline $Z_f{}^{\in \mathbb{Q} \to \mathit{Values}_f}$ encodeable finitely (*e.g.*, say each fluent changes finitely many times with respect to the size of the plan) for each known fluent $f \in \mathit{Fluents}$ (so for one thing $\left|\mathit{Values}_f\right| \in \mathbb{N}$). Let Z^* denote one possible 'ground truth' of behavior, *i.e.*, as we imagine it: $Z^{\in \mathit{Fluents}^* \overset{\text{total}}{\longrightarrow} \mathbb{R} \overset{\text{total}}{\longrightarrow} \mathbb{R}^*}$ with each real timeline $Z_f^*{}^{\in \mathbb{R} \overset{\text{total}}{\longrightarrow} \mathbb{R}^{k_f}}$ being continuous, $Z_f^*(t) = \lim_{x \to t} Z_f^*(x)$, for each real fluent $f \in \mathit{Fluents}^* \supset \mathit{Fluents}$ (a superset of those modeled), each taking value in some k_f-dimensional Real vector space.

Let $\sigma_f \in \mathbb{R}^{k_f} \to \mathit{Values}_f$ denote the partial mapping of the true values of each fluent into modeled values. For apparently 'truly' categorical fluents such as "the color of Bob's shirt", the true value can be taken as the whole concerned distribution of electromagnetic wavelength, or if making the distinction, then instead the true value can be taken as the entire electrochemical state of the observer's brain. Clearly we cannot expect to obtain even remotely perfect computational access to such notions. Assume our imagination is at least halfway realistic: assume the mapping σ—up to finite accuracy and precision—may be physically realized as a analog-to-digital conversion from sensors. A little known fact, the study of which refers to Buridan's ass, is that it is best to assume, among other deeper properties, that σ cannot be <u>total</u> [136].

Extend the discrete abstraction (σ) to whole timelines (by skipping over any true values lacking discrete counterpart). So $\sigma_f(Z_f^*)$ is some hypothetical discrete timeline a model might conceivably be able to represent and reason about.

For a model to be correct (*i.e.*, sound) it must satisfy, for every plan it considers executable, for each known fluent $f \in \mathit{Fluents}$, for every 'ground truth' behavior Z^*_f, with Z the modeled behavior of the plan:

$$Z_f \subseteq \sigma_f(Z^*_f), \text{equivalently:} \tag{3.39}$$

$$Z_f = \sigma_f(Z^*_f)\restriction_{\mathrm{Dom}(Z_f)}; \tag{3.40}$$

it is also useful to keep in mind the reverse direction, as of course even just imagined ground truth should still be kept separate from the planner's grasp:

$$(\bigcup \sigma_f^{-1}(Z_f)) \cup ((\mathbb{R} \setminus \mathrm{Dom}(Z_f)) \times \mathbb{R}) \supseteq Z^*_f. \tag{3.41}$$

What the left-hand side of the last 'looks like' is a relation from values of time to plausible Real values of each fluent at that time; when the model declines to take a guess, then every value for the fluent is plausible. The notation is all very baroque for stating such a notion so very far from earth-shattering as what it means for a model to be correct. Indeed, it barely looks like the notation has said anything of significance at all. Consider the following claims though.

Lemma 3.29. *Keep in mind the assumptions that discrete abstractions be physically realizable, and operate in time-independent fashion. Discrete and sound models of continuous change are never instantaneous.*

Corollary 3.30. *Well-motivated, discrete, sound models allow for—and consist mostly of—durative change. In other words, instantaneous discrete sound changes never abstract—in the limited sense of abstraction above—a continuous reality.*

Proof. For simplicity ignore the vector space issue (assume $k_f = 1$). Argue by contradiction. So assume a witness of instantaneous change in a discrete model corresponding to a continuous reality.

For notation, let $Z_f(s) = u$ and $Z_f(t) = v$ be two discrete values of fluent f with timeline Z_f that hold over meeting intervals, *i.e.*, with $s \in S$ and $t \in T$ such that intervals S and T meet at time t^*. Consider any true timeline Z_f^* of the fluent, which is by assumption both continuous and not a counterexample to soundness of the model (so $Z_f \subseteq \sigma_f(Z_f^*)$).

From Lamport [136], we may assume that the pre-images of the fluent values, $\sigma^{-1}(u) = U^*$ and $\sigma^{-1}(v) = V^*$, are non-meeting proper intervals (*i.e.*, the plausible Real values of the fluent before and after the change are intervals, but, non-meeting). The core reason is that it is impossible to build a device that decides in finite time whether a given Real value is, or is not, greater than any particular threshold. Such devices are called *arbiters*, and Lamport proves that arbiters are as mythical as deciders of the halting problem. The statement of the associated semantic puzzle begins: Place Buridan's ass precisely equidistant from food and water. Buridan's ass is (fatally) lazy. In particular, the hypothetical animal dies whilst contemplating forever which of the food and water is closer (for notation, Buridan's ass dies while trying to push sensor readings through a physical realization of σ to obtain the value of the fluent proposition "Is the food or water closer?"). It *sounds* as if the animal is simply too dumb to live, and that it is easy to circumvent the difficulty of deciding which is closer. Like most such puzzles the truth goes far deeper: see Lamport's excellent treatment.

So there exists a non-empty (perhaps degenerate) interval W^* such that U^*, V^*, and W^* are disjoint and their union is an interval (with W^* lying between U^* and

V^*). Then note that the left-limit at t^* belongs to at most $U^* \cup W^*$; the right-limit belongs to at most $V^* \cup W^*$. (Without loss of generality, because our aim is only to demonstrate non-existence of the two-sided limit, assume that both one-sided limits exist.) In particular note that the limit at the meeting time only exists if both the left -limit and right-limit belong to W^* (which will mean that W^* is indeed degenerate).

At least one of S or T contains t^*. Then either the left-limit $\ell = \lim^{-}_{x \to t^*} Z^*_f(t^*) \in U^*$ is a pre-image of the before-value u—and so not in W^*—because it is the earlier interval S that contains the meeting time ($t^* \in S$), or the right-limit $r = \lim^{+}_{x \to t^*} Z^*_f(t^*) \in V^*$ is a pre-image of the after-value v—and so not in W^*—because it is the later interval T that contains the meeting time ($t^* \in T$). That is, one of the two limits is just the direct value of the fluent at that time, which the model declares as corresponding to only one of u or v. Therefore the two limits are not equal, $\ell \neq r$, as at least one of them does not belong to W^*.

Hence the limit at the meeting time is undefined: contradicting, which suffices, the assumed continuity of ground truth. □

This perspective allows us to be confident that our semantics will always be, more or less, *sound* with respect to the real world. That is, taking the real world to be fundamentally continuous goes a very long way towards getting every question correct to high precision. Then the semantics are conservative/pessimistic. (If we deem something a solution there really is every reason to suppose it could work in truth.) On the plus side that means such planners will reach the desired conclusion in domains such as the *bomb-defusing* domain depicted in Figure 3.5: trying to cut two wires perfectly simultaneously is physically impossible (failing to realize which leads to explosion).

The downside concerns reasonably natural models we would rather, in contrast, call solvable. Namely, in the *soup-bowl* domain, we would like to say that a bowl of soup may be lifted from both sides without spilling its contents [182]. Unfortunately, in essense, entirely reasonable and natural formal models of the two domains are easily the same up to renaming of symbols. (Lifting two sides of a bowl and cutting two wires easily look identical to a domain-independent planner.) That is, we want to label the two domains differently, but our hands are tied: we must consider them equally solvable/unsolvable.

Note that *either* choice may be extended to a logically consistent formal theory. Hence it is important to make this choice of semantics abundantly clear, and, as much as possible, evident in external syntax. That is, it should be obvious to a domain modeler whether this is the kind of language that makes the bomb-defusing domain easy to model correctly, or the kind favoring the soup-bowl domain. In other words, we should aim for formal syntax exposing the idea that reality is fundamentally continuous.

The exactly opposite alternative is of course to imagine that reality is fundamentally discontinuous (a.k.a. discrete, quantum, digital). This is the sort of philosophy that favors the soup-bowl domain. Now, we can still model any domain in any formalism: it is just that we pay a price. Then it is interesting to look a little deeper at the price we pay here. Perhaps the single best counter-to-continuous-and-vaguely-realistic application of planning technique is the domain of digital electronics.

Digital Electronics Viewed Continuously. For digial electronics, a/the natural philosophy concerns *square waves*. (Piecewise-constant functions with values only 0 or 1, changing at most at, and instantaneously at, multiples of some constant Real

μ.) Our philosophy, and formal languages, forbid the modeling of square waves: all changes to fluents must have positive duration. We can, though, at the level of philosophy, permit, for example, *trapezoidal waves*. (Continuous, piecewise-linear, bounded by 0 and 1, non-zero slope only in $k\mu + [x, y]$ for arbitrary Integer k, and constant Reals x, y, and μ, with $y - x \leq \mu$.) Subsequently we can blank out the sloping regions, yielding something almost like a square wave. (Piecewise, 0 or 1, continuous where defined, undefined at most for handfuls of gate delays surrounding clock ticks.) In other words, what were instantaneous transitions become instead very short duration transitions; replacing degenerate intervals by tiny intervals everywhere is a straightforward manipulation. Interestingly enough, the model thereby obtained is a (i) more accurate description of circuit dynamics (we could, for example, consider calculating the optimal clock frequency, or even perhaps have the machine dynamically adjust clock frequency [can we clock branch -free machine code faster?]), and (ii) essentially no more difficult to reason with (we anyways had to reason about proper intervals, if anything, it is only easier to address always nonzero duration intervals).

The hypothetical sacrifice seems, then, to be bounded by the loss that stems from simply being nonconventional. So, that the sacrifice (of applying a fundamentally continuous view to a/the canonically discrete domain) is tolerable, perhaps interesting, is a significant point in favor of our treatment.

Naturally, that we claim we have a particular philosophy hardly guarantees that the formal theory is in fact faithful. There are indeed quite a few dubious oddities of the formal definitions to consider carefully; we briefly discussed several, for example, at the beginning of this chapter. One particular oddity is not elsewhere resolved, and thematically fits well here, *i.e.*, next.

3.4.2.2 Intervals of Definition versus Extent

A striking oddity of the formal definitions concerns corresponding behaviors. More specifically the interval of definition of a lock, denoted by the relation $\tilde{\in}$, would appear to conflict with the definition of vault transition functions. That is, elsewhere and intitally we say locks have right-half-open temporal extents, *i.e.*, regarding the precise hand-off between when one releases and the next acquires. Such statement renders the later definition of $\tilde{\in}$ (perhaps) counter-intuitive. In notation, an 'obvious' relation to write in support of corresponding behavior is:

$$(\tilde{\in}) \neq \{(t, \ell) \mid \mathit{Readable}_\ell \text{ and } t \in [\mathit{Acquired}_\ell, \mathit{Released}_\ell)\},$$

because that matches the notion that locks have right-half-open temporal extent from their acquisition-time to their release-time. (That is, intuitively speaking, from one angle, the interval of definition for a read-lock 'should be' its interval of extent, and, for a write-lock, empty.) Then in contrast, consider the relation we in fact define:

$$(\tilde{\in}) := \{(t, \ell) \mid t \in \text{ if } \mathit{Readable}_\ell \text{ then } [\mathit{Acquired}_\ell, \mathit{Released}_\ell] \text{ else } \{\mathit{Released}_\ell\}\}.$$

There is a perfectly reasonable explanation for why the expressions differ. The **interval of extent** of a lock is just what it sounds like: when the fluent must not be further accessed (because further concurrent access would be formally ambiguous/unpredictable), *i.e.*, when locked. The **interval of definition** of a lock is also what it purports to be: when the fluent's value is surely known/predictable, *i.e.*, when unambiguously defined by the formal model. The two differ notably because

we assume continuity, so, in particular and short: the *interval of definition* is, essentially, the closure of the *interval of extent* with respect to the limiting equation ($f(t) = \lim_{x \to t} f(x)$) defining continuity of functions.

Incidentally, correctness here is in regards to intuition: we already went through the exercise of demonstrating internal/logical consistency of the definition of corresponding behaviors (see Propositions 3.9 and 3.10). This exercise is about external/physical consistency with the claimed philosophy. The proof is long only to be at a sufficiently precise level that similar attempt for temporal PDDL would fail. The property itself is *supposed* to be intuitively clear, relative to grasping the arguably subtle distinction between reality, philosophy, and formalism.

Lemma 3.31. *Corresponding behaviors of* CTP *and* ITP *are discrete abstractions of continuous functions, in the sense of Lemma 3.29. Moreover, such are 'maximally assertive' as defined in the proof, meaning: the structures encode all and only those predictions that are formally warranted.*

Proof. Recall the definition of the behavior corresponding to a formal execution X: $\text{Behavior}(X) = \{(f,t) \to \mathit{State}_Y \mid t \mathrel{\tilde{\in}} \mathit{Vault}_Y(f) \text{ and } Y \in \text{Rng}(X)\}$. The claims are that 'any' other such mapping from an exection X to a function of that type—the intuitive type for the 'meaning' of a plan—asserts either too much or too little. To assert too much would be to violate abstraction from a continuous 'reality'. To assert too little would be to say anything less than maximally possible subject to that.

Maximally Assertive up to Decomposability. We should first clarify a methodological meta-constraint: we desire that these intuitive semantics for formal executions be *compositional*. At first pass that means just that the behavior of an execu-

tion must always be a superfunction of the behavior of any of its prefixes. More specifically we mean that any competing definition must be incrementally computable in the sense that there exist an update rule r such that: $\text{Behavior}((X)_{[0,n]}) = r(\text{Behavior}((X)_{[0,n-1]}), X_n)$. It follows—upon considerable reflection—that our only competing choices for the definition of corresponding behaviors vary just the meaning of the relation ($\tilde{\in}$).

So the task comes down to identifying the maximal expression for the so-called "interval of definition", ($\tilde{\in}$). To be more precise the task is to deduce from the current lock the largest set of times for which the current value (as recorded in the corresponding state) is guaranteed—by the domain modeler—to be an accurate discrete abstraction of the imagined, continuous, behavior.

Then recall, from above, the two candidate expressions for the interval of definition. The former is the 'minimal' candidate: 'clearly' whenever a fluent is read -locked we could (and so must) assert that the fluent's value is predictable. Then it suffices to demonstrate that the maximal candidate is as is written, namely: the interval of definition consists of adding in just the release times of locks. (That is all and only the difference between the expressions; particularly, even write-locks, perhaps counter-intuitively, possess an interval of definition.) First let us elaborate correctness of all but the slightly tricky part, and second discuss what is happening with the behaviors of fluents at the release times of locks.

Keep in mind that the value stored in the corresponding state by the machinery is the value that will persist after the lock is released. In other words, the "current value" as recorded by the machinery is, in particular, the left-limit of value at the release time of the current lock.

Case: Read-locks, $[\mathit{Acquired}, \mathit{Released})$. During the extent of a read-lock there is no issue in asserting that: the true behavior of the fluent really will correspond to that constant value the formal model ensures over that interval. In short, the full extent of read-locks can, and so must, be included in $(\tilde{\in})$.

To be exceedingly precise, first note that read-locks ensure no other situation of the formal execution will ever declare a write-lock on those times read-locked. So no change will ever said to be occurring at those times: the formal value of the fluent is had by *persistence*. Hence—by the Frame Axiom—(1) we may assert that (2) the domain modeler means to guarantee that (3) the real behavior of the fluent will also merely *persist*. The two notions of persistence, in turn, mean most precisely that the true behavior will be 'sufficiently constant' with respect to the formal behavior: in the sense that it is the domain modeler that guarantees that whenever the formal value is had by persistence, and so is constant, then indeed the real value is had by persistence, meaning "constant up to some tolerance", *i.e.*, meaning that the real behavior lies strictly within the preimage (with respect to the domain modeler's notion of discrete abstraction) of the formal value. In short the 'contract' of domain modeling simply includes, in our formalisms, satisfaction of the Frame Axiom: from which immediately follows the case.

Case: Write-locks, $(\mathit{Acquired}, \mathit{Released})$. From the preceding discussion of philosophy, *i.e.*, for continuity, we know that any periods of change must be modeled as proper intervals of undefined value. Meaning that, during at least the interior of a write-lock, we must *not* assert that the corresponding behavior of the fluent is defined. In short, the interior of write-locks must be excluded from $(\tilde{\in})$.

To be precise, the technical point to be verified is: the machinery ensures that existence of a write-lock in the midst of an execution ensures that no other situation of that execution will ever declare any other locks on those times lying within the extent of the write-lock. Hence the formal value of the fluent over the extent of a write-lock is had by the definition of *assignment*. Then it is the responsibility of the domain modeler to ensure that the true value is also had by the corresponding notion. So for reference, the meaning of an assignment—an *axiom* of our intuitive semantics—with respect to true behaviors is: the true behavior of the fluent during an assignment is some arbitrary continuous function, connecting the arbitrary initial value it had at the time the effect began, to some unknown final value it will have at the time the effect finishes, for which it is guaranteed—by the domain modeler —that the discrete abstraction of the unknown final value will indeed be as given in the formal assignment statement. Therefore the case: we cannot, and so must not, make any definite predictions about the strict interior of the interval over which write-locks hold.

Case: Write-locks, [*Acquired*].. By the time locks are acquired the former value is forgotten/overwritten. So the acquisition-time of a write-lock must also be excluded from ($\tilde{\in}$).

To be precise, we *can* predict (at a higher level of understanding), due to continuity, the value of a fluent at the acquisition time of a write-lock. However, due to the specifics of asking that semantics be compositional, we lack the requisite flexibility to make that prediction from this case. Instead it is the former lock that can and so shall take responsibility for making the prediction.

Case: [*Released*].. To be perfectly clear: The release time of a lock does not 'belong' to that lock. Rather it is the next lock that 'owns' the time. Nonetheless we can and must include release times in ($\tilde{\in}$).

Consider for a moment that the next lock could be either a read-lock or a write -lock. As it so happens, the type is immaterial: *i.e.*, just recall continuity. Specifically, even though the fluent might be changing at that time, *i.e.*, write-locked at that time, it is also true that the fluent cannot possibly—by the assumption of continuity—*immediately* change its value. So regardless of the types of locks either before or after the release time in question, the release time can, and so must, be included in ($\tilde{\in}$).

To be more precise note that always the left-limit, right-limit, and actual value must all be the same in the true behavior. Then in particular we know the discrete abstraction of the left-limit of the release-time: such is the current value as recorded by the machinery. Hence we know, too, the (discrete abstractions of the) actual value and right-limit at the release-time. Therefore we may correctly assert—and so must—that the fluent's value will be predictable/defined at that time. □

Is such fuss over grabbing a single extra point in time really warranted? Indeed: a behavior is supposed to be a complete account of what humans intuitively perceive to be the meaning of temporal plans.

So consider automatically generating driving directions. Driving is an inherently uncertain activity. Then we cannot view directions as a naïve schedule of actions by start-times. That said, we could think of directions as consisting of actions scheduled by a far more abstract notion of the set of 'clocks' that emit a single tick when each senses the appropriate road signage (landmark, *etc.*).

We can generate such structures starting from even our simple sort of temporal planning (a CTP-planner would make sense, in particular). First we would generate a nominal temporal plan, based off of expected durations.

Secretly, *i.e.*, outside the planner, we know full well about the uncertainty. (Regarding that uncertainty: the hypothetical system as a whole would do exactly what real navigation systems do, which is replan when things go wrong.) So secondly we would take the plan it produces, calculate its <u>behavior</u>, and extract from that those changes in fluent values that are simultaneous with the nominal start-times.

Thirdly we would apply background knowledge about what fluents humans can and cannot readily sense, and so filter the set of expected observations down. If that was still too many observations per time point, then we might rank them by usefulness and just pick the top k or some such.

In the end we would take the expected, filtered, observations and substitute for the start-times. The result is a set of driving actions 'scheduled' by observations, *i.e.*: driving directions.

Then reflect carefully on what the <u>behavior</u> of a driving plan would look like. A driving plan is a sequence of actions. Each immediately write-locks, in particular, the location of the car: in general all fluents are changing at essentially all times (particularly the location of the car is always changing). So essentially the only predictions we will be able to model—in a strictly discrete model—are those instantaneous values that these fluents have at the precise instants that control of each gets handed off between each pair of temporally adjacent driving actions. Hence $\tilde{\in}$: to ensure that when write-locks meet, we declare that the value in the instant between them is known.

To wrapup:

- Our philosophy is that reality is fundamentally continuous.
- Our strictly discrete formalization thereof is entirely faithful.
- We can *prove* that claim—when pressed—down to the smallest of details.

Such is a nice little improvement upon existing theory for an important niche. Specifically our formal treatment is rather convenient for better working with that fragment of temporal planning closest to classical planning.

It is hard to appreciate the value of the contribution without considering the meta-problem resolved. So next we shall harangue temporal PDDL. The balance of the matter, is, of course, that its merits far outstrip its flaws. Indeed, our treatment cannot stand apart; formally defining syntax is, for example, quite important, and we do not. Note that it is easy to read *semantics* as more important than *syntax*: that is false. In practice, concrete example communicates meaning far more effectively than abstract notation. (Syntax is the 80%; semantics is the 20%.) In short our aim is merely to improve upon select pieces of temporal PDDL, not to supplant it. Then to understand the following critique, keep in mind that the intent is only to reflect well upon the value of our contribution. In particular, the following is nothing like a fair and balanced discussion of the merits and flaws of temporal PDDL.

3.4.2.3 Selected Philosophy of Temporal PDDL

The following is a reasonably accurate portrayal of several of the key semantic choices underlying the formal details of temporal PDDL [71].[24]

[24]The specification, to be accurate, defines 5 languages. The syntactic features are durative actions, numeric fluents, and continuous effects. The last are only possible/interesting with both of the former (so 5, not 8, possibilities). Support for continuous effects, particularly in any general fashion (*i.e.*, more than strictly linear change), is notably lacking (so 4 possibilities remain). First-

No Moving Targets The value of a fluent is *undefined* while changing, *i.e.*, "is a moving target". The upshot is that *mutex* means essentially the same thing here and there: primitives are mutually exclusive when either writes to a dependency of the other.

Nonzero Separation Mutually exclusive (dispatches of) primitives must be separated from one another (in time) by a proper interval.

Bounded Precision No physical agent can guarantee dispatching a primitive arbitrarily precisely.

Then for one thing, guaranteeing that one primitive occur strictly after, yet arbitrarily close to, another is considered deific. Formally: so there must exist some constant $\varepsilon > 0$ such that the separations between mutex primitives are at least ε long. By "constant" we mean that a planner's task is to find a plan for a fixed agent: ε, *i.e.*, the inverse of a clock frequency, is chosen prior to planning.

Instantaneous Primitives All primitives (effects, changes, transitions, events, "to do", . . .) are, 'without loss of generality', instantaneous.

Durative Constraints Constraints (invariants, states, "to be", . . .), but not effects, may be durative. Durative entities are not "primitive": hence exempt from the rules governing primitives.

Note that any dispute is philosophical: these statements amount to *axioms*.

---

class numeric fluents—in contrast with, say, using numbers only to track (complex) notions of plan quality—introduce difficult computational issues [101]. For example, first-class treatment easily permits encoding/simulating arbitrary Turing Machines (*i.e.*, write out the number in binary). Much of the state of the art understandably avoids such unbounded use of numbers. So in short we, by and large, pretend the specification defines just 2 languages: without, and with, durative actions.

3.4.2.4 A Practical Shortcoming of Mixed Discrete-Continuous Planning

Abstractly enough, the philosophy underlying temporal PDDL is—in isolation—entirely reasonable. Specifically, imagine the reality of causality to consist of just *continuous and durative processes* separated by *discrete and instantaneous events*. For example, movement is easily perceived as the durative and continuous process of navigating from here to there:

1. *decide to start* navigating, subsequently
2. *while time passes*, actually carry out said *process*, and finally
3. *decide to stop* navigating.

In general we could always model reality this way, which is what is meant by "reasonable"; Hybrid Automata, and moreso PDDL+, are formal treatments [70, 111].

That said, there is a significant drawback to mixed discrete-continuous planning: its computational characteristics. Particularly, formal treatments of such are too general, relative to present domain-independent automated planning technique, to permit effective implementation. (Implementations are domain-dependent or ineffective [12, 137, 154, 168].)

Oddly enough, at the same time as being overly general, said formalisms fail to subsume present technique. By which we mean that the details fail to permit natural expression of a key special case: discrete temporal planning problems. Such is a notable shortcoming. A specific negative consequence is that many so-called temporal PDDL planners deliberately alter its semantics, procedurally, in order to address the shortcoming. That greatly hampers the value of empirical analysis.

To fix this, it is theory that needs to change: the deviations are too pronounced to label as 'bugs'. Specifically, the lesson here that emerges from practice is that

durative discrete change is too important to ignore (that is, too important to support only *via* compilation). So for concreteness:

"(over all ...)" should be legal syntax within ":effect".

Formally the (nonobviously) relevant theoretical point to be made is:

Proposition 3.32. *The labeled graphs called Hybrid Automata (which best formalize temporal* PDDL*, see* PDDL+ *[70]) are disjoint from both (1) the labeled graphs that serve as formal semantics for Conservative Temporal Planning, as well as (2) the labeled graphs that serve as formal semantics for Interleaved Temporal Planning.*

Proof. Discrete change must occur instantaneously in Hybrid Automata (which is also true of temporal PDDL and PDDL+). Discrete change must occur non-instantaneously in both Conservative Temporal Planning and Interleaved Temporal Planning. □

In other words, Hybrid Automata associate (proper) intervals of time with only vertices; conversely, we associate (proper) intervals of time with only edges.

In a sense the choice is arbitrary. So in isolation, the formal semantics of Hybrid Automata are entirely reasonable. However, for the purpose of temporally generalizing classical planning technique, it is evidentally (*i.e.*, in practice) significantly preferable to ascribe durations to edges/actions.

At least, such seems to us to be a/the right way to reconcile temporal planning theory with the current practice concerning the precise meaning of *discrete* and *continuous*. Specifically, in theory, the two are taken as perfect antonyms. However, in truth—the truth that practice recognizes—neither *discrete* nor *discontinuous* need imply one another. Explaining such is our real purpose here (in contrast

with the means, haranguing PDDL, to that end). Particularly the notion that discrete (to a planner) need not imply discontinuous (to a domain modeler) calls for the significant mental gymnastics we began with (*i.e.*, we must separate the levels of interpretation, because indeed discrete does imply discontinuous within one level of interpretation).

3.4.2.5 Selected Semantic Flaws of Temporal PDDL

While the most basic philosophy underlying Temporal PDDL ("mixed discrete-continuous planning") is entirely reasonable, the specializations made by the specification are dubious in certain respects. That is particularly true when we consider a key special case: exclude continuous change. We take issue with:

- It is dubious to insist that all effects be primitive and instantaneous, *cf.* "Durative Constraints" and "Instantaneous Primitives", as such loses natural expression of processes.

  We *could* 'express' processes by compiling into just *events* and *invariants*. However, modeling causality as just invariants+events is notably less well-motivated than modeling causality as just processes+events.

- Taking invariants ("over all" conditions) to have the semantics of processes (*cf.* "Durative Constraints"), that is, exempt from the rules governing primitives, is dubious. Such is especially true when invariants are the only syntax that ends up with process semantics.[25]

[25] By "process semantics" we mean, expressed syntactically, that the specification implicitly asks us to reread "(over all ...)" conditions as "(over all (left-limit ...))" conditions. So more formally the PDDL-syntax "(over all ϕ)" in theory has meaning resembling "$\forall t : \phi(\lim^{-}_{x \to t} Z(\cdot, x))$", instead of the conceptually simpler expression "$\forall t : \phi(Z(\cdot, t))$". (By "$Z(\cdot, t)$" we mean the partially defined state, *i.e.*, skip over any undefined fluents, at time t with respect to the behavior Z of the plan of

For example, two actions, starting simultaneously, respectively with condition "(at start (or p q))" and effect "(at start p)" are said to violate the "No Moving Targets" rule. Yet the quite similar condition "(over all (or p q))" is held to be exempt; PDDL permits that invariant to be concurrent with instantaneous effects on "p" or "q". Such exemption is dubious at best.

Consider that it is natural to interpret "(over all ...)" as meaning something like "(forall (?t - all) (at ?t ...))".[26] That is, it is natural to interpret durative conditions as being equivalent to the corresponding infinite set of instantaneous conditions. Such is, counter-intuitively, and to no technical advantage in context, false of PDDL. In short it seems strictly better to just call invariants "primitive": treat instantaneous and durative conditions the same.

- Insisting that plans be plausibly executable by physical agents ("Bounded Precision") subsumes "Nonzero Separation". Usually we prefer to omit redundant axioms.

- The "No Moving Targets" rule is dubious given "Instantaneous Primitives", doubly so given "Bounded Precision". Intuitively speaking, when all change is instantaneous, there is no "moving" to speak of. The relevance of bounded precision is as follows.

- An additional consequence—besides ε-separation—of "Bounded Precision" is that physical agents cannot guarantee simultaneity. To hit both birds with one stone: temporal planners *ought* to verify that the nominal schedule will

interest.) Note that, while exempting *limits* from the "No Moving Targets" rule does make sense, excluding expression of the conceptually simpler kind of invariant does not.

[26] Quantifying through time values, and specifying intervals besides start/end/all, is not legal PDDL-syntax. ZENO is perfectly happy with such notions [168].

work even with tiny errors (a.k.a. perturbations/delays) introduced throughout. Firstly: efficiently and correctly enforcing that constraint is tricky, to say the least. Secondly: if/when enforcing that constraint, the "No Moving Targets" rule becomes meaningless (indeed, detrimental), as follows.

Consider any two instantaneous effects scheduled closer than ε to one another. It is natural to suppose that the space of perturbations the planner should consider may be usefully divided in half: one half for each of the two possible orders the effects occur in. Which is to say that it is natural to exclude the third possibility: the effects occur in neither order, *i.e.*, they occur simultaneously.[27] For analogy, let us pretend that flipping a coin has only two possible outcomes. In short, there is no value, in practice, to worrying about perfect simultaneity of instantaneous effects.

Then note that, given both "Instantaneous Primitives" and "Bounded Precision", the "No Moving Targets" rule is at best irrelevant (if we exclude consideration of perfect simultaneity), and at worst detrimental (when consideration thereof leads to rejecting reasonable plans).

3.4.2.6 Counterexample to VAL*-Soundness: The Bomb-Defusing Domain*

It is challenging to merely formalize, let alone (correctly) implement, the philosophy of Temporal PDDL. Here we shall show that, indeed, even the specification fails to correctly formalize its own philosophy: Temporal PDDL is unsound with re-

[27]A formal argument against simultaneity leverages: independently chosen Reals are *precisely* equal, *i.e.*, to infinitely many significant figures, with probability 0. Clearly *infinitely improbable* is close enough to *impossible* for any practical purpose. As it turns out (typically counterintuitively), even for pure theory it makes sense to *define* $(1/2)^\infty := \lim_k (1/2)^k = 0$. *I.e.*, *infinitely improbable* and *impossible* are perfect synonyms in Real Analysis. *Nonstandard Analysis* permits distinguishing, *i.e.*, there "$0 < (1/2)^\infty < (1/2)^k$" holds for finite k.

spect to ‘itself’. (So we find it especially compelling to forgive, and work towards understanding, noncomformance of so-called implementations.)

For concreteness, consider a compelling potential application of planning technology to modern warfare: *Semi-Autonomous Robotic Nuetralization of Improvised Explosives*. Fielding such an application involves numerous challenges, easily far beyond the scope we consider, only the least of which concern temporal reasoning. So the motivation is one-way. Specifically consider the super-special case where we have a perfect model of the bomb in question, perfect motors, perfect sensors, devil-may-care human operator(s), and so forth.

Now, some bombs are such that water nuetralizes the fuse, rendering the bomb useless; alternatively water might nuetralize the explosive material itself. So sometimes the best plan is to dunk the bomb in the nearest toilet. Other bombs are low -power enough that the best plan is to deliberately detonate them in a reinforced chamber. Occasionally physical access to an electronic fuse is possible (always so in Hollywood): the best plan could be to cut (and/or short) wires.

Any wrong choice, though, of plan is typically fatal (achieving that is of course a large part of the goal of the bomb-designer). For example, cutting the wrong wire is fatal. Moving bombs with fuses containing accelerometers is surely also a bad idea. Any chemist worth her salt could name explosives that react despite, or even because of, aqueous environment.

So making the right choice is crucial. In particular a technical challenge that would really matter is making the best possible choice given *uncertainty* about the nature of the bomb. (*Robots* are especially compelling here insofar as few, if any, are willing to declare a dollar figure on human life.) That is, it is—of course—entirely unrealistic to suppose that we might, in general, have perfect (or near-perfect)

knowledge of a bomb's design. But just suppose we were so fortunate. (Perhaps that is because the larger system is choosing to individually reason about a limited number of especially plausible designs, *i.e.*, given, say, background/expert/operator-provided knowledge on likely fuses.) Surely, given a perfect model, we wish to (have our planners automatically and) perfectly evaluate which nuetralization plans succeed and which fail.

Specifically consider the following anti-nuetralization-motivated modification of an electronic fuse. Take the original design (say red), duplicate it (say blue), and everywhere insert additional circuitry to check that each (red) wire and its (blue) twin are always equal. So if ever the equality check fails, then immediately detonate.

Then counter-terrorist plans working from physical modification of the circuitry (*i.e.*, cutting and shorting wires) are *greatly* hampered by the modification. Particularly, such plans must somehow preserve all these *invariants* equating the digital logic values of the red and blue wires: to cut any wire, we must cut its twin *precisely simultaneously*. Such is a conclusion we want the planner to automatically reach on its own power (meaning we wish to model the domain without exploiting the problem-specific observation), and, from there, make the correct follow-up: cutting wires is a very poor approach to disarming such a bomb.

Well and truly modeling the domain and problem in temporal PDDL is no simple thing. One particularly vexing syntactic shortcoming is effective support for modeling of *domain invariants/trajectory constraints* in general. Let us claim without proof that it is possible to pull the feat off.

To give some small taste of the details, see Figure 3.5. The figure is attempting to sketch a planner's perspective on the behavior of two candidate plans: (1)

cut a red wire and its blue twin precisely simultaneously (at a moment when the compilation of the invariant-checking—red wires must equal blue wires, else explosion—could surely[28] formally permit access) versus (2) dunk the bomb in water (which, suppose, bypasses relevance of the circuitry, because the explosive itself is nuetralized). So the point here is just that PDDL planners ought to consider the perturbations of the wire-cutting plan; it is not enough to gaze at just the nominal behavior depicted.

Particularly, while the nominal behavior passes all the formal tests, no perturbation would. Then as long as the planner considers such perturbations, it will realize the superiority of dunking over wire-cutting for this particular bomb, and so surely make the correct choice. Otherwise the planner will mistakenly sign off on correctness of the wire-cutting plan, perhaps even preferring it. Let us abstract the mistake: the wire-cutting plan is incorrect because it *requires precise simultaneity of instantaneous effects belonging to distinct actions*.

Very long story short: the philosophical point named "Bounded Precision" is indeed well-motivated. In particular, physical agents cannot, in general, guarantee simultaneity of instantaneous effects. This should all be relatively unsurprising. What *is* astonishing however is that the specification—despite similar discussion—

[28]In a sense the detail of Figure 3.5 regarding compilation of invariant-checking is unnecessary. That is because invariants are already supposed to permit concurrent modification of the underlying fluents, *i.e.*, to possess "process semantics". However, such point is preciously fine and quite counter -intuitive to boot. In particular, implementations do not respect such reading of "(over all . . .)": *nor do we*. While such semantics are not, in practice, directly expressible, we can nonetheless exploit compilation techniques to obtain (close enough) expression thereof. We note that, in the design-space of temporal PDDL, appeal to even rather nasty compilation (for lack of syntax) is legitimate, if not encouraged. Specifically the figure is sketching the general solution—following Halsey [72] —to reducing limit calculations into non-limit calculations in the context of only discrete change. Doing so is conceptually easy, because with only discrete change, functions of time are piecewise-constant (and limits of constants are just the constants themselves). The details, though, are messy enough to take note of, as similarly discussed at far greater length by Smith [186].

fails to correctly formalize this particular consequence of “Bounded Precision”. We can more or less formally prove *that* in reasonable space. Indeed, we already have, long ago: at the beginning of this chapter. To recap:

Theorem 3.33 (PDDL is Unsound)**.** *The given formalization of temporal* PDDL *is unsound with respect to its own philosophy: it permits problems solvable only by scheduling independent actions precisely simultaneously, in violation of “Bounded Precision”. As manually validating plans against the formal semantics of temporal* PDDL *is arguably too lengthy to be reasonable, consider instead the behavior of* VAL *[119], which exists precisely to automate such validation. Then formally:*

There exists a temporal planning domain and problem in PDDL*-syntax such that every* VAL*-solution schedules instantaneous effects of distinct actions precisely simultaneously.*

Proof Sketch. Specifically, Figure 3.1 witnesses. In setting up that initial motivation we claimed that implementations reject solvability of the syntax. That is only *almost* true. VAL, in particular, accepts solvability of the given problem, which the reader may verify readily: the syntax of the figure is entirely legitimate (so copy-/paste the domain, problem and plan into separate files, and run the program on them). That every perturbation is a nonsolution is easily seen.[29]

The reader may also of course just manually simulate the formal specification, which result will further confirm. □

[29]Technically VAL preprocesses plans in an odd manner regarding events closer than ε/c together (it rounds the later dispatch-time down to the earlier one), for $c = 10$ as I recall, which is plainly a bug (*i.e.*, has no basis in the specification, and precious little basis in philosophy). Particularly we may exploit the behavior to convince VAL to sign off on plans that it would (and should) normally reject due to violating ε-separation. In short that is because ε-separation is not a transitive concept, and the design assumes it is. Entirely disabling that preprocessing has unclear downstream effect in general. As far as the proof that VAL is unsound goes though, such disabling is fine as far as the counterexample goes, because the flaw has to do with “(over all . . .)” conditions, which are anyways supposed to be exempt from the details of ε-separation.

3.4.2.7 Discussion: Explanation and Resolution of Unsoundness of VAL/PDDL

The 'reason' that VAL fails here is straightforward enough: to correctly formalize/implement "Bounded Precision" we must start by distinguishing between the nominal and perturbed behaviors of schedules. However, the official plan validator (and for that matter the formal side of the specification) only associate plans/schedules with a nominal behavior. This is no simple oversight: calculating infinitely many (perturbed) behaviors is hardly feasible. So we should desire formalization and proof of *rescheduling* flexibility: to implement "Bounded Precision" we could aim for constraining nominal behaviors far enough that of necessity all perturbed behaviors would pass muster as well. At *that* point we could justify checking plan correctness by calculating only a single behavior.

Then we explain the error of the formal specification by saying that it skips proper demonstration of "Bounded Precision". To be specific, the specification assumes, falsely, that, in the absence of continuous effects, the appropriate theorem follows just from its formalization of ε-separation between establishers of instantaneous conditions and the conditions themselves.[30] However, the desired theorem only *almost* follows.

The easiest way to disprove is just to exploit the exemption of durative conditions from the rules regarding primitives (*i.e.*, as Figure 3.1 does). As it turns out, that exploit is a bit of red herring: the desired theorem remains false even

[30]The details of ε-separation are, incidentally, far from entirely clear, even subsequent to understanding the source code of VAL. That is because it is apparent that the code is buggy with respect to ε, hence sheds no light. Which brings up the point that, unfortunately, such bugs greatly undermine the proof strategy of our theorem (appeal to the behavior of VAL). So for one thing there really is some value to manual verification. Slightly more realistically we refer the reader to Haslum's Independent VALidator (InVAL) [98]. Haslum's contribution is straightforward and quite significant practically: we can be rather more confident in our conclusions about PDDL when manual simulation of the specification, VAL, and InVAL *all* appear to agree. Unfortunately, InVAL does not support temporal plans (future work?).

when interpreting durative conditions as subject to ε-separation requirements. We emphasize the semantic detail anyways, because later work by that group implements different semantics regarding durative conditions. Namely, POPF enforces ε-separation with respect to durative conditions, and so in particular rejects solvability of Figure 3.1 [36].

That does not imply that POPF correctly implements "Bounded Precision": only that it avoids counterexamples of the kind given in the figure. Which is also not to say that POPF fails to correctly implement "Bounded Precision". Altering the interpretation of "(over all ...)" is simply inconclusive with respect to "Bounded Precision".

In fact POPF *is* reasonably held to correctly implement "Bounded Precision"; such is due to yet further reinterpretation of PDDL-syntax. To really demonstrate so we would need to first invent the missing formalization that it is in fact correct for, which technically involves substantial revision of many details (as could be expected from our preceding discussion). We would like to pretend that our ITP is said "missing formalization". To do so is patently unfair: we cannot complain about VAL's interpretation of Figure 3.1, yet in the next breath take POPF as implementing ITP.

Still, to capture the essence of the matter, albeit the authors of POPF surely (and rightly) disagree, what POPF does is to reinterpret PDDL-syntax by forbidding *instantaneous change*. So more specifically we prefer to perceive POPF as reinterpreting syntax as follows.

- "(at start ...)" is reread as roughly "(over $[\text{start}, \text{start} + \varepsilon\rangle$ ...)",
- "(at end ...)" is reread as roughly "(over $[\text{end} - \varepsilon, \text{end}\rangle$ ...)", and

- "(over all . . .)" is reread as roughly "(over [start, end$\rangle$ . . .)".

  This last has quite notable exception: when reading "all" as *all* creates a self-mutex action, then instead POPF shortens the interval by excluding the concerned endpoint(s). So also possible are the readings:

  - "(over [start, end $-$ $\varepsilon\rangle$ . . .)",
  - "(over [start $+$ ε, end$\rangle$ . . .)", and
  - "(over [start $+$ ε, end $-$ $\varepsilon\rangle$ . . .)".

We prefer this view as it permits a short story for correctness/soundness/reasonableness of POPF. Namely, we see POPF as correctly implementing a slight generalization of our Interleaved Temporal Planning. That, in turn, we take as soundly formalizing our philosophy upon reality (which naturally we call "reasonable"). Said philosophy is just that reality is *fundamentally continuous*; the key consequence is that *instantaneous change* is forbidden. Such is key as it directly ensures inexpressibility of simultaneous, instantaneous, change; with that ruled out we can take a different approach to the semantic puzzle that "Bounded Precision" refers to.

Our Resolution to Physicality. We take durations as *upper bounds on the inherent uncertainty of real values of time*. So our model for physicality of agency is *not* that start-times of schedules are to be perturbed. Rather, we view the inherent uncertainty of physical clocks as implicit in the statement of every proper interval. In other words, what we suppose will really take place is some rather faster change over some unknown, likely smaller, subinterval. To be certain the agent will succeed, we model the interval of change as pessimistically as necessary. In particular,

to ensure physicality across the whole language, we build into the language the assumption that intervals must be proper.

Which means that we 'force' domain modelers to obey the paradigm. That is, by having the formal language rule out instantaneous change we guarantee that every formal model that may be expressed can be plausibly interpreted as pessimistically bounding physical agency. Of course, an unaware domain modeler might put the 'wrong' numbers, still leading a planner to sign off on impossible plans. That cannot be helped: there is nothing magical to be done to ensure that models actually describe reality (without really taking observations, *i.e.*, trying, failing, and learning). However, we take this manifestation of the error as 'self-correcting'. Consider once more the example of bomb-disarmament.

To model the bomb-disarmament domain in ITP we will have had to place a nonzero duration on the time it takes to cut a wire. If we take a general approach to compiling in process-style invariants—which is painful, but possible—then we will be able to pull off modeling the operation of arbitrary electronic fuses, in particular we will be able to model the fuse of interest, which forces each pair of red wire and blue twin to be 'always' equal.[31] In particular we will have been forced to assert a nonzero duration for the maximum time the equality of a red and blue wire can be 'violated' without actually failing.[32] Should what is presumably an error

[31]*Bounded maintennance constraints* may be expressed in Real Time Logic by saying "∞-Always e-Eventually r-Always ϕ" to mean that ϕ must be 'always' true but for any number of at most e-duration witneses of exceptions separated by r-duration witnesses of satisfaction. (A weaker but interesting consequence is that ϕ will be satisfied at least r/e frequently.) Forcing ITP-planners to enforce such constraints *via* domain-modeling shenanigans is possible, in large part because we only ask that plans be finite (in general Real Time Logic is undecidable).

[32]It is also possible to tie the checking of equality to particular clock ticks, *i.e.*, physically by feeding the equality gate into a (clocked) *latch* (rather than by asynchronously debouncing the gate). For modeling, we could start with a different expression in Real Time Logic (albeit a modal logic is not ideal for modeling clocked digital logic). Note that voltages will naturally differ briefly and

have been made in specifying those numbers, then the model might permit the wire -cutting plan. That is because the model would say that the agent can guarantee the precision of independent wire-cutting actions (the first nonzero duration) to faster than the electronic circuitry can detect (the second nonzero duration). Such is "presumably an error" as the detection of the (quasi-)invariant by the electronic circuit will be precise to the level of *gate delays* (just one?): detecting when the two wires differ is lightning fast. That said, there do exist physical agents faster than lightning: just precious few.

Then, that the model is somewhat dubious is not a strong critique; it is just barely plausible in domain-independent context, and, when implausible in real application, then fixing it is straightforward (just fix the numbers). In contrast, fixing the same sort of modeling error starting from PDDL-syntax is far more involved. That is because *durative discrete change* is not legitimate PDDL-syntax. So we might have begun with instantaneous changes, because such is the only syntactic possibility. Then, only belatedly, we might come to realize that instantaneous change is indeed a very error-prone concept. At that point we shall be confronted with a conundrum. The nice way to fix the error would be to enlarge the durations of the not-actually-instantaneous changes. Temporal PDDL, however, syntactically clamps the durations of all changes to 0.

Naturally, there is a workaround (there is always a workaround, all planning formalisms are at a minimum PSPACE-complete), but we cannot muster the motivation to employ workarounds to model 'every' real world application. That is, we would prefer to use compilation techniques to fake instantaneous change rather than use

---

slightly (due to ambient magnetic fields, for example): the bomb designer will have had to build in *some* tolerance.

compilation techniques to fake durative change. By "we", note that "we", in effect, includes the author of *every* truly serious attempt at implementing an empirically effective discrete temporal PDDL-planner. We claim such, because, while several pretend at length that changes are instantaneous, the rules used to enforce "Bounded Precision", 'whenever' correct, are (we claim without proof) *formally isomorphic* to asserting that all are actually ε-duration, durative, discrete changes [25].[33]

3.4.2.8 Conclusions on ITP *versus* PDDL

Towards rounding out the picture: there is, of course, rebuttal. The war we wage here is a language war: there are no winners in (language) war. (We can formally prove that, perhaps surprisingly, by consideration of Kolgmogorov-Chaitin Complexity, the No Free Lunch Theorems, and so forth [27, 206].) For metaphor: Hammers suffice, but such hardly makes them the right tool for the job. In this case, the key question regarding purpose is the distinction between:

1. *Mixed Discrete-Continuous Planning*, *Control Theory*, *Operations Research*, *Real Time Verification*, (...,) *i.e.*, *Algebraic/Analytic Mathematics*, and
2. *Discrete Planning*, *Situation Calculus*, *Discrete Finite Automata*, *Linear Temporal Logic*, (*etc.*,) *i.e.*, *Combinatorial/Finite Mathematics*.

Or in short, a crucial question to ask and answer is:

Is time discrete or continuous?

[33] *Isomorphism* here is meant in a very particular sense far stronger than asserting Turing completeness subject to polytime-reductions and polyspace-bounds. Proposition 3.32 might shed light.

For us, time is, effectively, discrete (Corollary 3.23). Hence effects must be durative and, effectively, discrete.[34]

In contrast, for generalizing to continuous change it is convenient (but slightly fallacious) to retain expressibility of discrete effects by calling them discontinuous and instantaneous. That is, for such context, (experience proves) it is better to permit the expression of instantaneous discrete change [37, 70, 110, 137, 151]. We conjecture it would be better still to retain expression of durative discrete change (however there is no particularly apparent empirical evidence either way). Such is neither here nor there, as for us continuous change is well beyond scope. Still, for a bit of formalism, we find it compelling to approximate continuous equations such as:

$$f_c(t^{\in\mathbb{R}}) = \int_0^t x + \frac{1}{2}\,\mathrm{d}x = \binom{t}{2}, \qquad \text{by the discrete:} \tag{3.42}$$

$$f_d(k^{\in\mathbb{N}}) = \sum_{i=0}^{k} i = \binom{k}{2}, \qquad \textit{i.e.}\text{, it is the case that:} \tag{3.43}$$

$$f_d \subset f_c. \tag{3.44}$$

Then we note that languages such as Hybrid Automata typically prevent discrete fluents from ever having the form of f_d. The closest one may say is (note the rounding, the function is a step function):

$$\hat{f}_d(t^{\in\mathbb{R}}) = \sum_{i=0}^{\lfloor t \rfloor} i, \tag{3.45}$$

[34]Continuous effects in discrete time can be simplified without loss by forgetting the intermediate, inaccessible, behavior.

Or in PDDL the closest approximation arguably better resembles:

$$\hat{f}_d(t^{\in \mathbb{R} \setminus \mathbb{N}}) = \sum_{i=0}^{\lfloor t \rfloor} i; \tag{3.46}$$

either way, it seems unfortunate that inclusion fails:

$$\hat{f}_d \not\subset f_c. \tag{3.47}$$

Then let us close our long discussion of the relationship between our ITP and PDDL/VAL/POPF [36, 71, 119]. To a first-order approximation, and perhaps the fairest statement overall: we have reinvented the theoretical work of Fox, Long, Halsey, Coles, and Coles. Our differences are fairly slight, and by and large consist of 'merely' smoothing out any details less than widely supported in practice. The most concrete point is easily highlighted:

*We take durations (*i.e.*, of conditions, effects, and actions) as strictly positive.*

So, for future work, formal syntax best reflecting our semantics would differ notably. The 'why'—as *whys* are wont to be—is far from simple. We can, though, sharply bound the scope of the debate. All issues trace back to a single core semantic puzzle regarding time that must be answered by any formal theory. Namely:

The Type of Time: *Is time discrete or continuous?*

It should be clear that the answer has far-reaching consequence. That said, the choice is also arbitrary in that neither answer dominates the other. So we certainly cannot fairly say that ITP is hands-down better than temporal PDDL. At least, we cannot say such without first clarifying what *precisely* we mean by "temporal

PDDL". The true purpose of that line of work aims to reach mixed discrete-continuous planning: a promise delivered upon in the form of COLIN [37]. Naturally such purpose is perfectly reasonable; there are certainly compelling real-world applications of temporal planning at that level of expressiveness. For smaller/differing ambition it makes sense to use more specialized tools. In the end, such is the sense in which it is fair to say that ITP is contribution.

Then to put all specifics aside, in general, the points discussed here worth remembering are:

- We need to be clear about the nature of time and change.
- In making that choice, it is worth keeping in mind that just because change is discrete does not mean it need be instantaneous.
- Nor is it the case that just because we imagine reality to be continuous do we need to model it continuously.
- In particular the natural abstraction from continous to discrete is to take discrete change as durative.
- For short, in terms of graphs:
  - durations go to edges when edges model the bulk of change, and
  - durations go to vertices when vertices model the bulk of change.

Chapter 4

Analysis of Expressiveness of Temporal Planning Languages

In this chapter we aim to establish that the languages of Interleaved Temporal Planning and Conservative Temporal Planning well capture the theoretical capabilities of the state-of-the-art. What we actually show is only a piece of the puzzle [11, 24, 72, 74, 75, 78, 158, 186, 187]. Abstractly, we setup a large space of temporal planning languages possessing various syntactic capabilities . . . and then proceed to demonstrate that only one such feature 'really matters'.

Theorem 4.1 A language expresses required concurrency 'iff' causally compound actions are permitted.

Theorem 4.10 If all actions are causally primitive, then the language is isomorphic to Conservative Temporal Planning.

Theorem 4.12 'All' languages permitting causally compound actions compile into one another—arguably—well enough.

So Interleaved Temporal Planning can be taken as representative.

Motivation. Many different modeling languages have been proposed for planning with durative actions, and we are interested in their relative expressiveness. The language of TGP [188], for example, requires that an action's effects take place during its entire execution. In contrast PDDL demands that effects take place instantaneously at just the endpoints of actions; meanwhile conditions are permitted to constrain all of an action's execution (or just its endpoints). Many other sys-

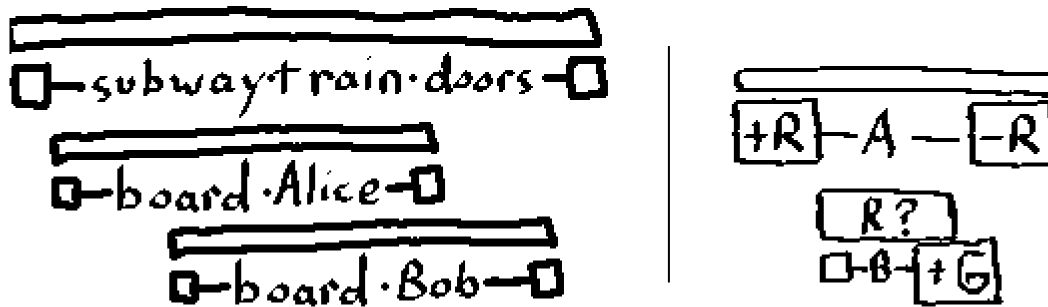

Figure 4.1: A natural pattern for required concurrency: action A provides temporary access to a resource R, which action B utilizes to achieve G.

tems (IxTeT, SAPA, TALplanner, ZENO, ...) permit varying degrees of access to subintervals of an action's execution. In some settings action definitions are even permitted to reach *beyond* their intervals of execution [12].

Here we examine a number of sublanguages of Interleaved Temporal Planning formed by permitting or forbidding access to various such subintervals; we also consider restricting what sorts of statements may be made over permitted subintervals. The goal is to characterize what makes temporal planning difficult relative to classical planning. To the extent that can be accomplished at the language level we shall have tools for more easily classifying whether a given temporal planner is going to be enjoying an unfair advantage relative to another, more temporally expressive, planner. (Or fail utterly should we instead pick a problem beyond its understanding.) The difficult part is that not all differences in syntax are created equal—some matter much more than others. So in other words, we wish to separate the temporal syntactic sugar from the *really* temporal feature(s).

The hypothesis is that required concurrency is the key semantic feature. To make this more concrete consider the special case depicted in Figure 4.1: exploit an only *temporarily* available resource (R) so as to achieve some desirable result

(G). Specifically note that thinking of action A as a primitive misses all solutions; the only lasting effect of action A is to delete the resource R. To model temporary availability of resources as state transitions, we need to think of them as consisting of at least two parts: a beginning and an end. However, adding detail to a model makes it larger—and so more difficult to solve—*i.e.*, doing so is precisely the opposite of abstraction [131]. So then, after breaking action definitions into (effectively) two pieces each, 'double' the number of decisions must be made in order to make plans. Figuratively: *Taking required concurrency as plausible detonates a combinatorial explosion.*

So: required concurrency matters. Does anything else?

Organization. Section 4.2: We completely characterize expressibility of required concurrency within the particular scope of inquiry setup. Figure 4.2 depicts the result, in words: Expressibility of compound actions characterizes required concurrency. The latter two theorems aim to support the notion that required concurrency is, in the first place, the right feature to be examining. Section 4.3: We elaborate upon why the guaranteed absence of required concurrency makes planning easier. Section 4.4: We argue that all ways of permitting required concurrency are equally hard. (So anything else that matters may be understood as significant through the lens of required concurrency.) Wrapping up the chapter is Section 4.5. First:

4.1 DEFINITIONS: REQUIRED CONCURRENCY, CAUSALLY COMPOUND, AND SUBLANGUAGES

There is a twist to the sense of “required” in “required concurrency”. Intuitively the meaning is just that all solutions schedule at least two actions concurrently: then a planner is ‘forced’ to generalize beyond considering mere sequences of actions. However, as we have already seen, *deadlines* contradict that intuition. Namely, Conservative Temporal Planning permits deadlines, yet, does not force generalizing beyond action-sequences, Theorem 3.17. So we need a better notion of “forced”. We appeal to causality:

Definition 4.1 (Required Concurrency)**.** A (solvable) problem **(causally) requires concurrency** if every solution is causally concurrent. A problem is **causally sequential** if every executable effect-schedule is causally sequential.

Definition 4.2 (Causally Sequential/Concurrent)**.** An effect-schedule is **causally sequential** when deordered-equivalent to a classically-sorted effect-schedule. Inversely, an effect-schedule is **causally concurrent** when not deordered-equivalent to a classically-sorted effect-schedule.

A **classically-sorted** effect-schedule always immediately dispatches every part of a compound action.[1] For notation, the effect-schedule $(a, t)_{[n]}$ is classically-sorted when: for every $i^{\in[n]}$ such that $a_i = \texttt{fin-}\alpha$ holds then both $a_{i-1} = \texttt{bgn-}\alpha$ and $a_{i-2} = \texttt{all-}\alpha$ hold.

So a classically-sorted effect-schedule is literally sequential, *i.e.*, more or less the same as an action-sequence; every way of reordering it, respecting mutual exclusions, is causally sequential. If every schedule can be had by such reordering,

[1]If all parts of an action appear contiguously in an effect-sequence or effect-schedule, then we could say the plan/schedule *compresses* the action [38].

then it would suffice to apply a Conservative Temporal Planner. Such simplification is of great note—so define:

Definition 4.3 (Temporal Expressiveness)**.** A language is **temporally expressive** when some permitted (solvable) problem (causally) requires concurrency. A language is **temporally simple** when every permitted problem is causally sequential.

The sacrifice here is the definition of causally sequential. The—much too ambitious—intuition is: Problems *somehow* reducing to Sequential Planning with negligible consequences are causally sequential. Even adequate formalization is of itself nontrivial [158]. Subsequently proving our intended theorems would be 'impossible'—*i.e.*, much the same as proving NP ≠ P.Then we limit our ambition to considering just the reduction by deordered-equivalence (*cf.*, Theorem 3.14, Page 128).

Intuitively speaking, a witness to required concurrency amounts to a single action whose parts cannot be brought together *via* deordering with respect to some larger plan. For that to happen it would seem that two of its effects need to be separated by a mutex effect of some other action. Actually though, the character of a witness to required concurrency can be somewhat more three dimensional. Suppose the (only) mutual exclusions between the effects of two actions α and β are such that the following partial-order characterizes a deordered-equivalence class.

$$\texttt{bgn-}\alpha < \texttt{fin-}\beta,$$

$$\texttt{bgn-}\beta < \texttt{fin-}\alpha.$$

In words: so both must start before either can finish, and all ways of doing so are causally equivalent. Note that we can bring together α without altering behavior: begin β, do all of α, finish β. Symmetrically we can bring together β: begin α, do

all of β, finish α. What is impossible is to do so for both at once; neither classically -sorted permutation is consistent with the given partial-order. That being said, it is still the case that the intuition more or less gets the formal definition right.

Definition 4.4 (Causally Compound/Primitive)**.** An action is **causally compound** if two of its nontrivial effects are temporally disjoint. Otherwise the action is **causally primitive**: all of its nontrivial effects temporally intersect.

For notation, define the **critical region** of an action as the intersection of the relative temporal extents of its nontrivial effects; so causally primitive actions are those possessing nonempty critical regions:[2]

$$\emptyset \neq \bigcap_{\mathit{eff}_{(\alpha,x)} \neq \{\{\} \mapsto \{\}\}} \left\{t - \mathrm{AST}_\alpha \mid \mathrm{AST}_{(\alpha,x)} \leq t < \mathrm{AFT}_{(\alpha,x)}\right\}. \tag{4.1}$$

Less generally and inversely, an action is causally compound precisely when the start-part and end-part are nontrivial and nonintersecting:

$$\mathit{eff}_{\texttt{bgn-}\alpha} \neq \{\{\} \mapsto \{\}\}, \tag{4.2}$$

$$\mathit{eff}_{\texttt{fin-}\alpha} \neq \{\{\} \mapsto \{\}\}, \qquad \text{and} \tag{4.3}$$

$$\mathit{dur}_{\texttt{all-}\alpha} \geq \mathit{dur}_{\texttt{bgn-}\alpha} + \mathit{dur}_{\texttt{fin-}\alpha}. \tag{4.4}$$

Details: A dispatch/part/effect is **trivial** if it is executable in every state and writes to no fluent. (The canonical trivial effect is denoted by $\{\{\} \mapsto \{\}\}$.) The **(absolute) temporal extent** of a dispatch at time s of duration d is all those times at

[2] A causally primitive action is much like a *unique main subaction* [207].

which a mutually exclusive dispatch may not start: the interval $[s, s + d)$.[3] A **relative temporal extent** takes the start-time of the dispatch of the action as the origin of time. Durations are constants; so, an end-part of duration d belonging to an action of duration D always has relative temporal extent $[D - d, D)$.

Remark 4.1. More generally we would say that an action is effectively primitive when, for all plans of interest (executable/solution/optimal/. . .) including it, if one of its parts is causally ordered before some x, then none of its parts are causally ordered after x, and *vice versa* (if one must be after, then all could be after). An upshot is that, if that condition is met by every action, then, by induction, every plan of interest may be classically-sorted—it would suffice, for planning, to abstract to treating every action as primitive. In particular note that, by Corollary 3.15 (every executable plan is deordered-equivalent to a time-sorted reordering), checking that every nontrivial part temporally intersects gives the desired property.

Wrapping up the definitions, consider the generic concept of imposing syntax constraints. For examples, we might consider permitting or forbidding any of: *conditional effects*, *deadlines*, *numeric fluents*, *negative preconditions*, *action costs*, *variable durations*, *continuous but nonlinear change*, *disjunction*, *quantifiers*, *parameters*, and so forth [11, 24, 72, 74, 75, 78, 133, 158, 186, 187]. Given such background—let us forbid all but the essentials.

Specifically consider the $64 = (2 \times 2)(2 \times 2)(2 \times 2)$ languages obtained by independently permitting/forbidding equality/assignment statements in each of the three parts of an action. A **degenerate** planning language forbids all change (*i.e.*, no assignment statements). Any self-respecting planner solves such 'problems'

[3] Less naturally, define temporal extents by when mutually exclusive locks cannot be released (rather than cannot be acquired), thereby obtaining left-half-open intervals instead.

without search. So rule out the degenerate cases, leaving $56 = 64 - 8$ languages of interest:

Definition 4.5 (Syntactic Restrictions)**.** Our top-level language for now will be syntactically restricted to $\mathtt{L}(\mathtt{pre}, \mathtt{eff};\ \mathtt{pre}, \mathtt{eff};\ \mathtt{pre}, \mathtt{eff})$, meaning: every part may have any effect in the relatively limited set $(\mathtt{pre} \mid \mathtt{eff})^*$ consisting of merely equality tests against, and assignments from, constants.[4] More generally denote by $\mathtt{L}(\gamma;\ \alpha;\ \omega)$ the sublanguage of Interleaved Temporal Planning **restricting** the form of action definitions by stipulating that each satisfy: the effect of the all-part is in the set γ^*, the effect of the start-part is in the set α^*, and the effect of the end-part is in the set ω^*.

Definition 4.6 ((Psuedo-)Syntax)**.** A **condition** is an 'effect' that writes nowhere (a **proper** effect writes somewhere); a **primitive condition** verifies that a given fluent presently has the desired, constant, value (by failing to be executable if otherwise). Say "$f = v$" is shorthand for $\{\{f \mapsto v\} \mapsto \{\}\}$, and let $\mathtt{pre}$ stand for all such primitive conditions.

A **primitive assignment** unconditionally writes a given constant value to a given fluent. Abbreviate $\left\{\{f \mapsto u\} \mapsto \{f \mapsto v\} \mid u \in \mathit{Values}_f\right\}$ as "$f := v$", and let $\mathtt{eff}$ be the entire set of such primitive assignments.

A **primitive transition** is the result of stating both a primitive precondition and primitive assignment upon the same fluent: for convenience, let "$f = u := v$", or "$f = u, f := v$", stand for $\{\{f \mapsto u\} \mapsto \{f \mapsto v\}\}$.

Say concatenation of representations of effects is defined by parallel composition of their interpretations. Dropping the distinction between syntax/semantics, for

[4] Restricting to constant values is equivalent to insisting upon STRIPS-style.

x and y two effects, so long as $Writes_x \cap Writes_y = \emptyset$, define concatenation (x, y) as parallel-composition '$(x, y) \coloneqq (x \mid y)$'; with $Writes_{x,y} \coloneqq Writes_x \cup Writes_y$ and $Depends_{x,y} \coloneqq Depends_x \cup Depends_y$ define parallel-composition by:

$$(x \mid y) \coloneqq \left\{ A^{\in States(Depends_{x,y})} \mapsto A' \mid A' = (x(A\restriction_{Depends_x}) \cup y(A\restriction_{Depends_y}))^{\in States(Writes_{x,y})} \right\}.$$

Applicability to the State of the Art. What matters most is for the space of languages setup to capture 'common denominators' of current practice. That seems true enough. For examples (the semantics of several systems are ambiguous, hence repeats): The actions permitted by the top-level, $\mathtt{L}(\mathtt{pre}, \mathtt{eff};\ \mathtt{pre}, \mathtt{eff};\ \mathtt{pre}, \mathtt{eff})$, are particularly compatible with one 'obvious' generalization of SAS to time [9, 64, 103]. The sublanguage $\mathtt{L}(\mathtt{pre};\ \mathtt{pre}, \mathtt{eff};\ \mathtt{pre}, \mathtt{eff})$ is a slightly more precise caricature of the sorts of actions the discrete temporal fragment of PDDL considers, because PDDL only permits changes at endpoints [71]. So that sublanguage captures a fragment of the capabilities of many temporal planners [54, 64, 85, 93, 137, 142, 208]. The sublanguage $\mathtt{L}(\mathtt{pre}, \mathtt{eff};\ \emptyset;\ \emptyset)$ describes TGP reasonably well, as TGP demands that all statements be with respect to the whole action [188]. Then more generally that sublanguage serves to describe reasonably well 'any' remaining temporal planner [6, 54, 57, 64, 87, 105, 115, 135, 202].

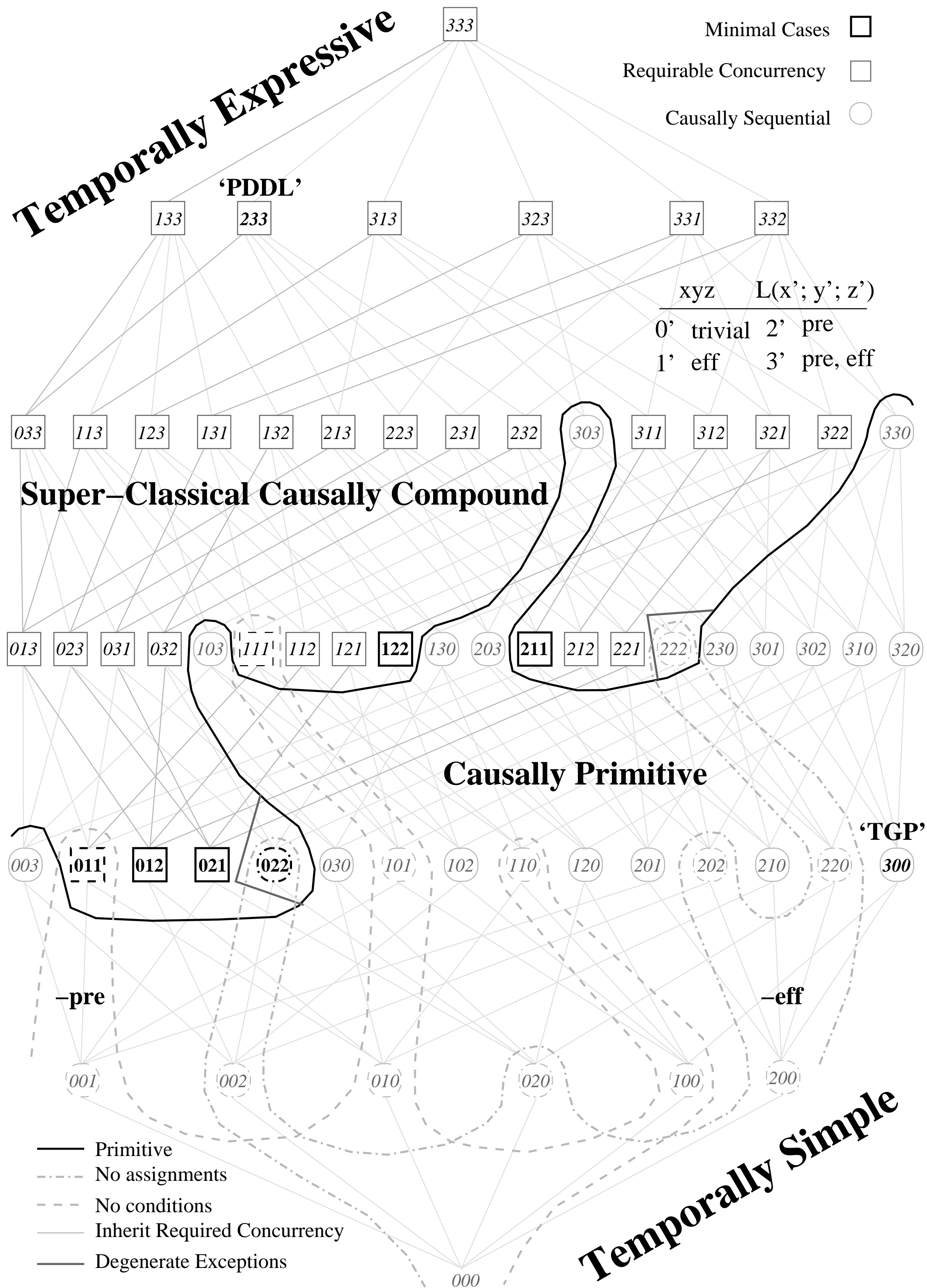

Figure 4.2: The taxonomy of temporal sublanguages, see Section 4.2.1.

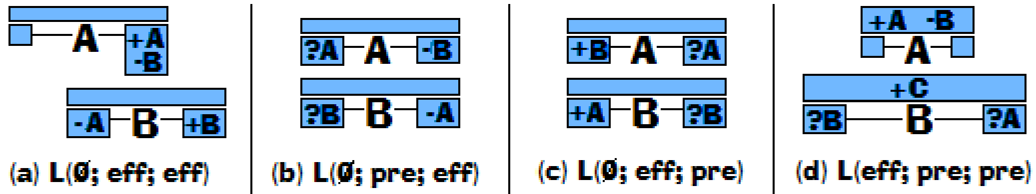

Figure 4.3: Minimal nondegenerate languages permitting causally compound actions are temporally expressive. See Propositions 4.6 through 4.9.

4.2 CAUSALLY COMPOUND ACTIONS CHARACTERIZE REQUIRED CONCURRENCY

In short: Figures 4.4 and 4.3 prove Figure 4.2.

Theorem 4.1. *Planning languages forbidding causally compound actions are temporally simple: if restricted to sublanguages of Interleaved Temporal Planning.*

Planning languages permitting causally compound actions are temporally expressive: if restricted to specifically those sublanguages named in Figure 4.2.

Proof. The proof is by reduction to various lemmas and propositions, as follows.

(First Claim) The definition of a temporally simple language is that every permitted problem satisfies: Every executable plan is causally sequential. Lemma 4.3 establishes that absence of causally compound actions guarantees such, ergo proves the claim.

(Second Claim) By definition, it suffices to demonstrate just one permitted solvable problem requiring concurrency per language. As superlanguages permit supersets of problems, it suffices to consider just the minimal cases. Concerning the Figure, removing all those languages forbidding causally compound actions leaves, by inspection (or by Lemma 4.4), four minimal elements. Propositions 4.6 through 4.9 prove that the four are temporally expressive, and hence further prove the entire claim. □

4.2.1 Brief Discussion of Figure 4.2

The figure—a depiction of the lattice of syntactic restrictions—has a lot of detail so as to assist in following the machinations of the proofs. That is, it is meant as a visual aid to keep with us while carrying them out. It is also useful as a device for doublechecking.

To really walk through its meaning is just to walk through the proofs. Then up -front let us clarify just a few bits, primarily: the mapping between the notation. (Writing out the verbose notation of the proofs would have the figure span multiple pages, defeating the point.) The vertex labels encode the syntax restrictions as triplets in base 4. For example, "123" stands for $\mathtt{L}(\mathtt{eff};\ \mathtt{pre};\ \mathtt{pre}, \mathtt{eff})$.

Note that there are two different ways in which elements of the lattice can degenerate. One is to forbid effects altogether. The other is to forbid conditions altogether. Those two sublattices extend past the line that is the technical definition of *causally compound*.

Hence it is incorrect to say that *causally compound actions* characterize *required concurrency*. For abstractness: there is a syntactic line, a semantic line, and the two almost, but do not quite, coincide. The figure serves to visually delineate the caveats.

The formal arguments serve to much more precisely define what those caveats are. Their precise identity is meaningless within the lattice depicted. However, the scope of the figure is certainly not the scope of interest. It is merely a surrogate for our interest: one small enough to completely investigate.

So precise understanding is worthwhile. In other words, the proofs matter because the particular lattice is only a tiny piece of the full puzzle. The *full* puzzle includes relational representation, quantifiers, object-valued fluents, conditional ef-

fects, arithmetic, calculus, trajectory constraints, preferences, uncertainty, sensing, execution, learning, and so forth onwards to infinity.

4.2.2 First Claim: Necessity

It is interesting to first distinguish a weaker necessary condition (necessary for expressiveness, sufficient for simplicity). The natural necessary condition amounts to checking whether any compound action actually *is* compound; if an action has only one nontrivial part there is no point in calling it compound. In other words $\mathtt{L}(\mathtt{pre}, \mathtt{eff};\ \emptyset;\ \emptyset)$ merely adds excess baggage to Conservative Temporal Planning:

Proposition 4.2. *The language* $\mathtt{L}(\mathtt{pre}, \mathtt{eff};\ \emptyset;\ \emptyset)$ *is temporally simple. It is moreover the natural embedding into Interleaved Temporal Planning of Conservative Temporal Planning restricted to* STRIPS*-style.*

Proof. We may assume by deordered-equivalence that all parts of every compound are contiguous in every executable effect-schedule; *i.e.*, the trivial parts are mutex with nothing and so may be freely moved into the desired positions by Theorem 3.21. Therefore every executable plan is causally sequential (*i.e.*, deordered-equivalent to a classically-sorted effect-schedule). Which suffices. For the moreover just transliterate between actions and all-parts; the omitted details are easily addressed. □

This is interesting insofar as we may consider substituting a far more sophisticated analysis of unreachability as justification for the desired sorting. (As alluded to in Remark 4.1.) For present purpose though it suffices to consider just a guarantee of critical regions.

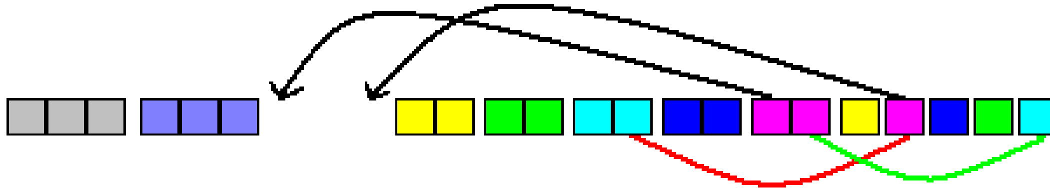

Figure 4.4: With actions' effects all concurrent, we may reduce to action-sequences. The earlier hypothetical mutex, which would be problematic, is impossible given the assumption that the later effects have soonest critical region.

Consider any two mutex effects. They must occur in some order, implying their critical regions do as well. Those critical regions are the common intersections of all the 'sibling' effects; all other effects of the respective actions must already conform to the same order. So no two mutex effects ever need cross paths in order to bring together all parts of every action. See Figure 4.4; precisely:

Lemma 4.3. *If all actions possess critical regions then every plan is causally sequential.*

Proof. So we aim to classically-sort any given plan without breaking deordered-equivalence.

1. Pick any critical point within each critical region.[5]

2. Sort the plan by iteratively moving to the front of the unsorted suffix the dispatches of all parts of that action with earliest critical point, among those remaining.

3. Break ties arbitrarily.

Then the increasingly long prefix sorted by critical-points is more specifically classically-sorted.

[5] If when effects occur is not constant, then some modifications are called for.

Therefore, if the sorting completes without breaking deordered-equivalence, then we are done. For a contradiction, suppose not.

In order to break deordered-equivalence, moving some dispatch to the front of the unsorted suffix must at some interim point swap positions with a mutually exclusive dispatch (by definition). Consider the first such move, for notation:

- the still deordered-equivalent plan is sorted up to just before index i,
- dispatch (b, t) at some index k of duration e has an earliest critical point y,
- dispatch (a, s) at some index $i \leq j < k$ of duration d is mutex with b, and
- *(for emphasis:)* the critical point of a, say x, is no earlier: $y \leq x$.

Mutually exclusive dispatches occur in time in the order dispatched (*i.e.*, by the locking protocol; *cf.* Proposition 3.12, which remains true in ITP). So $s + d \leq t$, because $j < k$. Temporal extents are right-half-open, and contain the critical points: $s \leq x < s + d$ and $t \leq y < t + e$. Then the critical point of the dispatch of a is strictly earlier than that of b: $x < s + d \leq t \leq y$. Which contradicts the assumption of a counterexample, *i.e.*, the dispatch of b was chosen for the virtue of having earliest critical point: but $x < y$ contradicts $y \leq x$. □

4.2.3 Second Claim: Sufficiency

It is clear that the sufficiency direction of the theorem extends far beyond the stated scope; if for no other reason, then simply because expressiveness is a 'poisonous' property. There are more interesting reasons to consider. It is almost, but not quite, true that sufficiency holds 'in general'. In particular, as long as the case analysis of the proof remains exhaustive, then sufficiency continues to hold: the propositions proving each case are unshakeable.

So in other words the only way that generalization could break sufficiency is to introduce new minimal cases. Moreover such must turn out to be, surprisingly (as we shall see), temporally simple despite permitting causally compound actions. Neither introducing new minimal cases nor having such be temporally simple is an easy feat. Consider:

Any part executable in every state and altering no fluent is trivial by definition. Then in general a part is nontrivial for one of two reasons: not always executable, or sometimes altering fluents. So in the 'simplest' cases a nontrivial part permits either primitive conditions, or primitive assignments. Hence, as a witness to a causally compound action needs two nontrivial parts, there are just four minimal cases even 'in general'. Namely the possibilities are: two disjoint primitive assignments, a primitive condition preceding a primitive assignment, a primitive assignment preceding a primitive condition, and two disjoint primitive conditions. Or, within the narrower vision of the Figure, the intuitive minimal cases are: $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{eff})$, $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff})$, $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre})$, and $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{pre})$.

Then we have our exception proving the rule: $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{pre})$ permits causally compound actions but is nonetheless temporally simple. This is, of course, only because it permits no proper effects whatsoever; all solvable problems it permits are solvable by the empty plan. Such is not the only exception; one way of generalizing to *continuous change* can be taken as admitting appropriate counterexamples. Still, at the high-level, the accurate view is that the theorem is true in general.

As far as formality goes: So we come to exhaustiveness of the case analysis.

Lemma 4.4. *The minimal nondegenerate sublanguages of the top-level permitting causally compound actions are:*

$$\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{eff}),$$
$$\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff}),$$
$$\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre}), \qquad \textit{and}$$
$$\mathtt{L}(\mathtt{eff};\ \mathtt{pre};\ \mathtt{pre}).$$

Proof. A witness of a causally compound action needs two nontrivial parts. So there are four minimal languages permitting causally compound actions. The first three are $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{eff})$, $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff})$, and $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre})$. By inspection, each is nondegenerate. So only the final putative witness remains: $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{pre})$, which is degenerate. As that is the only additional constraint it suffices to account for the degeneracy. Then consider replacing the degenerate 'witness' by its nondegenerate superlanguages: permit assignments in one of the three parts. We break the three specific possibilities up into two kinds.

The first kind permits an assignment temporally disjoint from one of the given pair of disjoint conditions. So, we could instead regard the case as an instance of an assignment before a condition or *vice versa*. Either way the form is already considered: such superlanguages are non-minimal. Specifically $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{pre},\mathtt{eff})$ and $\mathtt{L}(\emptyset;\ \mathtt{pre},\mathtt{eff};\ \mathtt{pre})$ are superlanguages of the witnesses $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff})$ and $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre})$.

The second kind is the negation of the first: no assignment may be temporally disjoint from any condition. (So even 'in general', proper effects are permitted only

over whole actions, more or less.) Specifically the language $\mathtt{L}(\mathtt{eff};\ \mathtt{pre};\ \mathtt{pre})$ fills the shoes left vacant by the degeneracy of $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{pre})$. □

So with that said, it suffices to prove the existence of a single problem requiring concurrency per minimal language. While all of our examples are small enough to just brute-force check 'every' executable plan, which perhaps yields simpler proofs, it is more interesting to consider a proof method amenable to implementation. Let us illustrate the details. Consider Figure 4.3 and Figure 4.1. Actions are drawn by their 3 parts, in each is shorthand, geared towards boolean fluents, for the conditions/assignments meant. In particular "+P" denotes $\mathrm{P} := \mathtt{True}$, "-P" denotes $\mathrm{P} := \mathtt{False}$, and "?P" denotes $\mathrm{P} = \mathtt{True}$. Each depiction of this kind is supposed to concern a 'uniquely' solvable problem, with the picture indicating the equivalent nature of all minimal solutions. So all of the following details concerning specifically the abstract problem depicted in Figure 4.1 are supposed to be 'obvious' and/or largely irrelevant; for a change let us drop all the way down to a purely functional notation:

$$
\begin{aligned}
\mathit{FluentDefs} &= \{G \mapsto \mathbb{B}; R \mapsto \mathbb{B}\}.\\
\mathit{ActionDefs}(\mathtt{all}\text{-}A) &= \Big(\Big\{\{\}^{\in States(\emptyset)} \mapsto \{\}^{\in States(\emptyset)}\Big\}, 4\Big).\\
\mathit{ActionDefs}(\mathtt{bgn}\text{-}A) &= \Big(\Big\{\ \cdot^{\in States(\{R\})} \mapsto \{R \mapsto \mathtt{True}\}^{\in States(\{R\})}\Big\}, 1\Big).\\
\mathit{ActionDefs}(\mathtt{fin}\text{-}A) &= \Big(\Big\{\ \cdot^{\in States(\{R\})} \mapsto \{R \mapsto \mathtt{False}\}^{\in States(\{R\})}\Big\}, 1\Big).\\
\mathit{ActionDefs}(\mathtt{all}\text{-}B) &= \Big(\Big\{\{R \mapsto \mathtt{True}\}^{\in States(\{R\})} \mapsto \{\}^{\in States(\emptyset)}\Big\}, 2\Big).\\
\mathit{ActionDefs}(\mathtt{bgn}\text{-}B) &= \Big(\Big\{\{\}^{\in States(\emptyset)} \mapsto \{\}^{\in States(\emptyset)}\Big\}, 1\Big).\\
\mathit{ActionDefs}(\mathtt{fin}\text{-}B) &= \Big(\Big\{\ \cdot^{\in States(\{G\})} \mapsto \{G \mapsto \mathtt{True}\}^{\in States(\{G\})}\Big\}, 1\Big).\\
\mathit{Initial} &= \Big(\Big\{f^{\in Fluents} \mapsto \mathtt{False}\Big\}, \{f \mapsto (t_{-\infty}, 0, \mathtt{False})\}, \{\alpha \mapsto \{\}\}\Big).
\end{aligned}
$$

$$Goal = \Big\{(S, V, D)^{\in Balances} \mapsto (S(G) = \texttt{True} \text{ and } D = \{\alpha \mapsto \{\}\})\Big\}.$$

Proposition 4.5. *Our analogue for* PDDL *is temporally expressive.*

Proof. The problem depicted in Figure 4.1 satisfies the syntactic restrictions defining $\texttt{L}(\texttt{pre};\ \texttt{pre}, \texttt{eff};\ \texttt{pre}, \texttt{eff})$: it suffices to demonstrate that specifically this problem requires concurrency. Technically solvability should be demonstrated. That $((\texttt{all-}A, 0), (\texttt{bgn-}A, 0), (\texttt{all-}B, 1), (\texttt{bgn-}B, 1), (\texttt{fin-}B, 3), (\texttt{fin-}A, 4))$ is a solution is easily verified.

The claim which matters is: No solution is causally sequential. This is easily seen. Treating A sequentially renders it useless, and then B cannot ever begin, so the goal can never be achieved. Applying planning graphs (or similar) in the following fashion is a powerful technique for having machines arrive at the same conclusion as a special case.

Setup. By Theorem 3.21, deordered-equivalence implies solution-equivalence. So to verify the absence of causally sequential solutions it suffices to consider merely their canonical representatives: the classically-sorted effect-schedules. That is, it suffices to show that no classically-sorted effect-schedule could possibly be a solution. Consider just failure of the classical constraints, as follows. Compute the net effects of each action upon states (by sequential composition of the associated state transition functions), and treat those as primitives of a projection/abstraction to Sequential Planning. Note that if the projection is unsolvable, then the original problem cannot be solved by classically-sorted effect-schedules. For completeness: Let $\mathcal{P}' = (\mathcal{F} = \mathit{FluentDefs}, \mathcal{A}, \mathcal{I} = (R{:=}\texttt{False}, G{:=}\texttt{False}), \mathcal{G} = (G{=}\texttt{True}))$ denote the projection to Sequential Planning. Concerning A, its net effect is just to delete

R, *i.e.*, $\mathcal{A}(A) = (R{:=}\texttt{False})$:

$$S'_A \coloneqq S'_{\texttt{fin-}A} \circ S'_{\texttt{bgn-}A} \circ S'_{\texttt{all-}A} = \{S \mapsto (S \oplus \{R \mapsto \texttt{False}\})\}.$$

Concerning B, in net it needs R and gives G, *i.e.*, $\mathcal{A}(B) = (R{=}\texttt{True}, G{:=}\texttt{True})$:

$$S'_B \coloneqq S'_{\texttt{fin-}B} \circ S'_{\texttt{bgn-}B} \circ S'_{\texttt{all-}B} = \{S \mapsto (S \oplus \{G \mapsto \texttt{True}\}) \mid S(R) = \texttt{True}\}.$$

Relaxed Reachability. Then consider the following analysis of (un)reachability in the projected problem. No reachable effect establishes $R = \texttt{True}$, indeed, no effect whatsoever includes $R \coloneqq \texttt{True}$. So R is monotonically decreasing in truth-value. As it starts false, it is false in every reachable state of $\mathcal{P}'$. Therefore B always fails to be executable as its precondition is never satisfied. Hence no reachable effect establishes $G = \texttt{True}$, as the effect of B is the only effect to include $G \coloneqq \texttt{True}$. So G is monotonically decreasing in truth-value. As G, which is the goal, starts false, it is false in every reachable state of $\mathcal{P}'$. Therefore the goal is unachievable by executable plans: unsolvability of the projection to Sequential Planning is verified.

□

Then for the following let us treat the details lightly; that is, for the details adapt the above proof of Proposition 4.5.

Proposition 4.6. *The minimal language witnessing permitted causally compound actions between pre/post-assignments is temporally expressive:*

$\texttt{L}(\emptyset;\ \texttt{eff};\ \texttt{eff})$ *permits a problem requiring concurrency.*

Proof. The problem depicted in Figure 4.3(**a**) witnesses: achieving “?A, ?B” from “-A, -B” requires action B to wrap the end-part of action A. □

Proposition 4.7. *The minimal language witnessing permitted causally compound actions between preconditions and postassignments is temporally expressive:*

$\texttt{L}(\emptyset;\ \texttt{pre};\ \texttt{eff})$ *permits a problem requiring concurrency.*

Proof. The problem depicted in Figure 4.3(**b**) witnesses: achieving A = `False` and B = `False` from "+A, +B" requires both actions A and B to begin before either may end. □

Proposition 4.8. *The minimal language witnessing permitted causally compound actions between preassignments and postconditions is temporally expressive:*

$\texttt{L}(\emptyset;\ \texttt{eff};\ \texttt{pre})$ *permits a problem requiring concurrency.*

Proof. The problem depicted in Figure 4.3(**c**) witnesses: achieving "?A, ?B" from "-A, -B" requires both actions A and B to begin before either may end. □

Proposition 4.9. *The minimal language witnessing permitted causally compound actions between pre/post-conditions is temporally expressive:*

$\texttt{L}(\texttt{eff};\ \texttt{pre};\ \texttt{pre})$ *permits a problem requiring concurrency.*

Proof. The problem depicted in Figure 4.3(**d**) witnesses: achieving A = `True`, B = `False`, and C = `True` from "-A, +B, -C" requires action B to contain action A. □

4.3 SEQUENTIAL PLANNING 'SUPPORTS' (ALL FORMS OF) OPTIONAL CONCURRENCY: SLACKLESS CLASSICALLY-SORTED EFFECT-SCHEDULES ARE ACTION-SEQUENCES

Conservative Temporal Planning is a relatively mild generalization of Sequential Planning: adapting some classical planner to work reasonably well is more or less straightforward. Namely it suffices to integrate with First-Fit Scheduling. The same (for an insignificantly-relaxed notion of "First-Fit") is true in general of any temporally simple language. We prove so by generalizing Theorem 3.4 (left-shifting dominates).

Two effect-schedules are slack-equivalent when (i) the underlying effect-sequences are identical, and (ii) both are actual. Two effect-schedules are deorder-equivalent when their deorders are identical (*i.e.*, $\prec_{\text{deorder-}X'} = \prec_{\text{deorder-}Y}$). Two effect -schedules X and Y are **deorder-slack-equivalent** when there exists an X' slack-equivalent to X such that X' is deorder-equivalent to Y. Alternatively, define deorder -slack-equivalent in the other order, *i.e.*, by existence of an X'' deorder-equivalent to X such that X'' is slack-equivalent to Y. That the relation *is* an equivalence-relation, and that the two definitions really are the same, is straightforward relative to Chapter 3. It may be helpful to visualize a third definition: verify that (1) their corresponding STNs are isomorphic as weighted graphs, and (2) both correspond to solutions thereof.

Theorem 4.10. *Assume any effect-schedule solving a problem of a temporally simple language. There exists a deorder-slack-equivalent solution that is more specifically: classically-sorted, slackless, unique up to tie-breaking, and* greedily *computable.*

Corollary 4.11. *Considering every action-sequence is a complete approach to any temporally simple language.*

Proof. Assume an effect-schedule X solving some causally sequential problem. By previous definitions and theorems: There is a (weakly) better solution Y that is slackless, classically-sorted, and unique up to tie-breaking. Then here we additionally show greedy computability and the corollary.

Before that though let us elaborate on existence of the better solution Y. By the definition of temporally simple, the effect-schedule X is deorder-equivalent to a classically-sorted (*i.e.*, all action parts contiguous) X'. Any such deorder-equivalent X' is moreover a solution, by the following.

- Theorem 3.21, specifically: deorder-equivalence implies behavior-equivalence.
- Proposition 3.10, specifically: behavior-equivalence implies result-equivalence.
- Definition 2.14, specifically: result-equivalence implies solution-equivalence.

By Theorem 3.18, *i.e.*, that slackless effect-schedules dominate their slack-equivalence classes, there exists a (weakly) better (slack-equivalent) rescheduling Y of the reordering X' of the schedule X. From the proof of that theorem (more specifically from Lemma 3.20), the choice of a slackless rescheduling is unique up to the choice of underlying effect-sequence. So tie-break the choice of the classically-sorted reordering X' arbitrarily. Therefore the schedule Y is deorder-slack-equivalent to the assumed solution X and is moreover: classically-sorted, slackless, and unique up to tie-breaking.

For the corollary: classically-sorted effect-sequences are in trivially-computable bijection with action-sequences. Then the corollary follows by the already established uniqueness of slackless schedulings.

For *greed*, argue by induction on prefixes of the action-sequence induced from the effect-sequence $(a)_{[3n]}$ underlying $X = (a, s)_{[3n]}$. First note that said inducing may itself be greedily computed: discard elements of X until arriving at the next dispatch of an all-part. The induction hypotheses are trivially satisfied by the empty prefix. For the inductive case then: let $Z' = Z_{k+1} = (Z_k, \alpha_{k+1})$ be some prefix with smaller prefix $Z_k = (\alpha)_{[k]}$. Recall the notion of simply expanding action-schedules: replace each action with all of its parts, in place. Specifically let $\hat{Z}_{k+1} = (b)_{[3(k+1)]} = (\texttt{all-}\alpha, \texttt{bgn-}\alpha, \texttt{fin-}\alpha)_{[k+1]}$ and $\hat{Z}_k = (b)_{[3k]} = (\texttt{all-}\alpha, \texttt{bgn-}\alpha, \texttt{fin-}\alpha)_{[k]}$ denote the classically-sorted effect-sequences that are the simple expansions of Z_{k+1} and Z_k.

(IH-greed-(k):) By induction the optimal solution $Y_k = (b, t)_{[3k]}$ to the simple temporal network $\text{STN}(\hat{Z}_k)$ corresponding to the simple expansion $\hat{Z}_k$ of the smaller prefix Z_k may be greedily computed. Let $Y' = Y_{k+1} = (b, t)_{[3(k+1)]}$ denote the optimal solution to the simple temporal network $\text{STN}(\hat{Z}_{k+1})$ corresponding to the simple expansion $\hat{Z}_{k+1}$ of the larger prefix Z_{k+1}. *(IH-greed-($k+1$):)* It suffices to prove that *(greed)* holds, meaning (1) Y' merely extends the smaller schedule by (2) assigning dispatch-times to the parts $(b_{3k+1},\ b_{3k+2},\ b_{3k+3})$ of just α_{k+1} in constant-bounded computation.

There are only two kinds of edges in the corresponding STNs: *precedence* and *duration*. These are themselves defined by induction, in particular, $\text{STN}(\hat{Z}_k)$ is a sub-STN of $\text{STN}(\hat{Z}_{k+1})$. Then consider that the precedence edges are all directed forwards. The duration edges, some of which travel backwards, travel only between parts of the same action. So all paths leaving $\text{STN}(\hat{Z}_k)$ in $\text{STN}(\hat{Z}_{k+1})$ never return

(because $\hat{Z}_k$ is classically-sorted). In particular the critical paths to the vertices in STN($\hat{Z}_k$) are the same with respect to either network. Then the prefix of the larger schedule is indeed identical to the smaller schedule: (1) $Y_k = Y' \upharpoonright_{[3k]}$.

We need only remember all of the weights of the prefixes of all critical paths crossing any given cut in order to later compute (without reexamining the graph prior to the cut) the weights of the longer ones. So consider again the cut separating STN($\hat{Z}_k$) and STN($\hat{Z}_{k+1}$). Again, all prefixes of critical paths cross the cut only once, meaning that: all edges that ever will be critical already are. While the number of crossing critical edges is not itself precisely constant, because read-locks are shareable, the number of weights we actually need to remember is. Specifically it suffices to remember just the current read-time and write-time of every fluent (in the form of the weights up to the vertices standing for the finish-times of the last effect to write to any given fluent and that yet later effect that just so happens to read last in the slackless schedule Y of Z_k).

That does not work in general when reasoning about STNs. We may get away with such here because we have already established (1): meaning we already know which of the read effects will be ending last as we extend the STN. In turn (1) holds only because of the special assumption that we are building STNs corresponding to just classically-sorted effect-sequences.

Then let us extend the induction hypothesis. For notation, let $W = (r, w)_{Fluents}$ denote the weights of the lightest weight vertices of STN($\hat{Z}_k$) that read from (r_f) and write to (w_f) each fluent f. *(IH-cut-(k):)* By induction, it suffices to remember at most two lightest weights per fluent ($W = (r, w)_{Fluents}$) in order to compute the weights of every critical path ever leaving STN($\hat{Z}_k$), and moreover the collection may be had in amortized constant runtime. Firstly: (2) the slackless dispatch-times

of the parts of α_{k+1} may be had in constant-bounded computation by referring to W and the duration constraints of those parts. So we have the first induction hypothesis fulfilled *(IH-greed-*$(k+1)$*)*.

The computation is polynomial in the number of fluents accessed by the parts: poly-time in a constant is still a constant. Then likewise and secondly, even if the parts of α_{k+1} were to end up writing to and reading from every fluent last: updating W to $W' = (r', w')_{Fluents}$ is still a constant-bounded amount of work. *(IH-cut-*$(k+1)$*):)* So it still suffices to remember just the two lightest weights per fluent in order to get away with forgetting the internals of STN($\hat{Z}_{k+1}$); moreover said memory (W') is had in constant space and constant amortized runtime. ◻

4.4 TEMPORALLY EXPRESSIVE LANGUAGES ARE EQUIVALENT UNDER COMPILATION

In general there are certainly more than two degrees of temporal expressiveness. For example, permitting continuous effects certainly complicates matters [37, 137, 154, 169]. Even restricted all the way down to the scope of Figure 4.2 there are still some interesting distinctions that could be made. For example, reasoning about preconditions is easier than reasoning about conditional effects in general. Then in some sense `L(eff; pre; pre)` is easier than `L(pre; eff; eff)`. In a different sense the two are equivalent. Namely both may be compiled into each other. The debate then is whether the consequences of such compilation are significant.

We examine the issue, with emphasis on the conclusion that our restricted scope contains only two degrees of difficulty. The notion is that whether compilation is a valid excuse for denying syntax depends upon the cleverness of state-of-the-art heuristic techniques. (More generally the dependency is upon the relative sophistication of any sort of inference in service of planning, not merely heuristics.) With respect to PDDL the matter is especially relevant [71, 72, 186]. We more or less side with Halsey, Long, and Fox: Compilation, in this case, resembles harmless.

The reason is that the computationally related criticisms of Smith are, to a point, nearly addressed by the state-of-the-art. These criticisms are significant practical concerns: not to be taken lightly. Hence proofs. Naturally the proofs only go so far; in the end the conclusion is a compromise. Namely:

- The only recently popular paradigm for planning implements inference (almost) powerful enough to see through the particular compilations.

- However, it is at best unclear whether more traditional—also more philosophically sophisticated—paradigms are able to see as clearly.

For example we strongly suspect that, at present, Smith's technical concerns hold rather strongly in the context of temporal generalizations of Partial-Order Planning [208]. Then a promising direction for future work (in any paradigm) is to extend the degree to which the open problem is closed; the issues are significant, and, attackable (in contrast to, say, $NP \neq P$). Let us clarify the motivations.

Make Forced 'Choices' Obvious. At its core, the technical flaw of compilations between declarative languages is that they introduce apparent choices that are ultimately forced—perhaps non-obviously. If the forcing is too subtle, it might only be discovered by trial-and-error: search. It is not unheard of to implement techniques such as Explanation Based Learning/Generalization and Dependency-Directed Backtracking in the context of planning [123, 209]. But it *is* unusual. Without such techniques: The inability to see the forcing, but for exhausting the wrong values of the apparent choices, will happen over and over again throughout the space of decisions. At *best*, that is a constant-factor slowdown. Even 'small' constants are already bleak: a slowdown by a factor of 10 is hardly palatable. At worst, and not atypically, it will be reasonably accurate to estimate the run-time as dependent on the number of apparent, rather than real, choices: the slow-down will be exponential. However, careful compilation, particularly with knowledge of the internals of the solver to be applied, may avoid the flaw entirely. The essence is to produce poly-time machine-verifiable proofs of the correctness of the compilation. In particular the requirement is to fit the proof within those poly-time inference

schemes already employed (presumably for more well-motivated purposes than addressing compiled problems).

Obvious to an Automated Planner Means Planning Graphs. A fundamental of automated planning is to implement poly-time procedures for relaxed analysis of reachability. The relaxations are all quite particular: If some proposition can ever be made true, then it remains true forevermore. The details differ in what propositions are investigated. For example, in the Context Enhanced Additive Heuristic the propositions in question are compound statements resembling "An inadmissible estimate of the cost of achieving $f = v$ from $S \oplus (f := u)$ is x.", with S the state of interest, f ranging over all fluents, and so forth [106]. As it so happens, if one encodes a domain using only boolean-valued fluents, then theory based in Causal Graphs becomes equivalent to theory based in Planning Graphs; the former is a strict generalization [106]. In other words, it casts a very wide net to assume that the internals of a planner are at least as powerful as reasoning about STRIPS-style actions using Planning Graphs [16]. Indeed, the net captures even 'non-search' approaches to planning [52, 126, 128, 176, 196].

In short the notion is to compile every temporally expressive language into one another, along the way demonstrating that planning graph based analysis is enough to perceive the forcing. There are caveats and exceptions. Call a language **syntactically super-classical** when it permits direct expression of Blocksworld. Then:

Theorem 4.12. *Every pair of syntactically super-classical temporally expressive languages in the scope of Figure 4.2 permits a forcing compilation in either direction.*

Proof in Outline. Given any problem in a source language the existence of a forcing compilation to a target language is witnessed by instead demonstrating a forcing compilation from any convenient super-language of the source to any convenient sub-language of the target. *I.e.*, it suffices to compile the maximum language into each of the minimal languages. Lemma 4.14 establishes the minimal languages, of which there are four. Lemma 4.15 completes the argument by compiling the top-level into each of the four. □

Caveat. The state-of-the-art is not quite sophisticated enough for our purpose. In particular temporally expressive planners fail, by and large, to implement any sort of reduction by consideration of causal independencies. To be able to construct forcing compilations we need such. In particular, for the forcing arguments to work, the state-of-the-art needs to advance by implementing a particular and interesting synergy between deordering and landmark analysis. For a placeholder:

Conjecture 4.13. *Relatively powerful forms of deordering and landmark analysis, as in Lemma 3.3, will prove empirically effective within temporally expressive planning.*

4.4.1 Scope

Some of the languages presently within scope are simply too restricted to be of practical interest. Specifically languages permitting preconditions nowhere would require us to invoke compilation to address even causally sequential problems. At a minimum it should be easy to state Blocksworld. Let us take as a baseline only those languages for which the abstraction into Conservative Temporal Planning (*i.e.*, into its image $\mathtt{L}(\mathtt{pre}, \mathtt{eff};\ \emptyset;\ \emptyset)$) also encompasses it. Call a language **syntactically**

strictly sub-classical if otherwise; a language is **syntactically super-classical** if it permits (i) primitive preconditions over at least one subinterval, and (ii) primitive assignments over at least one subinterval. (One may regard this as a better definition for degeneracy in the first place.) So L(∅; pre; eff) is a natural example of a super-classical language within scope. Naturally this is all an 'elaborate excuse' to ignore the temporally expressive cases we cannot compile to: L(∅; eff; eff) and L(eff; eff; eff).[6]

Lemma 4.14. *The languages of interest allow actions to depend upon fluents, alter fluents, and most importantly, state problems requiring concurrency of actions.*

---

[6]The two temporally expressive, syntactically sub-classical, sublanguages of ITP appear to be counterexamples to the spirit of the theorem, insofar as being only NP-complete, rather than PSPACE-complete. Proof of NP-completeness of L(eff; eff; eff): Take a POCL planning view. The only open conditions are top-level goals. Each needs/permits just one causal link. So the size of a set of minimal witnesses is bounded from above. Hence the appropriate variables are bounded by a polynomial (conceptually, # goals times # actions). The constraints for threat resolution should be polynomial as well, *i.e.*, no worse than quadratic in the variables. Then membership in NP is shown.

For hardness: Consider Satisfiability in Propositional Logic of arbitrary formulas in Conjunctive Normal Form. Setup a pair of complementary actions (*i.e.*, make-*p*-true, make-*p*-false) per proposition, a goal+fluent (*i.e.*, set-*p*) per proposition, and a goal+fluent (*i.e.*, check-*c*) per clause. Have each start-part refute the satisfaction of every clause (*i.e.*, delete every check-*c*). Have each end-part witness the satisfaction of the appropriate clauses (*i.e.*, *e.g.*, make-*p*-false adds check-*c* for every c such that $\neg p \in c$). Have each all-part indirectly witness that the corresponding proposition is given just one truth value (*i.e.*, add set-*p*), as follows.

Some action starts last (in the dispatch-order, the dispatch-times are largely irrelevant). Any action that ends before will have all of its useful effects overwritten, rendering it (largely) irrelevant. Otherwise an action starts before and ends after the key point (by maximality). Therefore all relevant actions are concurrent. Hence the relevant set includes at most one out of every complementary pair of actions (by the choice of all-parts and Proposition 3.12, *i.e.*, by the very strict locking protocol).

Then, by construction, if, and only if, the problem is solvable, the final set of pairwise concurrent actions names a partial assignment satisfying the formula. (The final start-part, all-false initial state, and all-true goal together ensure every clause is checked; the final set of end-parts witness the specific literals satisfying each clause; the final set of all-parts witness consistency of the set of literals chosen; in reverse, any partial assignment witnesses executability; any satisfying assignment further witnesses goal-achievement.) So hardness is shown, above and beyond membership, thus: NP-completeness of L(eff; eff; eff) is proven. □

Presumably L(∅; eff; eff) is also NP-complete; it suffices to find an encoding of the key mutual exclusion obtained 'for free' above. A theoretically interesting detail is that the ideal plan orders the decision concerning any irrelevant proposition before the key point. Optimizing the metric "the cost of a plan is the largest set of mutually concurrent actions" will result in a solution encoding a smallest conjunction entailing the formula.

There are 32 cases in the scope of Figure 4.2. There are 4 minimal cases:

`L(∅; pre; eff)`,		`L(∅; eff; pre)`,
`L(pre; eff; eff)`,	*and*	`L(eff; pre; pre)`.

Proof. There are $8 = 2^3$ languages within scope forbidding effects proper altogether: permit/forbid preconditions only, in 3 parts. Likewise 8 languages forbid preconditions altogether. Counting both double-counts the 1 language forbidding conditions and assignments altogether (*i.e.*, permitting nothing). So there are $15 = 8 + 8 - 1$ 'boring' languages: $49 = 64 - 15$ languages remain.

Of these, there are $17 = 3 + 14$ ways to forbid causally compound actions (as follows). By Theorem 4.1, to forbid causally compound actions is all and only the way to be temporally simple (within scope at least). Therefore there are $32 = 49-17$ syntactically super-classical temporally expressive languages.

Elaborating, let us count the super-classical languages permitting only causally primitive actions. There are 0 ways to allow precisely 0 temporally intersecting nontrivial parts. There are 3 ways to allow precisely 1 temporally intersecting nontrivial parts, *e.g.* `L(∅; ∅; pre, eff)`. With precisely 1 forbidden part, to achieve super-classical: there are (i) 2 ways to permit preconditions once and to permit effects once (picking one forces the other, in order to avoid forbidding 2 parts), (ii) 2 ways to permit preconditions twice and to permit effects once, (iii) 2 ways to permit preconditions once and to permit effects twice, and (iv) 1 way to permit preconditions and effects twice each. So there are $14 = 7 + 7$ ways to allow precisely 2 temporally intersecting nontrivial parts, in two symmetric groups of 7: forbid the start-part, or forbid the end-part (forbidding the all-part would force permitting

temporally disjoint parts). There are 0 ways to allow precisely 3 temporally intersecting nontrivial parts, because permitting all of them permits temporally disjoint parts.

For the minimal cases: only one of the four minimal nondegenerate temporally expressive languages (see Lemma 4.4) is sub-classical: $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{eff})$. Therefore minimality of the other three is shown: $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff})$, $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre})$, and $\mathtt{L}(\mathtt{eff};\ \mathtt{pre};\ \mathtt{pre})$ are minimal witnesses. It suffices to additionally enumerate just those super-languages of $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{eff})$ that minimally achieve super-classical status, *i.e.*, permit preconditions somewhere. Of the three possibilities, two are super-languages of the above witnesses, *i.e.*, the two non-minimal possibilities are: $\mathtt{L}(\emptyset;\ \mathtt{pre},\mathtt{eff};\ \mathtt{eff})$ and $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre},\mathtt{eff})$. The sole remaining possibility, $\mathtt{L}(\mathtt{pre};\ \mathtt{eff};\ \mathtt{eff})$, is, by inspection, not a super-language of the other three, hence, is the fourth and final witness. □

4.4.2 Compilations

So it remains only to prove that each of the four minimal cases captures the whole. There is just one syntactic ability that the minimal cases 'truly' lack: the ability to state primitive transitions. The game plan is to:

1. compile out primitive transitions by introducing extra parts per action, and

2. compile out the extra parts by introducing auxiliary actions and fluents.

In particular it is convenient to work in an intermediate form permitting any number of parts over arbitrary subintervals. It should be clear how to extend the machinery to directly support all the details. In any case we may adapt the compilation techniques of Halsey, Fox, and Long [72], also discussed by Smith [186].

Definition 4.7. Let PDDL* denote a discrete temporal planning language generalizing Interleaved Temporal Planning in the intended manner: allow actions access to arbitrary sub-intervals of their executions. Loosely, say:

- A compound action is given by a time-sorted sequence of parts.
- A part is given by an effect and its relative temporal extent.
- A relative temporal extent is a right half-open interval denoted by its start-time and duration.

So for notation, write that compound α is given by (with $i \in [0, n]$):

$$\alpha = (\mathit{eff}, ST, dur)_{[0,n]}, \tag{4.5}$$

$$ST_0 \leq ST_1 \leq \cdots \leq ST_n, \tag{4.6}$$

$$0 < dur_i, \qquad \text{and} \tag{4.7}$$

$$dur_0 \geq ST_i + dur_i. \tag{4.8}$$

Here n abbreviates the number of proper parts, for disambiguation write: $n = Parts_\alpha$. Further disambiguate if necessary by writing $ST_{\beta,2}$ for the relative start-time of the 2nd proper part of compound β. Part 0 is the all-part.

The definition leaves in the hands of the reader a number of easily addressed details, some consequences of which are surely not immediately obvious. For example, the effective unit of time (*cf.*, Corollary 3.23) is no longer the g.c.d. of durations. Rather "The g.c.d. of relative start-times and finish-times can be taken as the unit of time." is the accurate statement: $\mu = \gcd\{ST_{\alpha,i} \mid i \in [0, Parts_\alpha]\} \cup \{FT_{\alpha,i} \mid i \in [0, Parts_\alpha]\}$ is the right computation (with, of course, $FT := ST + dur$).

If we first accept that the following compilations are indeed correct, then a (painful) way to derive the formula would be to work backwards through the compilations. Of course an entirely proper formal account needs to independently and completely specify the semantics of PDDL* before proving that working backwards through the compilations ends up with equivalent definitions. As far as future work is concerned the upshot is that filling the gaps in the following proofs amounts to finishing the definition of PDDL* and proving the relevent generalizations of the fundamental propositions/lemmas/theorems.

In the course of the proof what we actually need is a restricted form: Say PDDL† denotes the sublanguage of PDDL* restricting to just primitive conditions and primitive assignments, *i.e.*: excluding primitive transitions. So each effect stipulates at most one of $f = v$ or $f := v$ per fluent: never both. (Slightly abusively we could write $\mathit{eff} \in (\texttt{pre}|\texttt{eff})^{\mathit{Fluents}}$.) From a semantic perspective, effects of this kind are of the form $\{P \cup E' \mapsto E \mid \mathrm{Dom}(E') = \mathrm{Dom}(E)\}$ subject to $\mathrm{Dom}(P) \cap \mathrm{Dom}(E) = \emptyset$.

Then the theorem, as already outlined, reduces to:

Lemma 4.15. *The top-level permits a forcing compilation into the minimal cases.*

Proof in Outline. By Lemma 4.16 we may compile the 'top-level' (but PDDL* is, of course, more general) into the intermediate form (PDDL†). By Lemma 4.17 we may compile the intermediate form into each of the minimal super-classical temporally expressive languages (as enumerated in Lemma 4.14). □

As a (necessary) precaution, let us begin by making sure that all of our various book-keeping manipulations will not collide with themselves in time. For every action α:

- Declare a (virtual) fluent α_{parity}; assume $\alpha_{\text{parity}} = \text{even}$ initially.

- Duplicate α—name one α_{odd} and the other α_{even}—and call both "real".
- Add $\alpha_{\text{parity}} = \text{odd} \coloneqq \text{even}$ as a primitive transition to the all-part of α_{even}.
- Add $\alpha_{\text{parity}} = \text{even} \coloneqq \text{odd}$ as a primitive transition to the all-part of α_{odd}.

Now observe that every two executions of a real action are quite separated in time (*i.e.*, the twin of each must intervene). For the following let us assume this manipulation has been done (whenever needed), and let us forget about it. That is, simply assume:

> No two executions of the same real action are ever too close in time.

Note that we can assert any constant as the definition of "too close", *e.g.*, by counting executions of α modulo some number larger than 2. We view the manipulation as harmless for various reasons, for example, it does not alter the number of executable plans.[7]

4.4.2.1 Compiling Out Primitive Transitions

So the first problem to solve here is that the minimal languages no longer guarantee us the ability to state primitive transitions. Abstractly, a key difficulty is that we cannot check that something is true and immediately delete that fact; at some later or earlier time in the action we may effect the fluent, but, not quite immediately. So for example, $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff})$ places great obstacles in the way of modeling even

[7] Technically, counting parity of executions of actions could increase the number of reachable situations. (This occurs when two schedules for reaching the same situation, prior to the manipulation, nonetheless differ in some parity.) Let us emphasize *situation*: the read-times, write-times, lock-types, and remaining obligations must all be identical as well. Which is relatively unlikely, to the point that: temporally expressive planners do not bother to implement duplicate situation elimination. Hence there is little practical significance to counting reachable situations.

just simple movement. The natural attempt is something like: at start check that the source and destination are connected, (and that the object is in fact presently located at the given source,) then, at end, assign the location of the object to the given destination. The problem with this is that nothing actually changes at the start of the action. So we could leave a given location in any number of directions; later on the formal model will (incorrectly) conclude that we successfully visit the whole neighborhood in a rapid series of 'teleports'. A correct model must redo the work undertaken by the machinery of locks: manually enforce the desired mutual exclusions. A fair amount of surgery is called for.

This may seem to be of little significance beyond serving as a step towards Theorem 4.12. Actually though it is not at all uncommon for temporal planning systems to differ upon fine details of mutual exclusion. So, while a relatively simple exercise, it does turn out to be of some practical significance to work out how to 'jitter' such fine details around in time. At least, as far as airtight empirical comparison is concerned we ought to ensure that the formal problems considered by two systems are 'as equivalent as possible'. A typical notion of equivalence would be to ensure that the precise count of solutions remains the same. Less usual from a theoretical stance is to prefer to compare systems when they work with graphs of similar size (presuming reduction to graphs . . .): even when of notably differing structure. (Which would mean that the formal semantics differ notably, but the comparison may nonetheless be interesting, perhaps because the intuitive semantics do not so differ.) Regardless of specific notion: detailed understanding of the various formal interpretations, and how to map between them, is surely useful.

Without further ado:

Lemma 4.16. *The top-level permits a forcing compilation into the language* PDDL†.

Proof. We have three subtasks: define the compilation, prove it correct, and demonstrate forcing. The compilation involves three surgical steps: double the number of fluents, stipulate particular effects upon them, and split primitive transitions upon the original fluents in two. Then we prove correctness, focusing on the direction of showing that (non-)executability of the compilation implies (non-)executability of the original. We gloss over the reverse direction; it is more interesting to prove a stronger result, namely, forcing.

The Compilation. Let μ stand for the unit-time of the original problem. Set the unit-time of the compiled problem as: $\hat{\mu} := \mu/2$. So we split every original unit of time in two. The first half serves to simulate the read/load half of a transition; completing the simulation, the second half simulates write/store. Then a portion of the book-keeping is dedicated to ensuring that only said book-keeping peers into these faster -than-μ changes.

Firstly, double the number of fluents: associate every real fluent f with a virtual fluent f_{lock}. These auxiliary fluents are surrogates, specifically, they serve to indirectly enforce the correct temporal constraints. They are unary—their only legal value is `True`.

Secondly, replicate, onto f_{lock}, the temporal constraints upon f, in detail:

1. For every primitive transition $f = u := v$, insert $f_{lock} :=$ `True`.

2. For every primitive condition $f = u$, insert $f_{lock} =$ `True`.

3. For every primitive assignment $f := v$, insert $f_{lock} :=$ `True`.

Clarifying, the first case takes precedence. So for example if the *only* thing some part of an action does to a fluent is to check $f = u$, then apply the second case. On a further technical note, call the insertions *virtual* (in opposition to *real*).

Thirdly, fake the semantics of primitive transitions. Say the original interval of change on f is from time s to time t. That is, in notation, say the temporal extent of the part is the right-half-open interval $[s, t)$. Then, to compile out primitive transitions, achieve the following (which PDDL† permits):

- Leave $f_{lock} \coloneqq \texttt{True}$ over $[s, t)$ as-is.
- Check $f = u$ over $[s, s + \mu/2)$.
- Carry out $f \coloneqq v$ over $[s + \mu/2, t)$.

(Leave primitive conditions and primitive assignments as-is.) Call the last part real (and the others virtual). As far as mapping between obligations goes, consider the first of the three parts above to be the image of the original.

Correctness. (Claim) The compilation is correct. That is, despite the compiled representation being sometimes 'wrong' about when the change in f occurs/begins ($s + \mu/2$ rather than s), still every plan ends up with effectively the same interpretation. Recall that the surgery ensures that every effect asserts read-locks or write-locks (as appropriate) upon the surrogates f_{lock}—for the correct intervals of time—and in the first two cases the manipulations end there: the locks upon f are merely copied to f_{lock}. So only the last case is of concern.

In the case that we have compiled out a primitive transition: (Claim) only the book-keeping surrounding the sole effect in question may observe the behavior of the fluent over the concerned interval. That is because write-locks are exclusive

and said interval is, indirectly, locked. Therefore it is of 'no concern' when we check $f = u$ and when we carry out $f := v$. We could, for example, have chosen to split the original interval into $[s, s + (t - s)/2)$ and $[s + (t - s)/2, t)$ instead.[8] Hence correctness is, roughly, shown.

For much greater detail, consider deducing the uncompiled form of an execution from the compiled form. Specifically, elide the various sequences in the following manner, with σ the mapping between dispatches:

- Let $\hat{S}_{\sigma(0)}, \hat{S}_{\sigma(1)}, \ldots, \hat{S}_{\sigma(n)}$ be the states just after the image of any real effect is applied (with the first for the initial state).
- Let $\hat{V}_{\sigma(0)}, \hat{V}_{\sigma(1)}, \ldots, \hat{V}_{\sigma(n)}$ be the vaults indexed in the same manner.
- Let $\hat{D}_{\sigma(0)}, \hat{D}_{\sigma(1)}, \ldots, \hat{D}_{\sigma(n)}$ again elide changes wrought by virtual effects.

Furthermore reduce their internal structure by throwing out all the book-keeping: details following. As observed above, only primitive transitions are of any significant concern.

(Case: State-sequences.) In the compiled problem we check the condition $f = u$ between states $\hat{S}_i$ and $\hat{S}_{i+1}$ (*i.e.*, of executions) and carry out the change between later states $\hat{S}_j$ and $\hat{S}_{j+1}$. From the preceding high-level argument, nothing between i and j can touch f (besides the compilation of the single effect): $\hat{S}_i(f) = \hat{S}_j(f) = u$. In the original problem we merely carry out both computations at once. So

[8] For $t = s+\mu$ as fast as possible, splitting the original interval in half is, entirely uncoincidentally, exactly the same as picking $s + \mu/2$ for the endpoint of the read portion. That is, the significance is that for every choice of x splitting $[s, t)$ into $[s, s + x)$ and $[s + x, t)$ then both x and $t - (s + x)$ are weakly larger than the new unit of time: $x \geq \gcd\left\{ST_a, FT_a \mid a \in \widehat{Primitives}\right\} = \hat{\mu}$ and $t - s - x \geq \hat{\mu}$. So considering $t - s = \mu$ as small as possible: $x = \mu/2$ is 'optimal'. If, though, $t > s + \mu$ is not as fast as possible, then it would be possible and perhaps better to split into $[s, s + \mu)$ and $[s + \mu, t)$ instead. There are naturally a great many such potential optimizations ignored by the argument.

set $\sigma(k) = i$ and $\sigma(k+1) = j+1$. Then more precisely, the point is that the equalities $S_k = \hat{S}_{\sigma(k)}\restriction_{Fluents}$ and $S_{k+1} = \hat{S}_{\sigma(k+1)}\restriction_{Fluents}$ are easily verified. In words: The state-sequences of the original executions are subsequences of the compiled forms, throwing away the book-keeping.

(Case: Vault-sequences.) Let decompile-vault($\hat{V}$) ignore the locks on real fluents f; instead, take the lock upon f to be the lock upon its surrogate, f_{lock}. So, with $V =$ decompile-vault($\hat{V}$), then define:

$$V := \left\{ f \mapsto \hat{V}(f_{lock}) \right\}. \tag{4.9}$$

Formally we ought to now check that the formal machinery of vault transition functions computes identical structures. (*I.e.*, check $V_i =$ decompile-vault($\hat{V}_{\sigma(i)}$).) We skip the exercise, but there is an interesting assumption worth extracting from those details. Specifically we assume: the identity of vault transition functions are independent of the precise definition of an effect. That is, all that matters is which fluents are read from, and which written to; the values themselves are irrelevent (*cf.*, Chapter 2). In other words, the correctness of the duplication-surgery follows chiefly from the (by design) independence of the locking behavior upon f from the behavior of f itself.

(Case: Debt-sequences.) We have left generalization of debt-sequences to the context of PDDL* in the hands of the reader; from any proper definition it should be apparent that $D_i(\alpha) = \hat{D}_{\sigma(i)}(\alpha)\restriction_{StrictIntervals}$ holds. Which more or less merely comes down to saying that: after applying the (compiled) all-part we should have some structure such that the substructure excluding book-keeping directly records the promised start-times of the start-part and end-part.

Then to summarize, roughly, the compilation of problems:

- Inject (into PDDL*) the various transition functions 'unaltered' but for extensions to larger domains.
- Use the surrogate fluents to decouple the injections of state transition functions and vault transition functions.
- Tweak the state transition functions by splitting $f = u$ and $f := v$ apart (thereby achieving the restriction to PDDL†).

Likewise summarizing the decompilation of executions of plans:

- Set $\sigma(i) := j$ when dispatch j of the compiled plan is the ith real dispatch.
- Set $S_i := \hat{S}_{\sigma(i)}\restriction_{Fluents}$.
- Set $V_i :=$ decompile-vault($\hat{V}_{\sigma(i)}$).
- Set $D_i(\alpha) := \hat{D}_{\sigma(i)}(\alpha)\restriction_{StrictIntervals}$ for every $\alpha \in Compounds$.

Verifying that the alternative way of computing executions (compile, solve, decompile) is equivalent is one direction (*i.e.*, soundness) of a proof of correctness. We should also demonstrate the reverse direction (*i.e.*, completeness): there does in fact exist some compiled representation/interpretation of any given original plan. Which is 'obvious': simply expand any original sequence of dispatches in place. Specifically, expand parts containing primitive transitions into their three compiled parts, say: lock, read, and write.

More formally, call a compiled plan **endpoint-interleaving-sorted** (in analogy to classically-sorted), for the following just say *canonical*, if all parts of any

compiled action appear as two contiguous subsequences of the plan. Then, as identifying a witness demonstrates existence, observe: for every executable plan of the original, the corresponding *canonical* compiled plan is also executable. Perceiving such is straightforward, *i.e.*, compose together the various subsequences of transition functions guaranteed to be contiguous in any canonical plan.

The practical upshot is that a target planner could prune away (preserving completeness, *etc.*) consideration of, more or less, all but the canonical plans. Achieving such pruning thereby defeats (some of) the criticisms of Smith [186]. The strongest possible form of such a rebuttal is for the planner to automatically produce a proof (of the correctness of the pruning) based on *just* the compiled problem it is given. That is, we want a planner to realize that a problem is the result of having carried out a compilation: without being told so. This is a tall order. There is a world of difference between verifying that a given equivalence reduction is correct, and finding one without being told to look for it. In other words, we come to demonstrating forcing.

Forcing. We first reiterate that the point is a matter of conjecture. Specifically we conjecture that temporally expressive planners will be able to effectively operationalize Theorem 3.21, *i.e.*, effectively operationalize the notion of deordered-equivalence. Which is perhaps a significant leap of faith. Having done so, then, for the present purpose it suffices to show that precisely one member of every deordered-equivalence class is canonical in the precise sense of “endpoint-interleaving-sorted”.

We shall do so in the spirit of Lemma 3.3. Namely, it suffices to demonstrate that: a planner could prove to itself that: having begun the "lock part" of the compiled form of a primitive transition, it is:

1. necessary to carry out the "read part" and "write part" in that order, and
2. sufficient to do so immediately.

This is because such are the only new parts introduced by the compilation, *i.e.*, these are the only parts that need to be reordered in order to reach the desired canonical form. Then, for the only case of interest, the last dispatch, say ℓ, added to the plan is of the "lock part" of the compilation of some primitive transition $f = u := v$. Let r and w stand for the respective dispatches of the "read part" and "write part".

In the context of PDDL*, (1) is free: Having begun any part of an action one is obligated to carry out its entirety.

For (2), for analogy, consider Proposition 3.2; consider the property that nothing mutually exclusive can intervene (between 'now' and when the remaining parts are added to the plan). Then each of the following (true) propositions/assertions is readily seen from examination of just the compiled problem. (And arguably worthy of investigation themselves, or as necessary consequences of more general observations, *i.e.*, worth analyzing without being told that the problem is the result of compilation.)

- The only things mutually exclusive with r or w touch f.
- Everything touching f is concurrent with something, say ℓ', touching the surrogate f_{lock}.

  Moreover such ℓ' are ℓ only for r and w. That is, anything mutually exclusive with r or w is concurrent with something (a) not ℓ, and (b) touching f_{lock}.

- The last dispatch, ℓ, more specifically acquired a write-lock upon f_{lock} over $[\mathrm{AST}(\ell), \mathrm{AFT}(\ell))$.

  As write-locks are exclusive, it follows: Everything mutually exclusive with r or w starts no sooner than $\mathrm{AFT}(\ell)$.

- The machinery gives us for free that ℓ contains r and w, *i.e.*, more specifically $\mathrm{AFT}(r) \leq \mathrm{AFT}(\ell)$ and $\mathrm{AFT}(w) \leq \mathrm{AFT}(\ell)$ can be read off from the current situation.

  It follows that nothing mutually exclusive with r or w may be added to the plan before they are. (For example, $\mathrm{AFT}(r) \leq \mathrm{AFT}(\ell) \leq \mathrm{AST}(\ell') < \mathrm{AFT}(\ell') \leq \mathrm{AST}(r) < \mathrm{AFT}(r)$ would be a contradiction.)

- In other words, the dispatches of the remaining parts of the compilation of a primitive transition are 'temporal landmarks' for everything they are mutually exclusive with, in every situation where they are possible (*cf.*, Lemma 3.3).

  Hence, if they are ever to be included it suffices to do so immediately, confirming (2) as desired.

Then to bring the argument to a close, we: compiled out primitive transitions, verified correctness, and demonstrated forcing relative to conjectured generalizations of standard technique. □

Discussion of Related/Future Work. The future work here is to entirely ground the preceding argument into an implementation. This is far from trivial. It is interesting to highlight some of the challenges to be tackled, relative to generalizing the techniques of classical planning.

The problem analysis needed here will be at least at the level of $h^2(\cdot)$ analysis over multi-valued fluents, in order to see the binary relationship between f and f_{lock}. As far as such generalization of planning graphs to non-boolean fluents is concerned, it is useful to consider the work in and surrounding Fast Downward [102].

More likely the analysis will need to go as deep as $h^4(\cdot)$ to see the quaternary relationship between, for example: the "write part", its 'container' (the "lock part"), any hypothetical future primitive also touching the fluent, and its 'container'. Here the work in *fluent merging* is a somewhat promising approach [97, 197]. That is because, intuitively speaking, the relationship is binary between just the two primitive transitions; that understanding is in theory recoverable *via* fluent merging. In particular by merging the fluents and their surrogates together, and then conducting $h^2(\cdot)$ analysis on that, one could also discover the truth of the particular propositions needed for the proof.

It is furthermore interesting to note that the analysis depends, somewhat, upon the temporal details/constraints. That is, what the machinery gives us for free as necessary is a particular *dispatch* of a particular primitive. Likewise, that it is sufficient to dispatch immediately concerns an analysis of (the impossibility of) all mutually exclusive dispatches. In contrast, Lemma 3.3 is stated for plans consisting of *primitives*. So the point is that the existing work in classical planning must be generalized to better address the distinction between primitives and dispatches thereof.[9]) Some significant work in this direction considers combining the tech-

[9]Consider compiling all the way down to classical planning. To sieze the pruning opportunity here points to an analysis of at least *six* entities: a primitive, *its dispatch-time*, an associated containing primitive, a hypothetical mutex primitive, *its dispatch-time*, and another associated containing primitive. For that matter it would be unsurprising if the dispatch-times of the containers were equally crucial. Consideration of $h^6(\cdot)$, or greater, over multi-valued and numeric fluents to boot, is, shall we say, less than promising.

niques of Linear Programming with Planning Graphs [37]. The work in Temporal Fast Downward (TFD) is also certainly worth investigation [64]. Especially the work of Bernadini and Smith within TFD considers quite specifically automated analysis of temporal planning problems [14].

4.4.2.2 *Compiling Many Parts into Few*

Our last, and hardest, sub-problem is to compile out the accesses to arbitrary sub -intervals of action executions. This is a bit trickier than usual [72, 186]. That is because the syntax restrictions of the minimal cases are quite excessive. Primarily, the details complete our formal analysis of how far the notion of merely two levels of temporal expressiveness can be taken.

The ulterior significance here is in regard to promising approaches to extending the state-of-the-art. Firstly, defeating the mechanisms—proving their absence —works toward proving that particular subsets of primitives may be abstracted into primitives without loss. (Which amounts to 'solving' *causally required concurrency*.) Secondly, implementing the analysis in support of forcing works towards proper justification of deliberately limiting, internally, to ITP rather than implementing direct support for, say, PDDL* (which is more well-motivated in practice). Conversely, the difficulty of doing so effectively may be taken as justification for (rather than justification against) directly implementing support for PDDL*.

For analogy, that compilation to SAT can be, and often is, an effective way to solve a breathtaking variety of problems turns on the sophisticated inference being carried out by such solvers in the name of efficiently inferring forced decisions (*i.e.*, unit-propagation, clause-learning, and variable-(re-)ordering are all quite important to the effectiveness of SAT as a *combinatorial substrate* for many kinds of

problems). Likewise, whether ITP can be taken as well-motivated turns on the sophistication of implementations of discrete temporal planning. More specifically, part of the significance here goes towards fleshing out the technical meaning of Conjecture 4.13: an effective implementation would happen to be strong enough to see through the following—and so also many other more well-motivated—compilations.

Then without further ado: the heart of the proof of Theorem 4.12 lies in demonstrating the following.

Lemma 4.17. *The intermediate form* PDDL† *permits forcing compilations into the minimal cases of Lemma 4.14.*

Proof. We have four minimal languages to compile into. For each we need to (a) define the compilation, and (b) prove it correct. We also argue towards (c) forcing, but there are technical holes remaining, discussion of which is inline. The general idea is to:

- split up any given action (α) into a chain-decomposition (β),
- compile each piece into its own action ($\hat{\beta}$ for all of them),
- setup an envelope to contain them all (*do*), and
- use two matched pairs of virtual actions to workaround the syntax restrictions that render difficult the expression of the conceptually simple relationship between the envelope and its contents.

The compilations are broadly similar, beginning with the fact that all separate over actions; the compilation of a problem is the result of separately compiling each

action. (In contrast, eliminating primitive transitions called for global surgery.) So let $\alpha = (\mathit{eff}, ST, dur)_{[0,n]}$ be some compound action consisting of a time-sorted sequence of parts. Then intuitively speaking the idea is to break this up into a sequence of PDDL-style compounds $\hat{\alpha}_i$: one per part. Further book-keeping then serves only the purpose of ensuring that dispatches of the whole sequence end up contiguous in time, and, ideally, contiguous in dispatch-order as well.

(α to β). More accurately, the compilations need to account for overlapping parts. Consider the natural partial-order on the parts of α: $\alpha_i \leq \alpha_j$ whenever $FT_i \leq ST_j$ (where of course $FT \coloneqq ST + dur$). Let (1) β denote a *chain decomposition* of this partial-order, (2) augmented by inserting do-nothing *struts* between parts adjacent in a chain but not contiguous in time, (3) presented more or less as a reformulation into a set of compound actions. That is, let $\beta_1 = (\mathit{eff}, ST, dur)_{[k_1]}$ *partition* the interval $[0, dur_{\alpha,0})$ into a k_1-length sequence of parts, each either with trivial effect, or, equal to some arbitrary choice of a proper part of α. Then let β_2 do the same, but exclude those parts of α already covered by β_1. Say by β_k we exhaust all of α but for its all-part, α_0. Let $\beta_{0,1} \coloneqq \left(\mathit{eff}_{\alpha,0}, ST_{\alpha,0}, dur_{\alpha,0}\right)$ finish off the decomposition (by picking just the all-part of α). So among other things we have $k + 1$ chains, of lengths k_i ($k_0 = 1$), such that the $n + 1$ parts of α all appear precisely once. Then, refining the outline view: The bulk of each of the compilations consists of converting each such part $\beta_{i,j}$ of the decomposition/partition into its own action $\hat{\beta}_{i,j}$, with some additional work to ensure that the latter are forced to occur contiguously in time.

Discussion of Durative Effects/Conditions. We shall exploit a feature of our semantics with ‘no support whatsoever’ in current practice; precious few temporal plan-

ners permit one to specify the durations of discrete effects. Indeed, most pretend that such are instantaneous. Given that the truth of the matter is that such do have duration within the formal semantics of the planners (*i.e.*, as implemented), and in particular that such durations effectively dictate the domain of time, it seems to us to be a self-defeating limitation of existing practice to insist that such intervals be 'as small as possible'. Instead it seems that such systems should benefit, or at least be unharmed, by taking effects at endpoints to be of significantly longer duration. In support of that point, consider that at the extreme (the effects anchored at endpoints occupying the whole interval of the action), we recover the semantics of Conservative Temporal Planning (*i.e.*, we simplify to the setting of TGP).

So we have some preference for permitting direct specification of the durations of effects. In any case we do not really have to exploit this feature of our definitions; there are other ways to get the job done in paradigms of instantaneous or nigh-instantaneous effects [72]. As we do not anyways expect our compilations to be of *direct* practical significance, let us simply take the route at present most formally convenient. Namely, we exploit—in the following manner—setting the durations of the start-part, end-part, and all-part of an action to be the same.

(β to $\hat{\beta}$). Long story short, we may indeed achieve the picture painted above, in any of the minimal cases, despite their severely restricted form. Which means that, in every minimal case, we do literally take $\hat{\beta}_{i,j}$ to be an action (with every part of equal duration) encoding part $\beta_{i,j}$ of the decomposition/partition β of the original action α. The manipulation here is to take the conditions of $\beta_{i,j}$ and place them in a syntactically valid part of $\hat{\beta}_{i,j}$, likewise for its effects. By Lemma 4.16, we have already ensured that doing so does not result in patently self-mutex/self-contradictory ac-

tions. For notation, let us setup the formal definition *sans* book-keeping initially, and then add in the book-keeping by redefinition (*i.e.*, yet more formally we would separately name the intermediate structures). So, with:

- $i \in [0, k]$ the index of a chain of the decomposition/partition of α,
- $j \in [k_i]$ the index of a part along that chain,
- $b = \beta_{i,j} = (\mathit{eff}_b, ST_b, dur_b)$ the part of α (or *strut*) in question,
- $\hat{b} = \hat{\beta}_{i,j}$ the compilation of the part into a 'compound' action,
- p naming a part permitted to express primitive conditions,
- e naming a part permitted to express primitive assignments,
- x naming any other parts,
- $\{P \cup E' \mapsto E \mid \mathrm{Dom}(E')\} = \mathit{eff}_b$ destructuring the effect of b,

initially set, in the case that p and e are distinct:

$$\widehat{ActionDefs}((\hat{b}, p)) := (\{P \mapsto \{\}\}, dur_b), \tag{4.10}$$

$$\widehat{ActionDefs}((\hat{b}, e)) := (\{E' \mapsto E \mid \mathrm{Dom}(E') = \mathrm{Dom}(E)\}, dur_b), \tag{4.11}$$

$$\widehat{ActionDefs}((\hat{b}, x)) := (\{\{\} \mapsto \{\}\}, dur_b), \tag{4.12}$$

or, in some non-minimal case where $p = e$ is possible then initially set:

$$\widehat{ActionDefs}((\hat{b}, p)) := (\{P \cup E' \mapsto E \mid \mathrm{Dom}(E') = \mathrm{Dom}(E)\}, dur_b),$$

$$\widehat{ActionDefs}((\hat{b}, x)) := (\{\{\} \mapsto \{\}\}, dur_b), \qquad \text{(for both } x\text{).}$$

Observe: To ensure the compilation is correct it suffices to guarantee that if any $\hat{\beta}_{i,j}$ occurs, then they all do: *contiguously in time*.

For uniformity through the cases let us use something resembling an envelope to ensure the temporal constraint [72, 186]. As a warm-up, consider ensuring that all of some set X only ever occurs after x and before x' (in the dispatch-order):

- Create a boolean fluent $executing_X$, always initially and finally false.
- Add $executing_X = \texttt{True}$ to each element of X.
- Add $executing_X \coloneqq \texttt{True}$ to x.
- Add $executing_X \coloneqq \texttt{False}$ to x'.

If, as we may freely assume (by Theorem 3.21), the dispatch-order is time-sorted, then also we have the desired containment relationship in time as well. As far as intuition goes that is how envelopes work. Accurately though, we can, and usually do, omit the above manipulation. The reason is that, rather than a set, we have a partial-order; enforcing the ordering constraints, as it so happens, goes a long way towards enforcing containment.

(Book-keeping: Ordering Constraints). For the following, continue to let $\hat{\mu}$ denote an effective unit of time for the compiled problem (half of the original unit); much of the book-keeping consists of such unit-duration parts or actions. Enforce the partial-order on which β is based, for $i \in [0, k]$ and $j \in [k_i]$, by the following manipulations.

- Create boolean fluents (*token* i j), always initially and finally false.
- Add (*token* i j) $= \texttt{True}$ to $\hat{\beta}_{i,j}$.

- Add $(token\ i\ (j+1)) \coloneqq$ `True` to $\hat{\beta}_{i,j}$.
- Add $(token\ i\ (j-1)) \coloneqq$ `False` to $\hat{\beta}_{i,j}$.
- Create book-keeping actions *setup* and *reset*.
- Create boolean fluents $(token\ i\ 0)$, always initially and finally true.
- Create boolean fluents $(token\ i\ (k_i+1))$, always initially and finally true.
- Add $(token\ i\ 0) =$ `True` to *setup*.
- Add $(token\ i\ 1) \coloneqq$ `True` to *setup*.
- Add $(token\ i\ (k_i+1)) \coloneqq$ `False` to *setup*.
- Add $(token\ i\ (k_i+1)) =$ `True` to *reset*.
- Add $(token\ i\ 0) \coloneqq$ `True` to *reset*.
- Add $(token\ i\ k_i) \coloneqq$ `False` to *reset*.

(To add statements to such actions, without qualification, add it to whatever part in fact permits statements of that kind, as detailed above.) We may check that no fluent is accessed twice at any given point in time, which is enough to ensure that the book-keeping will be able to meet the syntax restrictions of the minimal cases (without running afoul of mutual exclusions).

As far as correctness goes the first observation to make is that: the sums of the durations of the compilations of each part along each chain is the correct relative start-time of the next part. For notation: $ST_{\beta_{i,j}} = \sum_{1 \le k < j} dur(\hat{\beta}_{i,k})$. This holds by the construction of β, specifically we demanded that each chain be augmented by inserting do-nothing struts of the proper durations. So fastest dispatches of the

chains start all the parts at the right times. Then all we need to do, as far as ensuring that the relative start times are correct, is to ensure that the fastest dispatch is the only legal dispatch.

If the minimal cases permitted us to downgrade the *setup* and *reset* actions into mere primitives then we would be immediately done. (Here is where related compilation arguments stop, more or less.) Specifically we would finish by making these into the start-part and end-part of an action with appropriately limited duration. However, as each uses both conditions and assignments, in the minimal cases they are syntactically invalid as mere primitives.

The second observation to make is that the book-keeping is employing a standard manipulation for ensuring that any given chain occurs uniquely: a *token* is passed along each. Normally one would require and immediately delete the prior token, making uniqueness very clear. Here we (sortof) delete the tokens too late. That is, at a purely propositional/classical level the machinery does *not*, entirely, prevent unwanted repeats of the various parts. As it turns out, merely ensuring the temporal correctness (by making the fastest dispatch be the only dispatch) also ensures the classical correctness (in particular, that each part of the simulation occurs once and only once per simulation). That is, ensuring that each part starts at the right time also ensures that copies of parts cannot appear within a single simulation of the entire action; copies are mutex with one another, so, forcing all copies to start at the same time means that no copies actually exist. For the present purpose then it suffices merely to enforce the temporal constraint: the uniqueness property follows.

More generally the uniqueness property is of interest regardless of the temporal constraints. That is, besides correctness of this compilation, the uniqueness property is of direct interest towards search-space reductions as in Lemma 3.3.

In short, we demonstrate correctness of the compilation(s) in four parts: *existence*, *ordering*, *duration*, and *uniqueness* (which is implied by *duration*). Roughly, we show that if any part occurs then every part:

- *(existence)* exists at least once,
- *(ordering)* is ordered correctly, and
- *(duration)* is dispatched as fast as possible; it follows that
- *(uniqueness)* every part 'exists at most once', more precisely, it follows that every part is included at most once (and so exactly once by *(existence)*) per simulated instance of the action α.

Moreover we desire a proof amenable to implementation, *i.e.*, we desire to demonstrate something resembling forcing.

(Existence and Ordering by Landmarks). Then we begin by noting a number of easily inferred landmarks, with $i \in [0, k]$:

- The *reset* action separates those states where the *setup* action is not executable from those states where it is. Precisely, the *reset* action separates states satisfying $(token\ i\ 0) = \texttt{False}$ from states satisfying $(token\ i\ 0) = \texttt{True}$; because no other action has the effect $(token\ i\ 0) := \texttt{True}$. For notation:

  Each $(\neg(token\ i\ 0), reset, (token\ i\ 0))$ is an action landmark.

- The *setup* action likewise separates executability of $\hat{\beta}_{i,1}$. For notation:

  Each $(\neg(\mathit{token}\ i\ 1), \mathit{setup}, (\mathit{token}\ i\ 1))$ is an action landmark.

- For each $j \in [k_i]$, the action $\hat{\beta}_{i,j}$ separates executability of $\hat{\beta}_{i,j+1}$. For notation:

  Each $(\neg(\mathit{token}\ i\ (j+1)), \hat{\beta}_{i,j}, (\mathit{token}\ i\ (j+1)))$ is an action landmark.

Or, taking $j \in [0, k_i + 1]$, performing the arithmetic on j modulo $(k_i + 1) + 1$, understanding $\hat{\beta}_{i,k_i+1}$ as the *reset* action, and understanding $\hat{\beta}_{i,0}$ as the *setup* action, then we may write just: Each $(\neg(\mathit{token}\ i\ (j+1)), \hat{\beta}_{i,j}, (\mathit{token}\ i\ (j+1)))$ is a trivially inferrable action landmark. We may likewise note the uniqueness of the delete effects, continuing with $i \in [0, k]$, $j \in [0, k_i + 1]$, and the modular arithmetic (*etc.*):

- Each action $\hat{\beta}_{i,j}$ separates 'relevance' of its (intended) predecessor $\hat{\beta}_{i,j-1}$. By which we mean only that each $((\mathit{token}\ i\ (j-1)), \hat{\beta}_{i,j}, \neg(\mathit{token}\ i\ (j-1)))$ is an action landmark.

Summarizing: the only way to make the token at some particular index true is to execute its intended predecessor, similarly the only way to then make the same token false is to execute its intended successor. Which is obvious—even to rather naïve automated planners—as the statements merely formalize the uniqueness of the add and delete effects on the tokens.

Far less obvious is the inductive closure: the following landmarks. For all $a \leq b$, $b < c$, $a' \equiv a$, $b' \equiv b$, $c' \equiv c$, under equivalence modulo $k_i + 2$, then:

$$((\mathit{token}\ i\ a'), \hat{\beta}_{i,b'}, (\mathit{token}\ i\ c'))\ \text{are each action landmarks.} \tag{4.13}$$

Which means: If some token is true now, and later some other token within the same chain is true, then *every* operator along the directed cycle between those two

tokens occurs (in order). For an example, if the third token is true now, and the first token is true later, then: understanding $1 \equiv k_i + 3$, infer that every $\hat{\beta}_{i,j}$ occurs between now and then (in order) except for $\hat{\beta}_{i,1}$ and $\hat{\beta}_{i,2}$. In other words, to perceive the truth of Equation (4.13) is to perceive *(existence)* and *(ordering)*.

Perhaps these landmarks are intuitively clear enough; perhaps not. Regardless: How hard are they to prove? Consider the following invariant analysis.

(Invariant Detection in Support of Forcing). It is true that, for all original actions α, for all reachable states S, and for all $i \in [0, k]$:

$$|\{f \mid f = (token\ i\ j) \text{ and } S(f) = \texttt{True}\}| = 2.$$

The fact almost follows just by noting that every relevant operator deletes one member of the set and adds another (*i.e.*, is balanced in the sense of Bernadini and Smith [14]). The tricky aspect to proving the invariant is to guarantee that the delete effects are only ever applied when there is something to delete. To notice such more or less requires the refined observation, for all $b \not\equiv a \pm 1$:

$$S((token\ i\ a)) = \texttt{False} \text{ or } S((token\ i\ b)) = \texttt{False}.$$

That would follow from, again for $b \not\equiv a \pm 1$, the hypothetically true:

$$h^2((token\ i\ a) = \texttt{True} \text{ and } (token\ i\ b)) = \infty.$$

Whether a straight up planning graph (with mutex propagation) might indeed notice the invariant is an interesting question [16, 99]. Our point here is just that significant

negative interactions may be detected by standard technique, even if not precisely as above: motivating the following deeper, more expensive, analysis.

Let $X_i = \{(token\ i\ j) \mid j \in [0, k_i + 1]\}$ be the propositions relevent to chain i. Consider the state-space graph of the temporal problem; *i.e.*, ignore all the temporal aspects, focusing only on the classical constraints. Naturally, only the reachable component is of interest. For any given i: Contract together every vertex of the reachable component differing only in the values of fluents outside of X_i. The resulting graph is called a **Domain Transition Graph**: write, say, $\mathrm{DTG}(X_i)$.

Such graphs may be built in poly-time with respect to their size, by exploiting a logically equivalent definition.[10] Namely, we may project/abstract the original problem onto the fluents X_i, the meaning of which with respect to state-independent effects is just to erase every statement concerning unrelated fluents. (Supporting the general case, state-dependent effects, is more complicated, but in principle remains straightforward enough.) We care for the contraction viewpoint because it makes the relationship to the original problem very clear. Specifically: if we cannot find paths in a Domain Transition Graph, then we cannot find paths in the full state-space, so we certainly cannot furthermore find schedules solving the full temporal problem.

Then consider the following fact, which we claim without further (meta-)proof. The state of the art in exploiting Domain Transition Graphs can prove [107], provided the set of fluents X_i encoding the state of the token, that, for all $i \in [0, k]$:

$$\mathrm{DTG}(X_i) \overset{iso}{=} C_{k_i+2} + L_{k_i+2},$$

[10] To be accurate, building a DTG in practice is not precisely identical to contracting the reachable component of state-space. To make the proof work all that we require are sufficiently tight overestimates of reachability, which does here hold.

where C_ℓ denotes the directed cycle on ℓ vertices:

$$C_\ell = ([\ell], [\ell], \{x \mapsto (x, x' + 1) \mid x' \equiv x \pmod{\ell}\}), \qquad \text{and}$$

where L_ℓ denotes the graph on ℓ vertices consisting solely of directed loops:

$$L_\ell = ([\ell], [\ell], \{x \mapsto (x, x)\}).$$

Discussion of Domain Transition Graphs for Automated Detection of the Token-Passing. The claim here is that, given the identity of the right fluents to consider, subsequently writing out the corresponding contracted state-space is a mechanical, feasible, exercise. Doing so results in said augmented/loopy cycle, which the reader may readily verify; the only hypothetically tricky part to full automation is guessing which fluents to consider. That guess is somewhat important, because proof of the invariant fails if one considers anything less than all of the tokens along a given chain. (For $Y \subset X_i$, in general $|V(\text{DTG}(Y))| = 2^{|Y|}$; only for $Z \supset X_i$ do we have $|V(\text{DTG}(Z))| \leq 2^{|Z \setminus X_i|}|X_i|$, and so only by guessing X_i correctly will automated analysis be feasible.) However, it seems not too large a stretch to suppose that an automated planner could (feasibly) manage to figure out on its own that these chains of tokens are all quite relevant to one another, *i.e.*, worthy of deeper analyis. (By, for example, building h^2 and taking the presence of many negative interactions as motivation for merging the fluents together.) We say so, in large part, because the machine is free to make many wrong guesses: just not exponentially many. For an example of a closely related working implementation, see the work of Bernadini and Smith [14].

Detecting the loopy cycles plays into landmark analysis very nicely, because every non-loop edge is a cut (*i.e.*, a landmark). (In the general case one looks not for cycles but rather for domain transition graphs with many small cuts; especially interesting from the point of view of forcing are those vertices with just 1 leaving edge.) Specifically, all the landmarks described by Equation (4.13) fall directly out of the cuts of a loopy cycle. In other words, as every solution always projects onto a (legal) walk through any given Domain Transition Graph, and the graphs we have describe the right orderings for the pieces of the simulation of α, we already have that every solution looks quite close to 0 or more correct simulations of α. What is missing is to prove that the loops can be pruned away (for *(uniqueness)*), and that the temporal constraints are satisfied (for *(duration)*).

(Regular Expressions for Denoting Complex Landmark Knowledge). We may view Domain Transition Graphs as Deterministic Finite Automata, and hence as *regular expressions*. Let us indeed take regular expressions as inspiration for notation, with some small liberties for the sake of simplicity. In particular let us denote our knowledge of the token-passing by writing, for each i:

$$(\mathit{setup}+, \hat{\beta}_i+, \mathit{reset}+) * . \tag{4.14}$$

The liberty we take here is with respect to the contents. To be accurate we should express that each $\hat{\beta}_{i,j}$ may individualy repeat in isolation (rather than implying that each chain can internally repeat as a whole). That is, to be formal we should roll out the notation as in, for each i:

$$(\mathit{setup}+, \hat{\beta}_{i,1}+, \hat{\beta}_{i,2}+, \ldots, \mathit{reset}+) * . \tag{4.15}$$

Allow the abuse, but just for the following and just for the compiled view $\hat{\beta}$ of the action α.

(Duration/Uniqueness by an Envelope). What has yet to be shown is *(duration)*: the fastest dispatch is the only dispatch. We need this for two reasons; doing so ensures that all the various parts of the simulation are dispatched at the correct relative times, as well as ensuring that all the various parts of the simulation occur only once per iteration, *i.e.*, for *(uniqueness)*.

Regarding the latter, for motivation, consider say $\hat{\beta}_{1,2}$ and $\hat{\beta}_{1,3}$, and suppose some x elsewhere in the full problem writes to a fluent that $\hat{\beta}_{1,2}$ also does. Suppose further we take into account only the classical constraints (the state-sequences of executions). Then we will fail to notice non-solutionness of plans with subsequences such as: $\hat{\beta}_{1,2}, x, \hat{\beta}_{1,2}, \hat{\beta}_{1,3}$. In particular note that the latter instance of $\hat{\beta}_{1,2}$ is potentially useful, rather than a no-op. (Typically we will exclude the immediate repetitions based on inferring their uselessness.) Also note that, thus far, there is no constraint preventing such unintended double use of pieces of the compilation. So we need to extend the book-keeping to exclude such *internal repetitions*.

To exclude internal repetitions, we wrap all of *setup*, $\hat{\beta}$, and *reset* inside of an envelope with duration too short to permit repeats. Specifically, say *do* is a new book-keeping action of duration $dur_{\alpha,0} + 4\hat{\mu}$. To deal with the particulars of the minimal cases, we need another matched pair of book-keeping actions (*i.e.*, very much like *setup*/*reset*). So say *before* and *after* are book-keeping actions (of durations $\hat{\mu}$), serving to separate (desirable) repeats of *do*. The details differ somewhat in each case, but the general idea is the same. Namely, we want to force plans to

look like (when viewed in DTG(X_i), for each i):

$$(\mathit{before}, \mathtt{bgn}\text{-}\mathit{do}, \mathit{setup}, \hat{\beta}_i, \mathit{reset}, \mathtt{fin}\text{-}\mathit{do}, \mathit{after}) * .$$

The strategy is to—per language, problem, and original action instance—ensure:

1. $(\mathit{before}+, \mathit{setup}+, \mathit{reset}+, \mathit{after}+)*$.
2. $(\mathit{before}+, \mathit{do}+, \mathit{after}+)*$.
3. Every envelope starts between the last *before* and first *setup*.

   That is, with (1) and (2): $(\mathit{before}+, \mathtt{bgn}\text{-}\mathit{do}+, \mathit{setup}+, \mathit{reset}+, \mathit{after}+)*$.
4. Every envelope ends between the last *reset* and first *after*.

   That is, with (1) and (2): $(\mathit{before}+, \mathit{setup}+, \mathit{reset}+, \mathtt{fin}\text{-}\mathit{do}+, \mathit{after}+)*$.

Given such guarantees (deferred to Appendix C) the rest follows. Specifically, complete demonstration of the correctness of the compilation by:

- Deduce first, since actions are self-mutex, from the last two guarantees, that one and only one instance of the envelope occurs per nearest instances of *before* and *after*. Write: $(\mathit{before}+, \mathtt{bgn}\text{-}\mathit{do}, \mathit{setup}+, \mathit{reset}+, \mathtt{fin}\text{-}\mathit{do}, \mathit{after}+)*$.
- Recalling $(\mathit{setup}+, \hat{\beta}_i+, \mathit{reset}+)*$, further infer that the rest of the intended contents are indeed contained, *i.e.*, for each i:

$$(\mathit{before}+, \mathtt{bgn}\text{-}\mathit{do}, \mathit{setup}+, \hat{\beta}_i+, \mathit{reset}+, \mathtt{fin}\text{-}\mathit{do}, \mathit{after}+) * .$$

(‡) By construction, the duration of the envelope is too short to permit any repeats of its contents (and just long enough to actually fit them all). So, for each i:

$$(\textit{before}+, \texttt{bgn}\text{-}\textit{do}, \textit{setup}, \hat{\beta}_i, \textit{reset}, \texttt{fin}\text{-}\textit{do}, \textit{after}+) * .$$

Eliminating the internal repetitions of *before* and *after* is all that remains.

- It shall be the case that *before* is mutex with only: *do*, *setup*, *reset*, and *after*. Hence nothing mutex will be able to occur between the repeats we are concerned about, by (1) and (2). So by Proposition 3.2 we may assume the repetitions occur immediately; note that immediate repetitions are no-ops. Then prune, and hence deduce that, for each i:

$$(\textit{before}, \texttt{bgn}\text{-}\textit{do}, \textit{setup}, \hat{\beta}_i, \textit{reset}, \texttt{fin}\text{-}\textit{do}, \textit{after}+) * .$$

- Likewise and finally, concerning *after*, deduce that, for each i:

$$(\textit{before}, \texttt{bgn}\text{-}\textit{do}, \textit{setup}, \hat{\beta}_i, \textit{reset}, \texttt{fin}\text{-}\textit{do}, \textit{after}) * .$$

Elaborating on the last, the meaning is as follows. Up to certain equivalence and dominance reductions (*e.g.*, pruning no-ops), with respect to each original action:

- projecting each solution of the compiled problems onto
- all and only those primitives comprising its (intended) compilation
- (indeed) yields just zero or more repetitions of the whole set
- with each such repetition ordered as intended.

Or for short: *(existence)*, *(uniqueness)*, and *(ordering)* hold. Moreover those guarantees were had by way of ensuring the temporal constraints—see (‡) and *(duration)*—hence *correctness* follows.

So we are done but for forcing and the case analysis. Whether a target planner can generate this proof is merely conjecture—meaning future work—and further discussion follows. The details of the book-keeping needed for each case are deferred to Appendix C. □

As far as forcing goes the question (Conjecture 4.13) amounts to whether the target planner can generate the proof. Certainly present implementations are not strong enough.

Regarding the temporal reasoning, note that restricting the number of parts per compound is ill-motivated in the first place should the target planner fail to grok the mechanics of *envelopes* and similar [186]. So either the temporal inference can be automated effectively, or we should just implement direct support for PDDL* to begin with.

For the longer term, pursuing the former is the better research goal; it is also close enough to take seriously. Specifically the supporting, purely classical, landmark-style inference up to the temporal reasoning is well within the near-term grasp of present technique [174]. (Note that generalizing "must occur" to "must occur by time *t*" is entirely natural.) For that matter, there is promising related work to draw from regarding temporal generalizations of problem specific inference.

Indeed, in some sense, any temporal planner is quite relevant: temporal inference is just a basic necessity for performing (nondegenerate) temporal planning in practice. (That is because enumerating all schedules is futile, which heuristics alone cannot remedy, hence the requirement for something resembling temporal

inference.) Perhaps the two best systems to consider in depth regarding forced choices are EUROPA and CPT [12, 202]. That is because both are rooted firmly in (temporal) constraint satisfaction, which is a particularly apt perspective.

EUROPA, for example, uses a most-constrained-decision-first meta-heuristic in selecting the precise form of its search-tree: significantly, "decision" can concern any temporal interval. Particularly, in the extreme case that knowledge of the kind employed in the above proof is available, this flexibility of the search-tree construction permits immediate exploitation of that knowledge in the precise sense of <u>forcing</u>. Note that we may regard such (nonbacktracking) meta-choices over search decisions as a kind of 'automated inference'. As the reader might expect, EUROPA presently lacks any especially sophisticated notion of 'most-constrained'. Then fixing that by integrating the rather more sophisticated work of Bernadini and Smith on temporal invariant analysis into EUROPA (and/or CPT) is a promising research direction [14].

4.5 CONCLUSIONS UPON THE ANALYSIS OF TEMPORAL PLANNING LANGUAGES

What should a *temporal* planner be able to do? Reasoning about *temporarily* available resources is surely important (*cf.*, Figure 4.1):

- fix a fuse by matchlight,
- explore a cave by torchlight,
- navigate a crater on battery power,
- operate an instrument while it retains enough heat to avoid damage,
- overclock a cpu while it remains cool enough to avoid damage,
- achieve happiness in this lifetime,
- hold a meeting while the conference room is available,
- legally cross a traffic intersection,
- *and so on*.

Yet TGP—and a number of so-called temporal planners following suit—cannot even model (let alone solve) such problems! For such a fault to be discovered by an end-user would be embarassing, to say the least. Still, bugs are to be expected. How technically significant is the limitation? This is the mystery we have set out to investigate.

Summary. In this chapter we proved that *Required Concurrency* is a key dividing line in the space of temporal planning languages. On the one side are the temporally simple languages—unable to express problems requiring concurrency. "Simple" is entirely deserved (indeed, "degenerate" is not far off): such languages are unable to express any problem fundamentally beyond the understanding of classical planners (*cf.*, Theorem 4.10). On the other side are the temporally expressive languages—able to express some problems requiring concurrency, and as it turns out all (*cf.*, Theorem 4.12). A single syntactic facility is sufficient (*cf.*, Theorem 4.1):

- If a language demands that every effect of an action intersect over some critical region in time then it cannot express required concurrency.
- Otherwise a language permits causally compound actions and as a result expresses required concurrency.

Consider: It is a straightforward syntactic manipulation to stretch out the effects of every action so as to eliminate causally compound actions [58, 87]. Such abstraction dramatically reduces the decision complexity of planning. So much so that the winners of the temporal planning competitions all do [60, 76, 82, 105, 141]. The meaning of the abstraction: *Assume that everything temporary is insignificant.* Then perhaps the rules were followed [29, 105], but the spirit died long ago. In other words, one possible conclusion is that the competition winners, and every other temporally simple planner, are not *really* temporal planners.

Perspective. Realistically though, the temporal tracks simply never asked for much beyond classical planning abilities. One needs no rule to bar the entry of classical planners—benchmarks requiring concurrency would have served. Which the designers were perfectly well aware of (*e.g.*, examine the short-match domain of the

specification) [71]. So backpedaling: It is true that the difference between temporally simple planning and classical planning is *relatively* small. However, that is a *good* thing. It means we know, or almost know, how to effectively automate that kind of planning. The extension is even theoretically meaningful to a point (*cf.*, Theorem 3.17). More importantly, having planners be willing to integrate more closely with scheduling techniques is just downright practical [17]. Meaning: In practice the most relevant form of temporal planning is temporally *simple* planning. Put yet another way, temporal planning includes classical planning. The *difference* is required concurrency, but an effective temporal planner still needs to be able to do everything a classical planner can do. Then for a balanced conclusion:

- The right theoretical question is how to generalize to required concurrency—without sacrificing the fundamentals from classical planning.

- The right practical question is how to workaround the present theoretical gap: use sequential planning on temporal domains despite the 'impossibility'.

Contributions. The contribution of this chapter in support of that perspective, and towards those ends, is, in abstract, to ground the high-level intuitions concerning expressibility of required concurrency into solid technical results. Specifically we took the perspective of deliberate limitation of language expressiveness. The investigation most deeply concerned the relative feasibility of performing temporal planning by reduction/compilation to sequential/classical planning instead. We found that Required Concurrency neatly divides the problems that classical planners are already strong enough to handle from those that at present lie beyond their reach. The key results are:

Theorem 4.1 Causally Compound Actions characterize expressibility of Required Concurrency.

Theorem 4.10 Temporally Simple languages efficiently reduce to classical planning, augmented by first-fit.

Theorem 4.12 Temporally Expressive languages efficiently reduce to one another —assuming sophisticated landmark analysis.

"Classical Planning" is a moving target. Some day it may grow to encompass these problems we consider presently beyond its reach. That the technical challenge *could* be solved is precisely why it matters.

So now we, finally, turn our attention to the obvious approach. Namely: Perform temporal planning by direct application of a temporal planning algorithm proper. We shall immediately stumble upon a surprise. Even (some of) the planners that *can* model problems requiring concurrency *still* cannot solve them—in any amount of time.

Chapter 5

Worst-Case Analysis of Forward-Chaining Search Algorithms for Temporal Planning

Our primary aims in this chapter are to: (i) further clarify the computational relationship between Sequential Planning and Conservative Temporal Planning, (ii) debunk any form of direct search through the situations of Interleaved Temporal Planning, and (iii) support planning for such problems by temporally-lifted forward -chaining instead. The motivation is simple.

Motivation. Mausam and Weld surprise us with a proof that many state-of-the-art temporal planners are incomplete [147]: contradicting, for example, the Completeness Theorem for SAPA [54]. Indeed:

Competition winners fail to solve the motivating examples of the specification!

To be able to solve benchmarks well and yet fail on toy-size problems is already bizarre; of itself that sort of observation is always cause for concern (*cf.* Sussman's Anomaly [192]). To moreover fail to solve the *motivating* examples for the specific extension of the semantics to temporal planning is downright troubling [71]. (As for decision epoch planners in particular, these technically work on the motivating examples, but fail upon even slight modifications: see Figure 2.13.) To then subsequently take home the title of victor is alarming indeed [60, 76, 82, 105, 141]. There seems to be as many culprits as suspects; for continued drama:

- The benchmarks fail to address the spirit—required concurrency—of the temporal track [42].

- The letter of the formal specification is nigh impossible to implement correctly (which Chapter 2 aims to rectify).
- The empirical results are skewed by the use of domain-dependent search-control knowledge [29, 105].
- Less egregiously, the results are skewed by blind abstraction to conservative temporal planning [56, 58, 87, 105, 201].
- Even several of the good-faith implementations fail to appreciate the full nuances of required concurrency [6, 54, 64, 135].
- Those remaining by and large take the significance of required concurrency too literally, and pay dearly for it [142, 146, 184, 208].

The underlying facts are now reasonably well-known to the relevant experts; this state of affairs is due in no small part to our prior work [42, 43]. Here we delve into demonstrating the extent to which the truth of the algorithm-specific (*i.e.*, the last three) statements may be justified technically—or undermined, *i.e.*, for the contrary, which we shall also be.

Technical Approach and Organization. As always the ultimate aim is to answer:

How can we obtain the strengths, and eliminate the weaknesses, of both the efficient and principled approaches?

Towards such ends within Temporal Planning we conduct a worst-case analysis of abstractions of the forward-chaining fragment of the state-of-the-art implementations. Each kind is defined precisely in its own section, outlined below. For each

we investigate whether guarantees of completeness and systematicity may be given. (*I.e.*, interpret "principled" as complete, and "efficient" as systematic.) Both completeness and systematicity require qualification in the context of temporal planning, due to the infinity of time. We take both with respect to equivalence/dominance reductions.

Recall: A **dominance reduction** is an equivalence relation on plans such that if any member of an equivalence class is a solution, then some canonical representative is as well. An **equivalence reduction** is an equivalence relation on plans such that either every member of each equivalence class is a solution, or none are. Both must also satisfy appropriate computability constraints. For example, it should be reasonably easy to compute the canonical form of arbitrary plans.

Then say a planner is **(strongly) complete** with respect to some reduction when every solution-bearing equivalence class is covered at least once by the underlying search space. It is convenient to have a notion better suited to a kautz1999 perspective. So also say: a planner is **weakly complete** when it is guaranteed to solve every solvable problem. We assume that, in practice, a guarantee of the weaker notion is only ever achieved by way of the stronger notion. To see why, imagine repeatedly strengthening the goal just as a solution is about to be found (so as to rule it, and all variations upon it considered equivalent, out). Formally the connection breaks down, in part because the goal sublanguage is too limited, and also because planners use analysis of goals in order to guide search. However, we know of no implementation outright contradicting the higher level intuition. That is, every weakly complete planner that comes to mind is easily understood as strongly complete. Note that deliberate loss of completeness is not a concern. So for example, we prefer to see LPG-INTERLEAVED [85], a planner employing local search, as

(strongly) complete, because it defines a complete search space prior to applying local search.

A planner is **systematic** with respect to some reduction when every equivalence class is covered at most once by the underlying search space. So systematic planners consider every 'meaningfully distinct possibility' at most once. Deliberate loss of systematicity is hardly troubling. For example, that a sequential portfolio of systematic planners is no longer systematic is no concern. Accidental loss of systematicity is what we are concerned about: when the notion of equivalence fails to be implemented correctly.

Section 5.1 briefly recaps that Chapter 3 already implies a number of effective approaches to Conservative Temporal Planning. Then the remainder concerns primarily our general case: ITP. Section 5.2 critically examines the approaches based upon *a priori* selection of dispatch-times. Section 5.3 proves the theoretical merits of delaying those choices by way of temporal constraint reasoning. Section 5.4 concludes by way of a re-examination of our high-level understanding from a more pragmatic point of view.

5.1 FIRST-FIT CLASSICAL PLANNERS

The winners of the temporal planning competitions are all disconcertingly similar in one regard—they are all incremental modifications to underlying classical planners [29, 56, 60, 76, 82, 84, 105, 141]. Specifically each is an integration of the most -trivial scheduler into a classical planner [17]; by First-Fit Classical Planner (FFC Planner) we mean any such 'generalization' of a classical planner to time. News of their empirical effectiveness is good and bad. On one hand, it is a valuable insight that a number of reasonably compelling temporal planning problems nonetheless lie within the immediate reach of classical planners. On the other, it serves no purpose to dilute empirical evaluations by taking temporal syntactic sugar as computationally meaningful. Towards resolving the dilemma we have throughout drawn a sharp boundary around the middle ground: Conservative Temporal Planning. In Chapter 2 we developed formal definitions supporting plans as sequential compositions of transition functions; in Chapter 3 we reduced those semantics to either the Single-Objective or Multi-Objective Path Problem; in Chapter 4 we identified the key language feature as whether actions are compound or primitive; finally we take a direct algorithmic perspective, wherein it is the size and shape of theoretically pristine search-trees that are of interest, and we shall further confirm that both the nature of plan quality and the structure of actions are key.

To set apart conservative temporal planners from everything more general we examine here a formal characterization of an adequate forward-chaining search-tree for the former; following sections examine symmetrical treatments for interleaved temporal planners. So by contrasting, we may perceive computationally substantive differences between Conservative Temporal Planning and Interleaved Temporal Planning. Then the other side remains: separating Conservative Temporal Plan-

ning from Sequential Planning. More specifically we elaborate upon the relevant demonstration of Chapter 3 (Theorem 3.17): here it is enough to extend a classical planner to multi-objective search. Doing so may be reasonably taken as either trivial or difficult. For triviality we would say only: Merely turning off duplicate state elimination is enough to attain completeness. However, fully attaining theoretical 'perfection'—completeness and systematicity together—is notably more involved. The following formally fills the foundation; thereby we have ground to stand on in closing with further discussion.

Let us begin by formalizing a complete and systematic approach to Sequential Planning:

Definition 5.1 (Brute-Force Search for (Abstractions to) Sequential Planning)**.** A **forward-chaining action-sequence search node** $N = (\mathit{parent}, \mathit{op}; \mathit{State})$ is named by (a reference to) its parent node (*parent*) and the action ($\mathit{op}^{\in \mathit{Actions}}$) labeling the implicit edge from the parent to itself; following are any book-keeping structures worth remembering, given sufficient memory, such as the state (*State*) resulting from executing the action-sequence from the root to the search node. When the parent is null, the search node is the root of its search-tree and represents both the empty plan and some initial state: write along the lines $N_0 \coloneqq (\emptyset, \mathrm{init}; \mathit{State}_{\mathit{Initial}})$. Everything else must satisfy (or be deemed illegal) at least the notion of incrementally executing action-sequences; supposing actions are primitive as in SP and CTP, then with $\mathit{op} = a^{\in \mathit{Primitives}}$:

$$\mathit{State} \coloneqq S'_a(\mathit{State}_{\mathit{parent}}), \tag{5.1}$$

else, supposing compounds may be taken as sequences of primitives, so with $op = (a, \dots) \in \mathit{Primitives}^*$, then automatically and naturally abstract by:

$$\mathit{State} \coloneqq \cdots \circ S'_a(\mathit{State}_{parent}). \tag{5.2}$$

For ITP in particular:

$$\mathit{State} = S'_{\texttt{fin-}\alpha} \circ S'_{\texttt{bgn-}\alpha} \circ S'_{\texttt{all-}\alpha}(\mathit{State}_{parent}).$$

Conflate problems and search-trees: let SP denote the type, *i.e.*, the entire forest.

Classical planners (but not their creators), even those employing different paradigms for search, tend to falsely assume that this forest may, without noteworthy consequence, be severely pruned as follows. The **duplicate-reduction** is the dominance reduction on plans that *(result-equivalence)* holds as equivalent all plans with equal final situations after execution, and *(best-quality)* holds as canonical (tie-broken choices of) those plans maximizing the notion of quality (*i.e.*, canonical means optimal with respect to the equivalence class). (Where situations and states differ, then we write **duplicate-state-elimination** to mean that we keep enforcing the implementation from the classical context; it is unlikely that such will implement some correct reduction.) This is only well-defined, in the first place, if we (falsely) assume that the notion of plan quality will be at least totally ordered; otherwise there may exist incomparable optima of plan quality (*i.e.*, the Pareto dominating set), and so it becomes 'impossible' to make a unique choice (*i.e.*, throwing away any maxima will sacrifice optimality/completeness). For our purposes, because achieving

deadlines is an insufficiently simple notion of plan quality, such is the greatest flaw of the putative reduction.

In order to perfectly operationalize we also need at least Bellman's principle of sub-problem optimality, in other words, for systematicity, "canonical" and "the search-tree" must be together chosen so that: every ancestor of a canonical plan is canonical. Moreover, of course, the notion of canonical must be readily computable. However, as optimality is actually a property of a whole set of plans (in contrast to a plan in isolation, as for, say, executability), that (*i.e.*, implemented systematicity with respect to duplicate-reduction) typically implies computational difficulties too significant to entirely ignore. Specifically, for theory, one must, more or less [48], endure all of: (1) severely restricting the notion of plan quality, (2) exploring the search-tree using specifically A^*, (3) employing a consistent heuristic, and moreover (4) keeping the entirety of the corresponding explored portion of state-space in memory. Such are "the standard caveats", and are indeed significant enough to warrant attention—at some level. For example, rather than just enduring, a realistic implementation runs a portfolio for which only the last-ditch approach literally implements the theory. Everything prior is then free to employ a *transposition table* in contrast to a *closed list*, an *informed* rather than *consistent* heuristic, *et cetera*. Meaning that, at higher levels, it is enough to maintain only dim awareness: issues exist, but so too do resolutions.

So, keeping that in mind, we can take (and properly appreciate) classical planners as complete and systematic, *i.e.*:

Proposition 5.1. SP *may be taken as covering all and only, with the standard caveats, the duplicate-state-reduction of Sequential Planning.*

Proof. The result is standard, *e.g.*, it merely restates Theorem 3.1. □

The significant caveat as far as our purposes go is that *duration-optimality* is, perhaps counter-intuitively, not a 'legitimate' notion of plan quality in the eyes of a classical planner. Intuitively, the total duration of a plan is 'a single objective'—but such is *not* the technical meaning. For a counterexample, consider the subproblem of having a truck transport two packages to distinct depots. Whatever route is chosen, the state after following the route is the same. Which depot was passed through first, though, matters. That is, which continuation will permit the globally fastest scheduling of the overall plan is *dependent* on the choice of route. In contrast, a 'legitimate' notion of quality would display independence.

Of course it is not the place of a planner to dictate the desires of its users—we cannot really allow a planner to define "legitimacy" of notions of plan quality. A planner is certainly free, though, to declare an upper bound on its ability to understand. From a theoretical standpoint, it is best to say that a planner can only understand those notions of quality had by softening already expressible hard constraints. (Penalize any soft constraint to the point where it becomes effectively hard: ergo, for parsimony, insist that literal expression as a hard constraint be possible.) For example, whether optimizing the duration of a plan is said to be understood should amount to no more and no less than guaranteeing completeness for deadlines.

Albeit, from a practical standpoint, it is certainly worthwhile to distinguish optimality from completeness. We recognize that in the form:

Lemma 5.2. *To be complete for* CTP*, with respect to any reduction, is to be duration -optimal and complete, with respect to the same reduction, for Sequential Planning. Conversely, to be complete for* CTP sans *deadlines, with respect to any reduction, is to be possibly duration-suboptimal and complete, with respect to the same reduction, for Sequential Planning. In fact,* CTP sans *deadlines is entirely isomorphic to*

Sequential Planning; in particular duplicate-state-elimination implements a dominance-reduction for either/both. Either way, systematicity goes towards ensuring that minimal search nodes are considered.

Proof. See Theorem 3.4 and Proposition 3.5 for the lemma up to systematicity. The discussion preceding Proposition 5.1 clarifies the relationship of systematicity to optimality. □

So deadlines are the key feature distinguishing Sequential Planning from Conservative Temporal Planning, and a specific substantive difference is whether duplicate-state-elimination need be, ultimately, disabled in order to, eventually, find the very best of schedules. Naturally we would substitute at least the minimal reasonable reduction of Conservative Temporal Planning: the left-shifted-reduction (Theorem 3.4). That at least should take us down to all action-sequences (rather than action-schedules). For contrast, the naïve, and grossly infeasible, perspective:

Definition 5.2 (Brute Force for Action-Schedules)**.** A **forward-chaining action-schedule search node** $N = (parent, op, dispatch; State, Vault)$ is named by (a reference to) its parent node (*parent*), the action $op^{\in Actions}$ and time $dispatch^{\in \mathbb{Q}}$ labeling the implicit edge from the parent to itself; following is the situation resulting from executing the action-schedule from the root to the search node.

When the parent is null, the search node is the root of its search-tree. Such represent both the empty plan and some initial situation. Write along the lines $N_0 \coloneqq (\emptyset, \text{init}, t_0; State_{Initial}, Vault_{Initial})$.

Everything else must satisfy (or be deemed illegal) at least the notion of incrementally executing action-schedules; supposing actions are primitive as in CTP,

then with $op = a^{\in Primitives}$ and $dispatch = t$:

$$State \coloneqq S'_a(State_{parent}), \tag{5.3}$$

$$Vault \coloneqq V'_{a,t}(Vault_{parent}), \tag{5.4}$$

else, supposing compounds may be taken as sequences of relative dispatches, so with $op = ((a, s), \dots) \in (Primitives \times \mathbb{Q})^*$ and $dispatch = t$, then automatically and naturally abstract by:

$$State \coloneqq \cdots \circ S'_a(State_{parent}), \tag{5.5}$$

$$Vault \coloneqq \cdots \circ V'_{a,t+s}(Vault_{parent}). \tag{5.6}$$

For ITP in particular, with $op = \alpha$, $dispatch = t$, and the relative start-time of the end-part denoted by $x = dur_{\texttt{all-}\alpha} - dur_{\texttt{fin-}\alpha}$:

$$State = S'_{\texttt{fin-}\alpha} \circ S'_{\texttt{bgn-}\alpha} \circ S'_{\texttt{all-}\alpha}(State_{parent}),$$

$$Vault = V'_{\texttt{fin-}\alpha,t+x} \circ V'_{\texttt{bgn-}\alpha,t} \circ V'_{\texttt{all-}\alpha,t}(Vault_{parent}).$$

Then consider forcing the dispatch-times so as to reduce to action-sequences. That is, consider (and contrast with above) loosely integrating First-Fit into a classical planner:

Definition 5.3 (The FFC Planning Search Forest)**.** A **first-fit forward-chaining action-sequence search node** $(parent, op; State, Vault)$ is named by (a reference to) its parent node $(parent)$ and the action $(op^{\in Actions})$ labeling the implicit edge from the parent to itself; following is the situation, $(State, Vault)$, resulting from first greedily scheduling and subsequently executing the action-sequence from the

root to the search node. When the parent is null, the search node is the root of its search-tree and represents both the empty plan and some initial situation: write along the lines $N_0 \coloneqq (\emptyset, \text{init}; \mathit{State}_{\mathit{Initial}}, \mathit{Vault}_{\mathit{Initial}})$. Everything else must satisfy (or be deemed illegal) at least the notion of incrementally scheduling and executing; supposing actions are primitive as in CTP, then with $op = a^{\in \mathit{Primitives}}$:

$$\mathit{State} \coloneqq S'_a(\mathit{State}_{\mathit{parent}}), \tag{5.7}$$

$$\mathit{Vault} \coloneqq V'_a(\mathit{Vault}_{\mathit{parent}}) = V'_{a,\mathrm{EST}_a(\mathit{Vault}_{\mathit{parent}})}(\mathit{Vault}_{\mathit{parent}}), \tag{5.8}$$

else, supposing compounds may be taken as sequences of relative dispatches, so with $op = ((a, s), \dots) \in (\mathit{Primitives} \times \mathbb{Q})^*$, then greedily select t so that at least the computation of resulting vault succeeds:

$$\mathit{State} \coloneqq \cdots \circ S'_a(\mathit{State}_{\mathit{parent}}), \tag{5.9}$$

$$\mathit{Vault} \coloneqq \operatorname{argmin}_V \left\{ t \mid V = \cdots \circ V'_{a,t+s}(\mathit{Vault}_{\mathit{parent}}) \right\}. \tag{5.10}$$

For ITP in particular, with $op = \alpha^{\in \mathit{Compounds}}$ and the relative start-time of the end-part denoted by $x = \mathit{dur}_{\texttt{all-}\alpha} - \mathit{dur}_{\texttt{fin-}\alpha}$:

$$\begin{aligned} \mathit{State} &= S'_{\texttt{fin-}\alpha} \circ S'_{\texttt{bgn-}\alpha} \circ S'_{\texttt{all-}\alpha}(\mathit{State}_{\mathit{parent}}), \\ \mathit{Vault} &= \operatorname{argmin}_V \left\{ t \mid V = V'_{\texttt{fin-}\alpha,t+x} \circ V'_{\texttt{bgn-}\alpha,t} \circ V'_{\texttt{all-}\alpha,t}(\mathit{Vault}_{\mathit{parent}}) \right\}. \end{aligned} \tag{5.11}$$

Let FFC (without duplicate-state-elimination) denote, in whatever context, the entire forest; in practice, the more likely assumption is that duplicate-state-elimination remains enabled.

So we have many, more or less reasonable, baseline approaches to both optimal and satisficing versions of Conservative Temporal Planning:

Theorem 5.3. *Any* FFC *planner disabling duplicate-state-elimination is, or may be taken to be, complete and systematic for the left-shifted-reduction of Conservative Temporal Planning.*

Any FFC *planner enforcing duplicate-state-elimination is, or may be taken to be, complete and systematic for the duplicate-reduction of Conservative Temporal Planning* sans *deadlines (which is isomorphic to Sequential Planning); however, such planners are duration-suboptimal.*

Proof. The first result 'merely' recasts Theorem 3.4 in an explicitly algorithmic form. Technically the difference from Theorem 3.4 is that we need to check that pruning non-canonical nodes preserves all canonical nodes, said pruning is readily computable, and the resulting search-forest is isomorphic to FFC. So observe: (i) Every prefix of a left-shifted schedule is left-shifted, (ii) First-Fit is fast, and (iii) the search-forest FFC just implements the left-shifted-reduction by its classes (action-sequences) rather than by its canonical representatives (left-shifted action-schedules).

The second result is by Lemma 5.2 and Proposition 5.1. □

A significant limitation is that the left-shifted-reduction is merely the weakest reasonable reduction to apply. Unfortunately, as far as achieving tight deadlines go, duplicate-state-elimination goes too far. It would be nice to have a rather more reasonable, *i.e.*, far stronger, substitute for its absence than the left-shifted-reduction. For example, it is unfortunate that Theorem 5.3 leaves us with temporal planners that perceive $k!$ ways of simultaneously loading k packages onto as many trucks;

supposing it can be done cheaply enough, clearly it would be better to have just 1 search node for simultaneously loading k packages.

Define the **deordered-left-shifted-reduction** by further applying the equivalence-reduction from Theorem 3.14 (the order of the reductions is irrelevant). It is possible to pick canonical representatives in such a way as to permit pruning away all non-canonical plans. The details are not substantially different from the similar, later, implementation of the deordered-slackless-reduction for TEMPO: actions here are primitives/effects there, and left-shifted here is slackless there. In particular the pruning rule remains the same: (i) compute the rank of each plan step in the mutex-order of the plan, (ii) insist that ranks increase monotonically, and (iii) break ties any which way, *e.g.*, alphabetically. Leaving the details, then, to the reader:

Theorem 5.4. FFC *planners may be made complete and systematic for the deordered-left-shifted-reduction of Conservative Temporal Planning.*

Proof Strategy. The key tools are Theorems 3.4, 3.14, and 5.16; it suffices to simplify the proof of the last from ITP to CTP. For such mapping between CTP and ITP it may be helpful to refer to the proof of Theorem 3.21. □

5.1.1 Discussion

The final theorem is extremely powerful—and seems to be, as of yet, unimplemented—*i.e.*, we conjecture here a relatively easy improvement upon the state-of-the-art. First among its strengths, shared by Theorem 5.3, is simply the clear path to leveraging the state-of-the-art in classical planning.

- Importantly, it does not suffer from the limitations of its more general compatriot, Theorem 5.16. Indeed, here is exactly what there aspires to automatically simplify to: action-sequences in preference to effect-sequences.

- Locally we can note that enforcing deordering *via* the suggested pruning rule has much of the power—but none of the drawbacks—of duplicate state elimination. Specifically, it is independent of quality metric, search-order, and thus also heuristic; furthermore it is ludicrously cheap (*e.g.*, implementing the pruning rule does not require exponentially large memory). Yet such planners nonetheless think about (for example) loading packages in the right way: as sets, not sequences.

- To be sure, deordering leaves undisturbed any qualitatively distinct methods for bringing about the same state of affairs (which is how it is weaker than duplicate state elimination). Consider though that, especially for temporal planning, it is well-motivated to care about the journey as much as the destination. So perhaps even the weakness is a strength.

As to our larger purpose, contrasting Theorems 5.4 and 5.16 precisely captures the computationally meaningful difference between Conservative Temporal Planning and Interleaved Temporal Planning: compounds versus primitives. That is, so long as actions are useful exclusively for their long-term effects (so are effectively primitive), then we can expect Conservative Temporal Planners in general (and FFC planners in particular) to enjoy a substantial advantage. Specifically we may predict by comparing (executable, deordered) action-sequences to (executable, deordered) effect-sequences: the latter are exponentially greater in number. On the other side, if even one short-term aspect of executing an action should be crucial,

then such planners fail by definition—specifically, Definition 5.3 dictates that all actions be treated as primitives regardless of the truth of the matter. Experiment, *i.e.*, the planning competition, corroborates: Conservative Temporal Planners perform best, presuming causally sequential problems, else utter failure.

Delving deeper we find additional relevant advantage. Namely, when optimizing the duration of plans is purely optional, then Theorem 5.3 comes into full force. Abstracting: Sequential Planning and Conservative Temporal Planning differ meaningfully all and only to the extent that duration-optimality is meaningful. Then observe that the benchmarks indeed lack deadlines, and the temporal planning tracks have never demanded any form of optimality. There can be little mystery left to who wins and why. Albeit, the theory does permit better than the baselines—sprinkle temporal syntactic sugar on a classical planner—but the baselines are moving targets. Then so long as classical planning research continues to yield significant performance improvements, provided also that temporal benchmarks maintain their present character (sugar-coated classical benchmarks), we can expect temporal planning state-of-the-art to remain with the baselines.

The insight is neither good nor bad: just useful (*e.g.*, towards design of empirical evaluation). Thus we conclude our examination of abstract algorithms for Conservative Temporal Planning: the scope stops at forward-chaining, complete, and systematic search-trees, for which little else may be said without saying a great deal more. Next we shall treat Interleaved Temporal Planning in reasonably symmetric fashion, as promised; attaining the desired properties is though, as we shall see, rather more challenging.

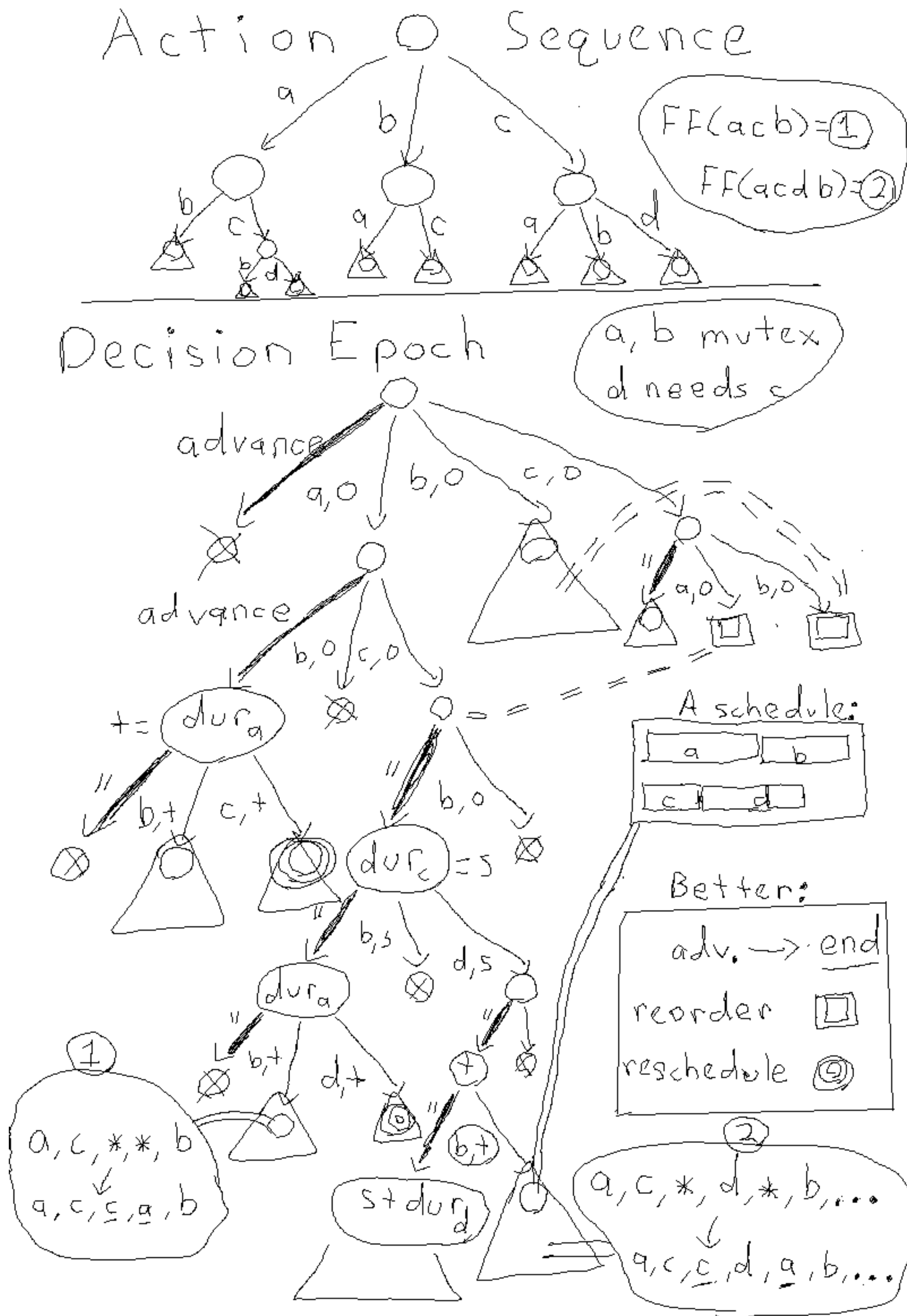

Figure 5.1: An abstract look at FFC versus DE search-trees. Interesting improvements such as writing end-parts rather than advances are possible. See Section 5.2.1.

5.2 DECISION EPOCH PLANNERS

Many effective temporal planners (as judged by the IPCs) are so-called 'state-space' temporal planners, where "state" is meant in the sense of situation. Those that aim at supporting required concurrency face, then, an infinity of choice. In this section we look in detail at the *decision epoch* solution to the infinity, exposing disconcerting weaknesses: incompleteness and nonsystematicity. SAPA [54], TLPLAN [6], TP4 [100], and HSP$^*_a$ [96], among others, are all examples of Decision Epoch Planners (DE Planners) [6].

Rather than consider each in isolation, we abstract the essential element of their search-trees. The essential element is the definition of successor nodes, *i.e.*, the definition of generating children. There are two kinds of children: *fattening* and *advancing*. The former are vanilla: these are just normal dispatches and follow normal rules for transitioning between situations. The special children advance the **decision epoch** to some selected, later, time; search nodes carry that along, using it as the only permitted dispatch-time for normal children. The particular (and only) selection rule that is widely implemented is referred to as **advance-time**: advance time to just after the soonest-to-finish pending effect indeed finishes. So, unqualified, assume the advance-time rule for selecting decision epochs.

5.2.1 Discussion of Figure 5.1

The relationship between a classical planner and a decision epoch planner is that the latter has an additional branch at just about every search node. (The exception is when no action is in the midst of execution.) An extra branch is a very painful thing; there are many more opportunities to waste computation. Especially this particular branching is extremely wasteful.

What is happening is that scheduling decisions are being made eagerly. So in particular note the relationship concerning the search vertices labeled (1) and (2). These are the same schedules found in very different ways. As far as these sort of scheduless go: the classical planner has the upper hand by far. That is because making extra search decisions (pushing disjunctions into the tree) is an *exponential cost*. In contrast, running First-Fit is linear.

That said, the DE planner is considering a larger space for a reason. There could be sneaky ways of exploiting concurrency that an FFC planner simply cannot find. Then perhaps we could tolerate higher computational cost.

If so, then it is interesting to note the considerably greater pruning opportunities afforded by exploiting the reordering and resheduling insights. Specifically the marked subtrees, on reflection, demonstrate reasonably well that, as we generalize, the opportunity for the theorems of Chapter 3 to really prove their significance increases rapidly.

There is one last subtle point being made by the Figure. Such is better appreciated upon returning from consideration of temporally lifting (TEMPO). For reference, note that it is equivalent to (a) call the extra branches by the common name "advance-time", or to (b) name them by the specific action whose end-part is there carried out.

Until then, our goal is to demonstrate that DE planners are fundamentally incomplete and nonsystematic.

5.2.2 Setup

Unfortunately our formal account will encounter difficulty. Let us first sketch slightly deeper the traditional approach to formalizing. The traditional treatment

labels vertices by: (3) the decision epoch, (4) the (classical) state, (5) some structure for maintaining invariants (for us, locks/vaults), and (6) some structure, "the event queue", for tracking the pending effects (for us, obligations/debts). The edges between such are labeled by either an action name, or by the special search-only operation "advance-time". It is common, particularly for implementations, to fold together the edges and vertices of search-trees into search nodes; a search node is just a vertex extended by: (1) a reference to its parent, and (2) the label of the now implicit edge.

Our notation is not crafted towards, and is slightly ill-suited for, elegantly describing this style of algorithm in the traditional manner. (That is largely because an event queue and a debt are not quite the same.) For the sake of cross-referencing we will eventually give a thorough formal account, *i.e.*, comparable to those of the other sections.

As far as our direct aims are concerned though, we do not require such heavy -duty support. Demonstrating incompleteness and nonsystematicity is far simpler than demonstrating their inverses. So instead we rely on the correctness of a relatively lightweight definition and proposition concerning decision epoch planners. That is, for our purposes, we need only rely on the correctness of:

Definition 5.4. Call a (*time-sorted*) action-schedule $X = (\alpha, t)_{[n]}$ **decision-epoch** when *(epoch-explainable)* each action start-time is the finish-time of an effect of an action earlier in the dispatch-order, and *(quasi-executable)* it is executable but for some suffix of only end-parts (as follows). That is, more precisely, it should be the case that every induced effect-schedule $Y = (a, s)_{[3n]}$ of X, *(suffix-canonical)* with a maximal m-suffix consisting of only end-parts (maximize m such that $Y\restriction_{[3n-m+1,3n]}$

‘is an event queue’), satisfies that *(prefix-executable)* its prefix $Y\upharpoonright_{[3n-m]}$ is executable, and optionally *(suffix-plausible)* its entire vault-sequence exists.

So for notation, X is decision-epoch precisely when the following holds; take $a_0 = (\alpha_0, \mathtt{all})$ as a dummy effect standing for the initial situation (and $s_0 + dur_{a_0} = t_0 = 0$). The/every time-sorted execution $(S, V, D)_{[0,3n-m]}$ holding out a maximal suffix of m end-parts, of appropriate effect-schedules $Y = (a, s)_{[3n]}$ induced from X, must exist. Moreover, for every action index $i \in [n]$, there must exist an earlier effect $a_k = (\alpha_j, \iota)$, *i.e.*, where $j \in [0, i-1]$ and $k \in [0, 3j]$, such that:

$$t_{i-1} \leq t_i, \tag{5.12}$$

$$t_i = s_k + dur_{a_k}. \tag{5.13}$$

Proposition 5.5. *The search space of a Decision Epoch Planner is all and only the decision-epoch action-schedules.*

Proof. For the only direction: By definition, the decision epoch always advances to the (soonest) finish-time of one of the effects of one of the actions selected prior to advancing time. So epoch-explainable is necessary, as is time-sorted. During the advance, each pending end-part passed over is applied. We assume, but it is of little consequence, that the order of application is time-sorted. If one should fail to be executable, then the entire attempt to advance time fails. (End-parts that coincidentally begin at precisely the time advanced to may or may not be applied, for our purposes it is simpler to suppose not; one way or the other only changes which advance-time will check the constraints.) Fattening choices check executability of the beginning of the action (all-part + start-part), but, do not insist that the end-part be executable quite yet; later fattening choices might be required in order to succeed

at advancing time past its end-part. (Hence the complications of the definition towards holding out a suffix of end-parts.) So quasi-executable is necessary. In short the only direction is straightforward.

Recall that advance-time always advances to the soonest finish-time: none are skipped. So read through any given time-sorted epoch-explainable quasi-executable action-schedule, re-interpreting it as a sequence of fattening and advancing choices. Specifically insert as many advance-times as necessary between the dispatches of actions in order to find the corresponding search path of a DE planner. By the assumptions of epoch-explainable and time-sorted, some number of inserted advance -times shall indeed suffice to (precisely) advance the epoch to each dispatch-time. By the assumption of quasi-executability the whole process will be executable up to and through the beginning of the last action. That establishes that a Decision Epoch Planner indeed examines the possibility. In other words the all direction holds. □

The key property is of course just the selection of decision epochs, that is, the rule for advancing time. In order for decision epoch planners to branch over action selection at any given time point, time, *i.e.*, the decision epoch, must have advanced to that point. The advance-time rule always advances to the finish-times of effects. So those are the times at which an action could begin. The significance of which is the inability to start actions at any *other* time.

5.2.3 DE Planners are Incomplete and Nonsystematic

Consider first that starting actions 'in the middle of nowhere' is easily made necessary: Figure 2.13 depicts an example. So incompleteness is clear: How incomplete? In *general*, decision epoch planners are incomplete.

Theorem 5.6. *Decision epoch planners are incomplete for any syntactically super -classical language further permitting required concurrency.*

Proof. It suffices to show incompleteness for the four minimal syntactically super-classical temporally expressive languages (*cf.*, Lemma 4.14); for each it suffices to demonstrate a single solvable problem that decision epoch planners cannot solve.

Specific durations are not important; say endpoint-effects are all duration 1, and actions have much larger durations. Write $A = (\gamma; \alpha; \beta)$ to mean that the all-part of A has effect γ, *etc.*, *i.e.*, omit the durations.

In each case we force action A to begin 'in the middle of nowhere'. Which suffices (by Proposition 5.5 and the definition of epoch-explainable).

Case. $\mathtt{L}(\emptyset;\ \mathtt{pre};\ \mathtt{eff})$. Consider $A = (\emptyset;\ I_A = \mathtt{True};\ G_A := \mathtt{True}, I_A := \mathtt{False})$, and $B = (\emptyset;\ I_B = \mathtt{True};\ G_B := \mathtt{True}, I_B := \mathtt{False}, G_A := \mathtt{False}, I_A := \mathtt{False})$. Both A and B must occur, to meet the goal $G_A = \mathtt{True}$ and $G_B = \mathtt{True}$. Both can only occur once due to each deleting its own precondition (which are initially true). So there are just six possibilities. It is easily checked that only one is potentially a solution: initiate B, initiate A, terminate B, terminate A. As long as the duration of A is even a little bit shorter than that of B, then A must begin when nothing else is happening.

Case. $\mathtt{L}(\emptyset;\ \mathtt{eff};\ \mathtt{pre})$. Consider $A = (\emptyset;\ G_A := \mathtt{True};\ G_B = \mathtt{True}, G_C = \mathtt{True})$, $B = (\emptyset;\ G_B := \mathtt{True};\ G_A = \mathtt{True}, P = \mathtt{True})$, and $C = (\emptyset;\ G_C := \mathtt{True}, P := \mathtt{False};\ \emptyset)$. All propositions start false but for P, which starts true. Each action must occur, to meet the goal $G_A = \mathtt{True}$ and $G_B = \mathtt{True}$ and $G_C = \mathtt{True}$. Each proposition is monotonic: it is causally fruitless to repeat any action/effect, so each

occurs more or less uniquely. To be precise though, by “the C” we mean the *first* instance of action C.

The A ends after both the B and the C begin: $\{\texttt{fin-}A > \texttt{bgn-}B, \texttt{fin-}A > \texttt{bgn-}C\}$ is a ‘subset’ of the mutex-order ($\prec_{\text{mutex-}X}$) of each solution X. The B ends after the A begins, but before the C begins: $\texttt{bgn-}C > \texttt{fin-}B > \texttt{bgn-}A$ *et cetera*. Then the A, due to the C, must wrap the end of the B: $\texttt{fin-}A > \texttt{bgn-}C > \texttt{fin-}B > \texttt{bgn-}A$ follows, so simplifying, $\texttt{bgn-}A < \texttt{fin-}B < \texttt{fin-}A$. Therefore, as long as the duration of A is slighty shorter than the duration of B, the A will start somewhat later than the B starts: for all solutions X, $\min\{s \mid (\texttt{bgn-}A, s) \in \text{Rng}(X)\} - \min\{t \mid (\texttt{bgn-}B, t) \in \text{Rng}(X)\} > 0$. We already know that the first C must wait until the first B ends: $\texttt{bgn-}C > \texttt{fin-}B$. Then, as also actions may not be concurrent with themselves and by the minimality of the first dispatches of A, B, and C: no action, in particular C, may start or end between the start of B and the start of A (if the goal is to ever be achieved). So no actions are available to gain access (*i.e.*, in the form of a decision epoch) to when the A must start. In short, A must begin when nothing else is happening.

Case. L(pre; eff; eff). Conceptually the counter-example is just: $A = (\emptyset;\ G_1 := \texttt{False};\ G_2 := \texttt{True})$, and $B = (\emptyset;\ \emptyset;\ G_1 := \texttt{True}, G_2 := \texttt{False})$. However, DE planners *can* solve that problem, by throwing in junk repeats of A in order to gain access to an appropriate time. So consider: $A = (\emptyset;\ G_1 := \texttt{False};\ G_2 := \texttt{True}, P := \texttt{False})$, and $B = (P = \texttt{True};\ \emptyset;\ G_1 := \texttt{True}, G_2 := \texttt{False})$. Both A and B must occur, to meet the goal $G_A = \texttt{True}$ and $G_B = \texttt{True}$. After the first A ends no instance of B could begin, for that matter, no instance of B could cross that time point; any number of instances of B could come earlier, and at least one

does (but of course none are concurrent with themselves). In particular between the beginning of the last B and the end of the first A no other effects occur. As A must begin before that last B ends, in order to achieve the goal, as long as its duration is slightly shorter than that of B, then A must begin when nothing else is happening.

Case. $\mathtt{L(eff;\ pre;\ pre)}$. Consider: $A = (G_A \coloneqq \mathtt{True};\ G_C = 0;\ G_C = 2)$, $B = (G_B \coloneqq \mathtt{True};\ G_C = 0;\ G_C = 1)$, and $C = (G_C \coloneqq G_C + 1;\ \emptyset;\ \emptyset)$. The goal is $G_A = \mathtt{True}$ and $G_B = \mathtt{True}$ and $G_C = 2$, with initial values of false and zero. The fluents are all monotonic, in particular, action C occurs precisely twice. Every A starts before and ends after both instances of C: so A is unique. Every B starts before and ends after the first C: so B is unique. Then it is clear that the only solution looks like: initiate B, initiate A, do C, terminate B, do C, terminate A. In particular no effects can intervene between the beginning of the B and the A. So have C be short duration, A be medium duration, and B be long duration. Then A must begin when nothing else is happening.

□

In fact, the incompleteness runs deeper, even, than required concurrency; somewhat loosely:

Theorem 5.7. *Decision epoch planners are duration-optimal if and only if all effects of each action begin simultaneously.*

Corollary 5.8. *So with deadlines,* DE *planners are complete iff delayed effects are forbidden.*

Corollary 5.9. DE *planners are complete (and duration-optimal) for Conservative Temporal Planning.*

Proof. If always every effect begins at the start-time, then it is completeness-preserving to consider just the classically-sorted effect-schedules (by Theorem 4.1). Slackless schedulings of which, by definition, start at least one effect of every action at the finish-time of some other effect. Moreover, by Theorem 4.10, the dependencies will be linearly ordered: at least one effect of every action will begin at the finish-time of an effect of an action *earlier in dispatch-order*. By hypothesis, where one effect starts, all do. So it is guaranteed: Every (effect of each) action starts at the finish-time of some effect of some action earlier in dispatch-order. Which is the definition of epoch-explainable. So by Proposition 5.5, the if direction is shown.

For a contrapositive, suppose a single delayed effect, negatively interacting with anything else in the problem domain. Force that particular effect to occur along a critical path of every solution by appropriate selection of initial situation and goal; if necessary expand the problem domain (*i.e.*, perform surgery as in the proof above) on the off-chance that somehow such selection is impossible. DE planners do not take such negative interactions into account when dispatching the action containing the delayed effect (discussion follows the proof); the dispatch-time must be the epoch, and the epoch-advancement is independent of subsequent fattening decisions. So if a DE planner does find a solution, it would be sheer coincidence (which we may surely eliminate with yet further surgery) for the delayed effect to end up scheduled slacklessly. Then assume slack. Slack along a critical path means duration-suboptimal. Hence the only if direction holds with respect to the mentioned caveats.

The corollaries are straightforward. □

The significance of required concurrency in regards to the above two theorems lies in defeating the *lookahead rebuttal* to the point that DE planners totally ig-

nore delayed effects in making scheduling decisions. See the discussion of "DEP+" in [43]. Suppose, by looking ahead, one can guarantee that just the right set of times are considered for dispatching an action. (Of course, if the lookahead is unbounded, then the 'lookahead' is in fact responsible for planning itself.) If the guarantee is had by bounded lookahead, then required concurrency is somehow not expressible in anything remotely resembling a general fashion: study Theorem 4.12 and its proof. So consider the extreme case where required concurrency is entirely forbidden. Then, by Theorem 4.10, it is superior to use First-Fit to lazily schedule action -sequences whenever the temporal information would be useful to have. In other words, in the complete absence of required concurrency, it is inferior to be trying to select the dispatch-times up-front (whether by use of a decision epoch, or any other means). Let us take a leap of faith to where required concurrency is no longer illegal, but still onerously restricted. So: If we can guarantee that bounded lookahead allows us to nail down dispatch-times with impunity, then by analogy we 'ought' to be able to guarantee that it would be better still to entirely forego selecting dispatch -times up-front. (In particular attempting to convert any such bounded lookahead into some form of on-demand scheduling seems quite promising.) Then in short, in the direction of temporally simple planning, to 'fix' Decision Epoch Planning: *Get rid of the epoch.*

To round out the negative results:

Theorem 5.10. DE *planners fail to be systematic with respect to even just the weakest reasonable reductions.*

Proof. The only interesting reduction that DE planners half-heartedly attempt to implement is something like the left-shifting or slackless dominance-reductions. Neither is achieved: dream up any action that is completely independent of every-

thing. It will be tried at every epoch. For that matter, it will also be tried after every fattening choice. Clearly it would be far superior to bound the new size of the search space by a factor of 2 in order to address the additional action (*i.e.*, decide only whether the action will or will not be in the plan: the when is irrelevant); allowing the branching factor to be 1 greater everywhere is grossly inefficient, *i.e.*, exponentially worse. □

It is, of course, unfair to claim that DE planners are nonsystematic without spending more time trying to come up with working pruning rules. (Which plays into why the heavy-duty formal treatment, next, is interesting.) In fact it *is* possible to, say, prune the DE space down so that no two schedules both dominated by the same slackless schedule are kept. The way to do that is just: Build the Simple Temporal Networks and solve them (*cf.* Theorem 3.18, and especially the next section on TEMPO). Having done so, however, it seems unnatural and counterproductive to continue to think in terms of decision epochs. The meaning of which is that, pursuing the details, we shall find that: 'all roads lead to TEMPO'. So in the direction of greater temporal expressiveness, the epoch turns out to be, again, a handicap rather than a benefit; from any angle it seems that to 'fix' Decision Epoch Planning is just to end up undermining its identity.

5.2.4 Formal Definition of Decision Epoch Planners as Search through Temporal Situations

Here we setup a heavy-duty formal treatment of the search-tree of decision epoch planners. The technical perspective is helpful towards much deeper analysis than that just pursued. Especially analysis towards 'degrees of truth' to properties as fundamental as soundness, completeness, systematicity, and optimality may benefit

from the perspective. Said another way, if one were to seriously consider implementation, despite the preceding theory, then: details are useful. For example, based on the details, we conjecture that the following are empirically true of DE planners:

- Alphabetizing fattening choices is better.
- Forcing actions to begin at earliest start-times is better.[1]
- Radically (*i.e.*, DE is a poor label after the change): Using a separate 'epoch' per fluent is better.

  Among other more interesting consequences, doing so permits enforcing deadlines without compiling the deadlines into pending effects.

We could go much deeper than mere conjecture here, incidentally. Empirically salvaging DE planning is, however, beyond our scope. Within scope: it is interesting —because, roughly, TEMPO just temporally lifts the following—to reference the following when considering the formal treatment of TEMPO.

Setup. Our purpose is to demonstrate that the preceding relatively lightweight formalization is legitimate. The proof is mathematically legitimate: the question is whether it really describes what it purports to describe. Does the theory actually apply to planners such as SAPA? So verifying that is the exercise we carry out next. Those not already familiar with TLPLAN [6], SAPA [54], or similar are invited to skip the exercise. In any event:

Our approach is to pick out all the walks through situations that are the executions that decision epoch planners consider. Then each all-part+start-part pair will

[1]Collapse the all-part + start-part in order to compute an EST, and demand that be equal to the current epoch.

correspond to a *fattening* choice; and likewise each end-part will correspond to an *advancing* choice. Computationally we ought to, and do, further demonstrate that the manner of picking out the walks may be implemented in an incremental manner (*i.e.*, roughly constant-time per child).

First just the naïve search-tree for ITP:

Definition 5.5. A **forward-chaining effect-schedule search node**:

$$N = (parent, op, t; State, Vault, Debt),$$

is named by (a reference to) its parent node P and the effect-dispatch (op, t) labeling the implicit edge from its parent; following are derived structures, namely, the situation resulting from executing the effect-schedule connecting from the root.

When the parent is null, then the search node is a root of the search-space. Such represent both the empty plan and some initial situation. Write along the lines:

$$N_0 \coloneqq (\emptyset, \text{init}, t_0; State_{Initial}, Vault_{Initial}, Debt_{Initial}).$$

Non-roots must incrementally satisfy executability, with $a = op$:

$$State \coloneqq S'_a(State_{parent}), \tag{5.14}$$

$$Vault \coloneqq V'_{a,t}(Vault_{parent}), \quad \text{and} \tag{5.15}$$

$$Debt \coloneqq D'_{a,t}(Debt_{parent}). \tag{5.16}$$

Every induced effect-schedule of a decision-epoch action-schedule is one way that a DE planner might navigate through the naïve search-space. Any specific

DE planner, however, presumably picks just one of the many behavior-equivalent possibilities. So for a canonical choice:

Definition 5.6. A **decision-epoch effect-schedule** is a canonical choice of an effect -schedule induced from a decision-epoch action-schedule. The induced schedules are time-sorted already, so only tie-breaking for simultaneously-starting effect-dispatches need be specified. Tie-break by:

Fatten First Order all non-end-parts first.

Pair All+Start Among the non-end-parts, preserve the dispatch-order of the action-schedule.

Alphabetize Ends Sort the end-parts by the corresponding action names.

Observe that the property is monotonic: every prefix of a decision-epoch effect -schedule is decision-epoch. So pruning leaves *all* (and only, of course) of the decision-epoch effect-schedules intact. Then:

Definition 5.7. A **decision-epoch search node** is a forward-chaining effect-schedule search node such that the concerned effect-schedule is decision-epoch. For an interesting abuse, let DE denote the type, which constitutes an infinite forest on every legal decision-epoch search node: one tree per interleaved temporal planning problem.

Each such search-tree is equivalent, as far as search goes, to the traditional description of how a DE planner works. That is, the definitions are correct—the implied isomorphism is true:

Lemma 5.11 (Heavyweight Replacement for Proposition 5.5)**.** DE *planners effectively search through all and only* DE*. Moreover, and crucially, decision-epoch search nodes may be identified incrementally.*

Proof. By Theorem 3.21, the choice of tie-breaking among induced effect-schedules is irrelevant as far as the final result is concerned; the executions all end in the same situation. So in a sense the details of a canonical choice ought to be irrelevant. However, decision-epoch action-schedules (for which "all and only" already holds by Proposition 5.5) specifically request executability only up to holding out a maximal suffix of end-parts. So it is important to order all non-end-parts first. Furthermore, pruning a fattening choice would happen immediately if either the all-part or start-part were to fail; so it is important to evaluate the start-parts immediately after their corresponding all-parts. The tie-breaking among end-parts, however, is arbitrary so long as it does indeed force a total-order; whatever rule is used is equivalent to implementing the details of advance-time. So alphabetizing them is fine, as would be enforcing any other total-order.

In other words the tie-breaking defined clearly establishes a bijection between decision-epoch action-schedules and decision-epoch effect-schedules, moreover, the mapping is computable, computably invertible, and search-topology-preserving as just outlined. Therefore the isomorphism is shown.

However, we have 'cheated' by relying upon the definition of a decision-epoch action-schedule. To 'really' implement DE as a pruning of the naïve search-space we must demonstrate that the pruning can be done without mapping back to the perspective from action-schedules. Otherwise there is no point to taking the effect-schedule perspective.

So in other words, for the moreover, we must *inductively* identify those forward -chaining search nodes that are moreover decision-epoch. For the base case there is nothing to show: Every root is trivially a decision-epoch search node.

Then assume a decision-epoch search node $P = (\cdot, a, t, S, V, D)$ with forward -chaining child $C = (P, a', t', S', V', D')$. It suffices, expanding through the definitions of a decision-epoch effect-schedule and decision-epoch action-schedule in turn, to incrementally check:

1. Time-sorted.
2. Epoch-explainable.
3. (Quasi-)Executability.
4. Tie-breaking:
    a) Fatten First.
    b) Pair all-parts with start-parts.
    c) Alphabetize end-parts.

Consider the 'event queue'. Define the *soonest start-time* of the pending effects with respect to a debt X by:

$$s^*_X := \min_{\alpha} X(\alpha, \mathtt{fin}).$$

From there pick out the *alphabetically-least-soonest* pending effect by:

$$a^*_X := (\min \{\alpha \mid X(\alpha, \mathtt{fin}) = s^*_X\}, \mathtt{fin}).$$

Time-sorted. By induction it suffices to ensure just that the last dispatch-time is monotonically greater:

$$t' \geq t. \tag{5.17}$$

Epoch-explainable. The constraint to check is that all-part start-times need to be equal to previously established finish-times. There are two cases: this is the first all -part to be dispatched for this time, or not.

In the former case, we need to mimic advancing, perhaps several times in a row, to the new time. More accurately, we need to exclude this particular branch unless the search path already includes the mimicry. So, above and beyond the obvious (checking that the new epoch is a previously established finish-time), we should ensure that all earlier pending effects have already been carried out. We need only prevent skipping ahead of them in time: check that the new epoch is sooner than the soonest of the parent's pending effects. (The sufficiency is by induction; the base case is a lack of pending effects, for which the soonest start-time should be understood as positive infinity.) By inspection, the set of all lock release times consists of (i) only previously established finish-times, and, by time-sortedness, (ii) all finish-times potentially greater than the last epoch (t). So we should choose the new epoch t' from the set of release times.

In the latter case, by the tie-breaking (which forces corresponding start-parts to immediately follow their all-parts), it suffices to additionally check just that the parent's effect-dispatch is of a start-part.

So, for every all-part $a' = (\cdot, \texttt{all})$, either:

$$(t' \in \mathit{Released}(\mathrm{Rng}(V))) \text{ and } (s^*_D \geq t' > t), \qquad \text{or} \tag{5.18}$$

$$(t' = t) \text{ and } (a = (\cdot, \texttt{bgn})). \tag{5.19}$$

Executability. As the child C is by assumption a forward-chaining search node: executability itself is ensured. However, one cannot just pick foward-chaining search nodes out of thin air. By inspection of the definition of forward-chaining search nodes: It is clear that executability up through the child C is computable incrementally from that of its parent P. Specifically check that each of the following is defined: $S' = S'_{a'}(S)$, $V' = V'_{a',t'}(V)$, and $D' = D'_{a',t'}(D)$.

Remark 5.1. Optionally check that the locking constraints implied by the pending effects in the current debt D' remain plausibly satisfiable ("suffix-plausible"), which of course can be done incrementally with respect to just the change wrought by the last action a'. One might perform a similar relaxed check of the state constraints. Importantly though, do not insist upon full executability itself through some completion (a descendant with no pending effects) of the child C: here is where "quasi-" has hidden itself. *I.e.*, it may very well be impossible to achieve the state constraints of the pending effects, and detecting such is *not* necessary. Of course responding appropriately to *easily* detected dead-ends is quite desirable (*e.g.*, as for inevitable violation of the locking constraints).

Tie-breaking. Note that we can only (incrementally) run afoul of the tie-breaking constraints if the current and last dispatch-times are equal: $t' = t$. For *fatten first*: So non-end-parts should check that they follow only non-end-parts. For *pair all+start*:

So firstly start-parts should insist upon the matching all-part. Secondly everything else should refuse to follow an all-part. For *alphabetize ends*: So the effect reaching here needs to have been the alphabetically-least-soonest pending effect (a^*_D). Then for notation, whenever tied ($t' = t$), for α an arbitrary action:

$$a' \neq (\cdot, \texttt{fin}) \Rightarrow a \neq (\cdot, \texttt{fin}), \tag{5.20}$$

$$a' = (\alpha, \texttt{bgn}) \Rightarrow a = (\alpha, \texttt{all}), \text{and} \tag{5.21}$$

$$a' = (\cdot, \texttt{fin}) \Rightarrow a' = a^*_D. \tag{5.22}$$

Furthermore ensure that nothing follows an all-part except its start-part, *i.e.*, regardless of whether $t' = t$ or not:

$$a' = (\alpha, \texttt{bgn}) \Leftarrow a = (\alpha, \texttt{all}). \tag{5.23}$$

In short generating child search nodes may be implemented in 'constant-time' per search node, as desired for the moreover. □

The proof is checking every last detail of ensuring that our picture of the search-forest DE is perfectly faithful. That inspires confidence that insight here is meaningful towards deliberately reworking the design of a DE planner, *i.e.*, so as to improve it. Let us highlight just one interesting example. To drive home firstly the significance: the below alluded to line(s) of code within SAPA withstood a decade of scrutiny [13, 41, 54].

Consider applying $(W)A^*$-ish search [48, 95]. From the effect-schedule perspective it should be reasonably clear how advance-time 'ought' to be evaluated (*i.e.*,

how to update g and h values). In particular, ignore all the temporal details and imagine how a classical planner would evaluate the end-parts.

In contrast, describing DE planners in the traditional way makes as clear as mud a reasonable approach to assigning a search priority (something like an $f = (1 - w)g + wh$ value) to the special child. The more specific challenge in the eyes of a DE planner engineer is that advancing time seems, in the eyes of a temporal heuristic, to be strictly a bad idea. That is firstly because 'just waiting around' does not achieve anything that was not 'already happening'. Then only the negative aspect remains: advancing time only decreases the time remaining in which useful things may be done. As far as an optimal planner is concerned there would be no issue here, but there cannot be such a thing as an optimal decision epoch planner (due to their incompleteness). For satisficing approaches (*i.e.*, something like $w < 0.5$) the issue is excessive backtracking due to the illusion that advancing time is always a bad choice.

Of course in truth the concerned edges of the search-trees are useful, indeed, *necessary*. Meaning events should have the status 'must happen or else', not 'already happening'. So a technical nuance is that fattening choices should *not* be treated intuitively (with respect to action-schedules): the intuitive implementation is to immediately increase cost-incurred-thus-far (g) so as to reflect the entire action. Instead, 'because' such would be natural from the computationally-equivalent and 'classically-friendly' perspective of effect-schedules, a portion should be set aside to be charged towards the advance-time that actually finishes the action off.

Abstracting, the planner engineering lesson is: *treat event queues as (temporal) landmarks*. As far as we are concerned the point is that there is significant insight to

be had by manhandling the description of decision epoch planners into ‘classically -friendly’ accounts.

5.2.5 Discussion

Decision Epoch Planners attempt to create slackless schedules by starting actions only at those times where events are already happening. Unfortunately for them, this translates into the following two erroneous notions concerning slackless schedules:

False: It suffices to start actions only immediately after another has finished starting or finished entirely.

False: Every prefix of a slackless schedule is slackless.

Glancing at Figure 2.13 immediately suggests an approach to remedying the first error. Namely, we can generalize to reasoning about causal interactions concerning the end-parts (above and beyond just the start-parts) [43]. When all is said and done, however, the second error remains. Indeed, the flaw is fundamental to even the most abstract form of the design. It may be stated as: Demanding that decision epochs *monotonically advance* is doomed. In light of the proof of Theorem 3.18 (that considering slackless schedules is completeness-preserving), a technically minded restatement is as follows. DE planners falsely assume that solvability of relatively arbitrary Simple Temporal Networks [47] may be decided in linear time. STNs are indeed simple: just not *that* simple. We can fix this flaw, by permitting the epochs to move backwards in time—and end up with a reasonable approach to temporal planning—but there is a Catch 22: The result can no longer be properly called a Decision Epoch Planner. That is because permitting ‘decision epochs’ to arbitrarily

travel through time, roughly as in LPGP [142], rules out the design goal: Develop a 'state-space' approach to temporal planning. In short, Decision Epoch Planners are broken—at least in theory.

Nonetheless the approach can be defended to an extent, chiefly by considering its practical advantages. It is easy to *understand*, *implement*, and *generalize*. For example, with generalization to continuous nonlinear change, searching through situations directly is much simpler than manipulating symbolic expressions. That will be incomplete, suboptimal, nonsystematic, *etc.*, of course, but making strong theoretical guarantees in planning models sporting continuous nonlinear change is generally impractical anyways. To emphasize ease of implementation we note that TFD [64] is a decision epoch approach developed in full knowledge of our theory [43]; at least, ease of implementation appears to be a significant factor behind the design choice. On a related note, engineered carefully, one might take a decision epoch approach as a Version 1 of a longer term plan to replace the epochs with constraint reasoning. As far as understanding goes, for example: In the domain-dependent context one may craft effective rules for selecting the epochs, thereby attaining 'completeness in practice'. Also, temporal heuristics easily benefit from knowledge-of/commitment-to precise timing decisions [54].

That said, all real systems seem to approach numeric infinities with notably greater sophistication. For example, we may (adaptively) discretize, (manually) abstract, apply local/stochastic search, reduce to (Mixed) (Integer) (Non-)Linear Programs, consider Hybrid Automata, or develop customized constraint languages (*e.g.*, flow tubes), among presumably many other techniques. Particularly pointed is that local/stochastic search is a strictly superior perspective. That is, if we are going to be deliberately incomplete, then we may as well go about it the right way.

So for example, just scratching the surface, randomized versions of decision epoch advancement may at least claim stochastic completeness.

That may be the best takeaway: *Decision Epoch Planning is a local search approach to temporal planning*—(re)design appropriately. Next, and finally for technical content, we shall attain forward-chaining, completeness, and systematicity together; the technique is to reason with temporal constraints upon the dispatch -times (in contrast to selecting them up-front).

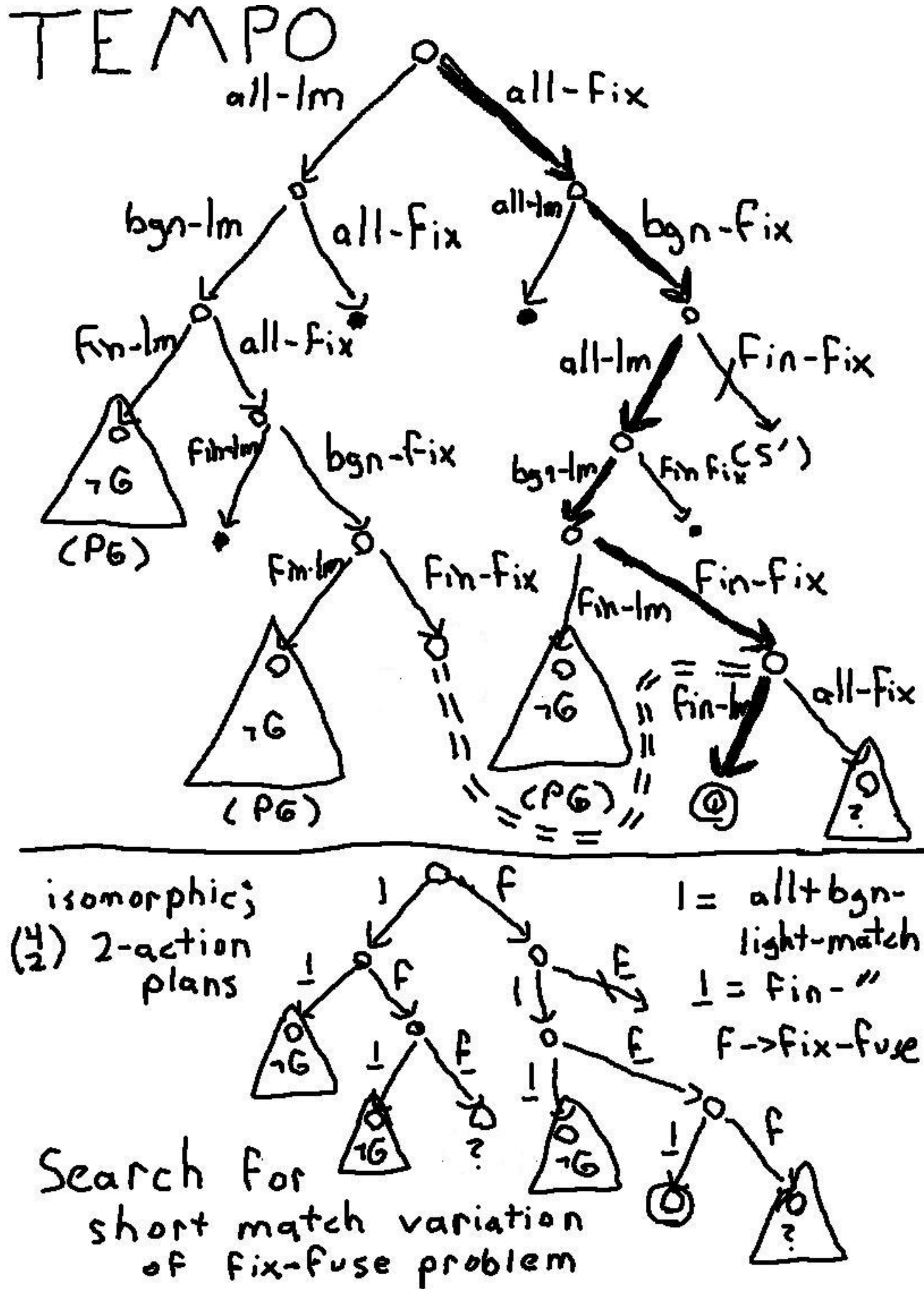

Figure 5.2: Two TEMPO search trees: with/without reduction by Proposition 5.18. See Section 5.3.1.

5.3 TEMPORALLY LIFTED PLANNERS

The key observation about decision epoch planners is that decisions about when to execute actions are made *eagerly*—before all the decisions about what to execute are made. In this section we construct technical support towards: *Lazily* making scheduling decisions is better.

Let a TEMPO planner be short for a lazily-scheduling forward-chaining temporal planner, equivalently: a temporally lifted (forward-chaining) planner. (By analogy we may call VHPOP [208] a lazily-scheduling partial-order causal-link temporal planner.) To our knowledge, only Fox, Long, *et al.* have pursued the concept to the level of implementation, beginning *circa* 2003 and continuing through 2012: CRIKEY/CRIKEY3/POPF/POPF2/COLIN [37, 93]. As far as practical motivation goes, as judged by the 2011 instance of the temporal planning competition: the version spearheaded by Coles and Coles, POPF2 [36], is at present the most empirically effective among the complete approaches to Interleaved Temporal Planning [76].

Recall that an overarching point is that required concurrency is an open—and potentially solvable—challenge facing the temporal planning state-of-the-art. Our final technical result goes a long way towards solving the challenge.

We formally define the search-tree shortly, preceded by a glance at partial evaluation. Following are our theorems demonstrating completeness and systematicity for the reductions by slacklessness and deordering. The latter effectively serves as our definition of required concurrency; to be complete and systematic for a reduction by deordering 'solves' the challenge. We close with some discussion of the significant issues just glossed over.

Definition 5.8. A **temporally lifted** expression is a symbolic expression entirely lacking temporal literals. In practice, to **temporally lift** a computation is to evalu-

ate 'as much as reasonable' subject to not evaluating temporal symbols. For accuracy: also any hardcoded temporal literals need placeholders. A **symbolic expression** is a quoted, *i.e.*, not evaluated, expression. Denote the type of expressions by *Expressions* and the type of symbols by *Symbols*.

So for example, "1+2+3" is a symbolic expression, and is *not* equal to 6. When writing in prefix-form it is common to instead write: '(+ 1 2 3). With $x = 2$ and $s =$ '(+ 1 x 3) then **partially evaluating** s, written (partial-eval s), results in 6 [92]. However, with $x^{\in Symbols}$ denoting a symbol with value as-of-yet unknown and $s =$ '(+ 1 x 3) then (partial-eval s) results in either s, '(+ 4 x), or '(+ x 4): reader's choice. (Partially evaluating again presumably reaches a fixpoint.) The point here is to support evaluating as much as possible about a hypothetical (equivalence class of) schedule(s) without actually committing to particular dispatch-times.

The search-tree of TEMPO planners is quite straightforward: all effect-sequences (sequences of parts of actions), connected by forward chaining. So (X, a) is the child of (X) for all effect-sequences X and effects a. More formally, the effect-sequences are the concatenation of the edge-labels along the path from the root to any given vertex, and distinct vertices are reached by distinct effect-sequences. (Meaning that an effect-sequence may leave the tree, but otherwise it uniquely names a path.) We are primarily interested in the point that we can compute a great deal about the situation that would result were we to assign dispatch-times and execute. In particular we can compute the resulting states exactly:

Definition 5.9. Let $N = \left(parent, op; t, State, \hat{V}, \hat{D}\right)$ denote a **temporally-lifted forward-chaining effect-schedule search node**, which is named by (a reference to) its parent node P and the effect $op^{\in Primitives}$ labeling the implicit edge from its parent; following are derived structures including the (temporal) symbol $t^{\in Symbols} =$ 'AST_d,

with d the depth ($d(N) \coloneqq d(parent_N)+1$), standing for the intended start-time of the effect op, the current state $State$, the current temporally lifted vault $\hat{V}$, and finally the current temporally lifted debt $\hat{D}$. If the parent is null, then the search node is a root of the search-space and represents both the empty plan and some initial situation; write along the lines $N_0 \coloneqq (\emptyset, \text{init}; \text{AST}_0, State_{Initial}, Vault_{Initial}, Debt_{Initial})$. Everything else must satisfy (or be deemed illegal) the temporally lifted notion of executing effect-schedules, with $a = op$:

$$State \coloneqq S'_a(State_{parent}), \tag{5.24}$$

$$\hat{V} \coloneqq V'_{a,t}(\hat{V}_{parent}) = (\text{partial-eval '}(V'\ a\ t\ \hat{V}_{parent})), \quad \text{and} \tag{5.25}$$

$$\hat{D} \coloneqq D'_{a,t}(\hat{D}_{parent}) = (\text{partial-eval '}(D'\ a\ t\ \hat{D}_{parent})). \tag{5.26}$$

For an interesting abuse, let TEMPO denote the type, which constitutes an infinite forest on every legal temporally lifted search node: one tree per interleaved temporal planning problem.

Note that all fields of a search node are determined by its parent and the edge(-label) connecting them: any given search tree is no more than all effect-sequences. In practice the definition already prunes many, because the temporally lifted effect-schedules (*i.e.*, the effect-sequences) easily fail to be executable despite only partial evaluation. For example, all violations of the classical constraints (*i.e.*, the state-transition functions) are caught. Similarly, if the parts of an action are executed out of order, then the temporally lifted debt in question will fail to exist. In contrast, technically, the computation of the temporally lifted vaults does nothing, as it will never fail (as written).

We include the vaults anyways, partially for symmetry, but mostly to reflect the incremental costs of building (and if desired, solving) the associated Simple Temporal Networks. Also, the gory details of the proof of Lemma 3.19 are made somewhat clearer by keeping them. In particular the lemma wrestles with the technicality that the partially evaluated vault-sequences are not literally the same as the precedence constraints of the associated Simple Temporal Networks. (They mean the same thing, and cost the same to build/solve, but are not literally identical.)

Technical commentary aside: It ought to be fairly apparent, from the form of the definition, that TEMPO planners easily achieve completeness (and, for that matter, soundness). Let us prove so in a bit more detail.

Lemma 5.12. TEMPO *may be taken to cover all and only slackless effect-schedules.*

Proof. More specifically we may check at every search node that the associated Simple Temporal Network remains consistent and prune if otherwise. (Claim) Under said pruning, both directions hold.

Assume a slackless effect-schedule, which implies executability. Its underlying effect-sequence labels a hypothetical path of the tree. As the constraints checked by the partial evaluation of the transition functions are a strict relaxation of the assumption of executability: the path is legal. Likewise satisfiability of the STNs along the way are witnessed by the assumed schedule (see Lemma 3.19). Then the path remains unpruned by the only additional pruning under consideration: the all direction is shown.

Assume a legal search node, with satisfiable Simple Temporal Networks all along the path reaching it. See Theorem 3.18 (for context) and Lemma 3.19 in particular. By the lemma, the partial evaluation of the execution, along with satisfac-

tion of the STN, guarantees a slackless scheduling of the effect-sequence reaching this search node. So the only direction is shown.

Therefore, both directions are shown. □

So, in other words:

Theorem 5.13. *Some* TEMPO *planner is complete and systematic, with respect to the dominance reduction by slacklessness, for Interleaved Temporal Planning.*

Proof. By Theorem 3.18, *slacklessness* does indeed define a potential dominance reduction within ITP. By Lemma 5.12, some TEMPO planner may be taken as implementing that reduction—perfectly, *i.e.*, covering every equivalence class once and only once. □

Remark 5.2. Duration-optimality follows (as normal, if desired), as does optimality for any other metric that may be computed solely from the effect-sequence underlying a given schedule. For example, TGP's notion of quality is to wait as long as possible without increasing overall duration (*i.e.*, TGP right-shifts). Some TEMPO planner is optimal for that metric (maximize sum of start-times subject to minimizing total duration), because we may reverse the direction of time in the STN for each effect-sequence (flip the direction of every edge and negate its weight).

Then we have a workable approach to ITP. However, just *slacklessness* alone is a relatively weak characterization of 'strongest possible reduction in general'. Indeed, it is more or less the weakest reasonable reduction: it serves only to eliminate the infinity of time.

Figure 5.3: A deordered-slackless-reduced TEMPO search tree (*cf.* Theorem 5.16). Also see Section 5.3.1 and Figure 5.2.

5.3.1 Discussion of Figure 5.2 and Figure 5.3

The upper part of Figure 5.2 is strictly for literalism with the definitions. As alluded throughout, the minimum reasonable implementation only uses 2 parts per action. So the first equivalence reduction to implement is Proposition 5.18 (below).

Having done that, we have a reasonable looking search space for the problem of fixing the fuse. (The bottom half of the figure.) For larger problems though, we have some hints of problems lurking even here.

5.3.1.1 Shallow Technical Remarks on Figure 5.2

The “?” labels a sub-tree that would be possible if fixing a fuse was faster than burning a match. Whether our planner will explore that will depend on whether it can do the *algebra* to notice the impossibility. (In contrast with *arithmetic*.) So to prune that subtree, the implementation will need to be just a touch more complicated than in the DEP case. Namely, implementing Theorem 5.13 is quite important.

On top of that we should likely consider pruning *before* the STN becomes obviously inconsistent. Particularly with many pending end-parts, at a minimum, we should go ahead and create their vertices sooner rather than later. The formal definitions omit the optimization. In part that is because, really, an implementation should have yet greater ambition.

The more sophisticated thing to do is to use inference techniques to predict that even more than the endings of compounds are forced. In particular, landmark analysis ties in seamlessly with understanding that end-parts of compounds must be carried out.

There are two differences from ordinary landmarks. Durative actions carry a quantitative constraint on the maximum time that could elapse before they must go into the plan. So they are like deadline-annotated landmarks. Second, that forcing is built in; no extra code, nor runtime, is needed to have the knowledge.[2]

These are minor points: just the typical sort of detail usually better ignored. What is really compelling here is to implement deordering.

5.3.1.2 True Ambition: Figure 5.3

The difference between the two figures cannot seem large: we began with a toy problem. The difference is that one part of the search tree is pruned one level sooner. What must be appreciated is that such is pruning that is happening early in the tree: the savings are exponentially larger as we scale up. The reason that it may be pruned is that it is semantically equivalent to a different path in the tree;

[2]So to a classical planner, temporal planning looks like domain-dependent planning, in just about the same way that HTN planning looks like advice rather than physics. In practice, such is a point in favor of temporal planning.

it is “non-canonical” with respect to the notion of equivalence enshrined under the name *deordering*.

As we shall see shortly, the computational (and software engineering) cost to pull the detection of the equivalence off is negligible relative to managing STNs. So in particular, at least on this one toy example, the approach from temporally lifting—implemented well enough—would easily trounce DE approaches. Indeed, the state-of-the-art in forward-chaining temporally-lifted interleaved-concurrency planning (POPF [36]) beat out its decision-epoch counterpart (LMTD [120]) in the latest planning competition [76]. Then we should find polishing off our last theorem a compelling prospect.

5.3.2 Combining Deordering and Slacklessness

Then consider that two effect-sequences may end up with isomorphic associated STNs, and hence, intuitively speaking, also end up with identical schedules; formally the relationship is deordered-equivalence. So we should be able to keep just one of them. By Theorem 3.21, deordered-equivalence does in fact define an equivalence-reduction. Then above and beyond implementing slacklessness, we should implement the reduction by deordering. For a name, say **deordered-slackless-reduction** is the reduction by both deordering and slacklessness; the order in which the reductions are applied does not matter. Then our next and final technical goal is to demonstrate that TEMPO can implement the deordered-slackless-reduction.

Implementing the pruning rule itself is easy. Proving it correct calls for a detour through lemmas. What needs to happen here is that we need to efficiently recognize isomorphic partial-orders: two slackless schedules are deordered-equivalent precisely when the two concerned sets of precedence constraints are equal (*i.e.*, equal

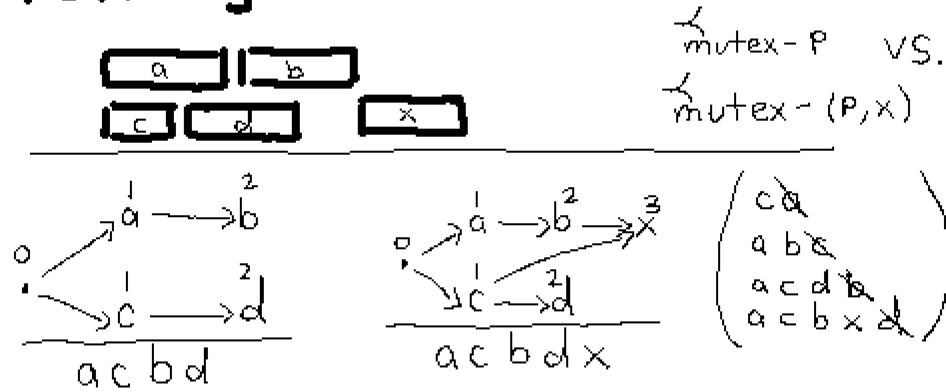

Figure 5.4: Isomorphism of labeled partial-orders is easy and exploitable.
See Proposition 5.14 and Lemma 5.15.

as partial-orders). Graph isomorphism in general, even restricted to only graphs of partial-orders, is quite the special problem. Here, though, our partial-orders are far from arbitrary. Indeed, our formal problem is actually isomorphism of *labeled* partial-orders, which is trivial, as follows.

Definition 5.10. Say the effect-schedule $X = (a, t)_{[n]}$ is **deordering-canonical** when for all $(i < j)^{\in[n]}$:

$$\text{rank}\,(X_i) \leq \text{rank}\,(X_j), \text{and if equal then tie-break by:} \tag{5.27}$$

$$a_i \prec_{\text{id}} a_j. \tag{5.28}$$

Details: To compute the rank of each dispatch of X, view the deordering of X, which is a particular partial-order on its dispatches, as a directed acyclic graph. The **rank** of a vertex in a directed acyclic graph is given by $\text{rank}\,(v) := 1 + \max_{u \in N^-(v)} \text{rank}\,(u)$, and by $\text{rank}\,(v) := 0$ when v is a root (in-degree 0). Assign an arbitrary totally-

ordered set of identifiers to *Primitives*: write $<_{\mathrm{id}}$. For example, take their names in dictionary order.

So, insist that ranks monotonically increase and tie-break arbitrarily but consistently. The connotations hold:

Proposition 5.14. *Our choice of a canonical representative for a deordered-equivalence class is in fact unique.*

Proof. Pick some effect-schedule $X = (a, t)_{[n]}$ arbitrarily. First let us demonstrate existence.

Suppose there is a descent in rank: $\mathrm{rank}\,(X_i) > \mathrm{rank}\,(X_{i+1})$. By the construction of deorderings, the two are non-mutex and belong to distinct actions, as otherwise contradicts the supposition. So we may swap them, preserving deordered-equivalence by definition. That reduces the number of inversions. So by induction on the number of inversions, eliminate all descents. Then we have some X' that is monotonically increasing in rank and deordered-equivalent (to X).

Consider any two equal-rank dispatches of X'. The two are non-mutex and belong to distinct actions, as otherwise contradicts equality of rank. Then every such pair are non-mutex. So every way of permuting the equal-rank subsequences is deordered-equivalent, by induction on its definition. In particular a deordered-equivalent permutation of just the equal-rank subsequences meets the chosen tie-breaking: A canonical representative exists, say Y.

Then it remains to show lack of plurality. For a contradiction, thereby establishing lack of plurality, suppose existence of a distinct deordered-equivalent canonical schedule Z. Note that, from above, we already have that Y is unique up to permuting the equal-rank subsequences of X'. Meaning that Z could only differ in

some *set* of equal-rank dispatches (rather than sequence). So take rank i as minimal such that the sets of dispatches at rank i differ. Wolog (swap Y and Z), let dispatch $x \in \text{rank}_Y^{-1}(i)$ and $x \notin \text{rank}_Z^{-1}(i)$ witness the difference. By minimality, every predecessor of the witness x has equal rank in both Y and Z (because every predecessor of x has rank less than i in Y). Rank is a function of only the predecessors, so the rank of the witness in Z is the same as in Y: $\text{rank}_Z(x) = i = \text{rank}_Y(x)$ is forced. Which, as desired, contradicts that the dispatch x witnesses a difference: $\text{rank}_Z(x) = i$ contradicts $x \notin \text{rank}_Z^{-1}(i)$. Hence lack of plurality, and uniqueness follows. □

We may extend the notion of deordering-canonical to effect-sequences directly. Which is what an implementation ought to do: define the vertices of a deorder by the multiset of effects underlying dispatches (rather than using the dispatch-times to disambiguate). In contrast would be building a STN: in order to schedule the sequence, in order to build the deorder, in order to, finally, compute the ranks. For reference, the computation of ranks can be done in constant-time per effect (linear in the fluents that an effect depends on, which is a constant throughout a search tree), at a small cost in memory usage (one, small, natural number per fluent).

Picking canonical representatives is the relatively obvious part of implementing a reduction. The easy part to overlook is ensuring that the particular choice of property is lost forever whenever lost.

Lemma 5.15. *Under forward-chaining, with respect to deordering: Pruning non-canonical schedules loses no canonical schedules.*

Proof. A deordering-non-canonical schedule is witnessed by a descent in rank or a descent in the tie-breaking. It suffices to show that such descents are invariant under forward-chaining, as then every extension must remain non-canonical.

The arbitrary choice of tie-breaking is invariant, because it is fixed before all search.

(Claim) The rank of a dispatch is invariant under forward-chaining. Argue by induction, with base case an empty schedule (and so nothing to show). Adding a dispatch to the end of a schedule only adds edges leaving the graph of the deorder. So no predecessors are added. Then all ranks remain as they were, because rank is a function of only the predecessors. □

With that, the correctness of implementation is a breeze:

Theorem 5.16. TEMPO *may be taken as complete and systematic for* ITP*, with respect to the deordered-slackless-reduction.*

Proof. Prune all search nodes that fail to have a solvable associated STN; so only schedulable effect-sequences remain. Consider them to be slacklessly scheduled: only slackless effect-schedules remain. By Theorem 5.13: all (and only) slackless effect-schedules remain.

Prune all that fail to be deordering-canonical: only deordering-canonical slackless effect-schedules remain. By Lemma 5.15: all (and only) deordering-canonical slackless effect-schedules remain. Which is the meaning of: (strongly) complete and systematic with respect to the deordered-slackless-reduction. □

The theorem is powerful. See Figure 5.3 for an example of applying it, as already discussed. The example understates the power.

Corollary 5.17. *Consider the deordered-equivalence class of some* causally sequential *schedule, in the context of an implementation of the deordered-slackless-reduction. There exists precisely one unpruned search node covering that class.*

So, more to the point, consider a causally sequential problem given to a TEMPO *planner implementing the reduction. The remaining search-tree is no larger than the search-tree on classically-sorted slackless effect-schedules.*

Proof. Straightforward applications of the theorem. □

In simpler terms yet, what the theorem does is *automatically simplify to Conservative Temporal Planning*. With respect to the present empirical regime, that is the difference between temporally lifted approaches being considered (1) a promising research direction, and (2) state-of-the-art.

Now, somewhere between the understatement of the Figure and the overstatement just made is the truth. The following is a much more accurate and precise discussion of the technical merits and flaws of the theorem. The short of the matter is that what we have demonstrated here is a very promising and rather detailed direction for future work.

5.3.3 Significance and Limitations of Deordered-Slackless-Reduced TEMPO

Recall the technical motivation for required concurrency: try to avoid considering any interleavings besides the degenerate, *i.e.*, sequential/conservative, cases. That is, the technical aim is to consider only classically-sorted effect-sequences (*cf.*, Theorem 4.10). So the significance of Theorem 5.16 and its corollary is that we can bound our worst-case by the right quantity. That is quite promising. The search

tree, though, has the 'wrong' shape. That could end up being quite significant when delving, for example, to analysis of heuristics.

The specific technical point concerns something like *macros*; what we would really like to do is to always immediately step through all parts of an action. That is what it means to prune down to classically-sorted interleavings. This final theorem, however, does not neatly line up all parts of an action for us. Indeed, it more or less works as hard as possible to separate parts of an action by as much as possible. Quite promising, though, is the point that, in a causally sequential domain, the bias amounts only to a sequence of pairwise non-mutex action-parts. (That is, the end-part in such domains has greater rank *only* by virtue of needing to follow the start-part.) Which is almost strong enough to allow us to reorder into the desired, classically-sorted, form. However, unless we are told or are able to prove the property (causally sequential), then we cannot freely assume the end-part will always end up with rank strictly 1 greater. Perhaps a related proposition paints a picture:

Proposition 5.18. *'Any' approach to* ITP *may be taken as searching through interleavings of just two parts per action,* i.e.*, rather than three parts.*

Proof. Consider the naïve graph on all executable schedules and reachable situations. Pruning by time-sortedness preserves completeness. The all-part and start-part must have identical start-time. Then by tie-breaking on action names, we guarantee that every remaining schedule, *i.e.*, unpruned, always immediately follows up an all-part with the corresponding start-part. Compose the situation transition functions of the two; replace both with this syntactically/notationally more complex object. (A regression search would presumably prefer to make a macro out of the all -part and end-part: also possible.) The reason to complicate syntax/notation is that doing so permits us to neatly relax away the pruning rules used as justification for

the manipulation (argument omitted); that is useful towards applying different completeness-preserving reductions, *i.e.*, different approaches. In short actions of ITP effectively have just two parts, as far as search complexity is concerned, regardless of approach. □

So what we would like is a similar result for end-parts. Which we have—provided appropriate knowledge—but automatically proving that a problem is causally sequential is, in general, just as hard (if not harder) than planning itself. That said, it is always worth keeping in mind that 'cheating' is practical.

5.3.4 Two Suggestions towards Solving Required Concurrency in Practice from within the Temporally Lifted Forward Chaining Perspective

There are several cute tricks we could apply towards immediately forcing end-parts into the plan in coincidentally causally sequential problems. The simplest is to just temporarily assume the knowledge, so apply a FFC planner; should that step fail, then fall back to a complete approach, *i.e.*, some TEMPO planner. Such a portfolio is easy to implement and surely quite effective in practice: a good baseline. A more complicated, and computationally superior in theory, technique is to apply 'iterative broadening' to some parameter that happens to take value 0 on causally sequential plans. Sketch and discussion thereof follows.

Consider that, having just applied the all-part (+ start-part, see above) of an action, we could immediately branch on whether the end-part will occur with minimal rank (1 more than the current rank), or greater (2 or more). Admittedly that undermines the notion that the search is a straightforward implementation of "forward-chaining". However, heavily penalizing the second choice branches is (much of) the desired exploit: too promising to pass up. Regarding them, note that it will be

necessary for something mutex with the end-part to occur first (else its rank will be minimal). So, in order to implement the intended semantics of the choice, artificially disable applicability of the end-part until something mutex does in fact intervene. (More accurately, have search nodes track the set of artificially disabled end-parts: re-enable each when its rank would be correct.) If such intervention is impossible, the primary example being causally sequential problems, then the second choice leads to a (potentially extremely large) dead-end. In practice it would likely be empirically important to bound how much search time is spent in that dead -end.

More importantly though, consider the first choices. To say that the end-part occurs with minimal rank is to say that nothing mutually exclusive intervenes between when it first becomes possible and when it is finally included. (A mutex effect with the same rank as the start-part might 'need' to happen first, an example is forthcoming.) Finishing actions, having started them, is by definition required: the end-part is trivially an action-landmark (for the goal). So consider Proposition 3.2. Then we may force the end-part to occur immediately after it first becomes possible. *If already possible then we will have essentially compressed the action* [36]. In particular, if the entire problem is causally sequential then we will—almost—end up forcing exploration of just classically-sorted effect-sequences, as desired.

Similar to forcing start-parts to occur immediately, there are some details to be aware of. In particular subsequent pruning by the property of deordering-canonical breaks if nothing more is done, because the current rank would increase too soon. So the definition+proof+implementation of deordering-canonical and corresponding pruning would be somewhat more complicated, more or less as follows. Mark (artificially) forced effects/dispatches. Redefine canonical so that a sequence

is canonical when instead some reordering of just its marked elements is canonical by the old definition (delay them to the right positions). The manipulation is deterministic and invertible (despite the existential implied by "some"), which would be the crux of proving correct the altered definition. Implementation is likely straightforward as pruning is presumably enforced incrementally. Specifically, prior to fixing the implementation of pruning the incremental rule is just: the rank of the child should either be larger than that of its parent, or if equal, then its id should be larger than that of its parent (failing both, then prune). Fixing which, *i.e.*, in the presence of artificially forcing end-parts, is straightforward. Alter just the pruning at their children: take the rank and id from the grandparent, rather than from the parent.

To round out the practical considerations, consider the potentially large search effort wasted in exploring the potentially difficult to detect dead-ends possibly brought about by following the second choices. 'Best' would be to reliably detect such dead-ends (and so prune), *e.g.*, by using landmark analysis as in the proof of Theorem 4.12. The quick and dirty (but surely effective) fix would be to penalize exploration of all such second choices by whatever scheme—anything would be better than no penalty. A specific and extreme possibility—and notably promising —is to use a new first component to the search priority: count the "degree of concurrency" currently unfulfilled. (The second choices increase it, the concerned end-parts decrease it: the size of the set of artificially disabled end-parts is the "degree of concurrency".) Note that, in words, such second choices correspond to demanding that the action in question ultimately ends up as a psuedo-witness to required concurrency.

(Counter-)Example. The psuedo-witnessing of required concurrency underneath the second choice branches is indeed imperfect: consider the deordering-canonical sequence "all-A + bgn-A (rank 0), all-B + bgn-B (rank 0), fin-B (rank 1), fin-A (rank 2)". In the modified search-tree the sequence would read "all-A + bgn-A (rank 0), non-minimally finish A, all-B + bgn-B (rank 0), minimally finish B, force fin-B (rank 1), fin-A (rank 2)". The rank annotations are saying that the all-part and start-part of A are non-mutex with every part of B; hence it would be deordered-equivalent (but not canonical) to dispatch all of B and then all of A. Therefore the canonical sequence is causally sequential, meaning A fails to witness required concurrency; to witness, the rank sequence would have read 0, 1, 2, 3. So it is not *quite* true that the described approach in general attains systematicity alongside considering all causally sequential plans first. It is close though, as discussed in a moment.

While arguably a 'hack', the approach has merit. To support a non-hack perspective one terms it an 'iterative' broadening approach: broadening on the "degree of concurrency". (As described there are no iterations *per se*; prepending a component to the search priority is similar enough.) Such would likely be at least slightly superior to applying a sequential portfolio consisting of a FFC planner followed by a TEMPO planner. That is because the second planner of the portfolio would redo the search effort of the first. In contrast the broadening approach would still enjoy systematicity. Furthermore, importantly, both more or less consider plans in the same order.

Specifically, the portfolio ensures exhausting all causally sequential slackless effect-schedules (*i.e.*, action-sequences) first. The broadening approach somewhat achieves that; a counterexample is above (the only way to find the action-sequence "B, A" in the example above is to take a second choice). However, note that, sup-

posing some isomorphic image of a conservative temporal planning problem, then, *without needing to know that fact*, the broadening approach does manage to exhaust all action-sequences first (as desired). More specifically suppose that nothing mutually exclusive with an end-part could ever occur, for arbitrarily complicated reasons, between the corresponding start-part and itself. Then, trivially, the parameter "degree of concurrency" is always 0 in every non-dead-end. That is: every second choice is a dead-end by supposition. So every potential solution would be tried, in the desired (classically-sorted) form, first. If still no solution is found, then *we* know the problem is unsolvable; the planner would continue on, exploring the dead-ends, until such became apparent (*i.e.*, until the "arbitrarily complicated reasons" are enumerated). Which is as desired, especially as far as comparisons with the portfolio approach are concerned.

It is, of course, *not crucial* to force exploration of all causally sequential schedules first. Indeed, consider a heuristic able to prove/estimate a lower bound over all solutions upon the maximum value of "degree of concurrency" throughout each. Then we can generalize the discussion above to include this heuristic as part of the calculation of the first component of search priority; that would be an improvement. In other words, if we can guess that the problem does, to some extent, require concurrency, there is every reason to avoid excessively penalizing the appropriate plans. Considering that the compelling examples of required concurrency are likely the results of compilation (*cf.* Theorem 4.12), then it does seem especially significant for temporally expressive planners to avoid penalizing causally non-sequential plans.

However, the far more important side of that coin is still the *absence* of required concurrency. To wit, empirically (*i.e.*, with respect to current benchmarks), and for

that matter theoretically: realistic problems have small (minimax value of) “degree of concurrency”. That is because we should be able to take any goal expression as (the crucial portion of) an initial state towards a second subproblem of a much larger problem. The minimax “degree of concurrency” for the two subproblems together will only be the maximum over the two subproblems—whereas the total size of the combined solutions grows additively. (The same is also true, perhaps surprisingly, of parallel decomposition of global problems into subproblems: because we do not insist that schedules be found in time-sorted order.) So, in general, relative to solution size, the “degree of concurrency” ought to be considered small. (Else the appropriate field is Combinatorial Search, not Automated Planning.) Therefore even an approach as heavy-handed as ‘iterative’ broadening upon the parameter, while certainly not computationally ideal for anything besides a minimax value of 0, is promising enough: because the number of interleavings combinatorially explodes, so ‘only the final iteration would count’.

5.3.5 Wrapping Up

Intuitively speaking, long-term effects are clearly more important, in general, than short-term effects. We should have our planners exploit that. Meaning we should indeed be designing our search spaces, as just discussed at length, to penalize hypothetical instances of required concurrency. The state-of-the-art seems to be missing that mark; the significance/challenge of required concurrency is to *ignore* it, as much as reasonable, without outright forbidding it. It is too far a stretch to claim that the challenge is solved by either the formal theory developed here or the discussion of relatively easy ways to operationalize. Entirely fair, though, is to say that: Our technical contributions here are worthy achievements towards understanding

and implementing efficient Interleaved Temporal Planners by way of leveraging forward-chaining classical planning technique. The technical results, to a degree, substantiate:

- Constraint reasoning is the right approach to temporally expressive planning.
- Catering to the sequential planning view should be a high-priority design goal.

5.4 SUMMARY

We putatively set out with the aims: (i) debunk single-objective FFC planners yet support multi-objective FFC planners, (ii) debunk DE planners, and (iii) support TEMPO planners. In fact, the technical results are on the whole, as they should be, far less polarized. So to look back with a balanced eye:

Section 5.1. As far as Conservative Temporal Planning goes, straightforward manipulation of classical planners is a tough strategy to beat.

Theorem 5.3 Integrating First-Fit into any classical planner yields a theoretically reasonable, and temporal planning competition winning, satisficing approach.

Theorem 5.4 For difficult to meet deadlines, equivalently, for duration-optimality, we ought to slightly rework the approach. Specifically operationalizing deordering (*i.e.*, attaining completeness and systematicity for the equivalence-reduction) as replacement for duplicate state elimination appears quite promising. In support we identified an appropriate—'easily' implemented, computationally 'free'—pruning rule: insist that mutex-order rank monotonically increase + tie-break arbitrarily.

Albeit, there is little, if any, empirical evidence as of yet to corroborate this theory: which is a point future work might do well to elaborate upon. Likely the closest related and recent implementations of the computational motivation are VHPOP and CPT [202, 208].

Section 5.2. Generalizing classical planners to Interleaved Temporal Planning, in contrast, presents notable challenge. The challenge is especially daunting to the state-space/forward-chaining approaches (which at present empirically dominate,

as judged by the classical planning competitions). The first issue—easily resolved —is that something must be done to reduce the infinity of time to anything rather less daunting. Various such reductions readily suggest themselves. As it turns out though, (flawlessly) operationalizing them within a forward-chaining paradigm is trickier than it appears. Specifically the *decision epoch* device possesses significant drawbacks.

Theorem 5.6 and 5.6 Directly navigating through temporal situations is bound to miss key possibilities: incompleteness is 'inevitable'.

Theorem 5.10 The approach is moreover somewhat prone to examining 'the same' plan many times over. With careful attention to detail though, the flaw (lack of systematicity), we conjecture, may be greatly mitigated or even eliminated; for example it seems promising to permit dispatching actions at any given decision epoch only when that time is an earliest start-time.

Such are the flaws of local search, which are hardly death knells, and there is certainly reason enough to remain committed to a direct search through temporal situations. Then the lesson, inasmuch as that commitment remains steadfast, is to mitigate as normal. For example, decision epoch selection by 'coin-flipping' seems to be a clear route to improved performance.

Section 5.3. Such cavalier sacrifice of our ideals is neither especially right nor especially wrong; true, completeness and systematicity are always inevitably trumped by performance, but forestalling tooth-and-nail is worthwhile. So we came to the other side of the coin that is reduction: the equivalence classes, rather than the canonical representatives. More specifically, there are two complementary ways to

characterize more or less equally well the manner in which the state-of-the-art classical planners reduce to search: state-space, or forward-chaining. From the latter angle the way from classical planners to completeness and systematicity for Interleaved Temporal Planning is clear.

Theorem 5.13 Integrating with temporal constraint reasoners (specifically solvers of Simple Temporal Networks) attains at least theoretical plausibility (satisficing or optimal as desired).

What scant empirical evidence exists suggests that temporally-lifted forward -chaining is indeed the 'right' approach to greater degrees of temporal expressiveness [36, 76]. Empirical analysis is murky, though, and for good reason—required concurrency—which point applies no less to decision epoch approaches, but there manifests less clearly (perhaps visible through Lemma 5.11), simply because such are not theoretically pristine. In any case we put forth exploiting coincidental, localized, absences of causally required concurrency between actions as the key computational challenge presently facing the state-of-the-art.

Theorem 5.16 Implementing deordering (*i.e.*, as a replacement for lack of duplicate elimination) is again quite promising, and far closer to "theoretically reasonable". The pruning rule itself remains the same; the drawback is that the search-tree is over effect-sequences rather than action-sequences.

In other words: Finally we shed some light on how one might indeed go about 'solving' required concurrency. The challenge remains open. Specifically our result only exploits causal independencies in general (rather than focusing on

the short-term/long-term distinction); while quite promising, the result falls somewhat short of the promise of Theorem 5.4.

Wrapup. In this chapter we sought and attained better understanding, through the lens of concurrency, of the forward-chaining fragment of the state-of-the-art temporal planning algorithms. Specifically with respect to our prior work we have slightly improved in all of breadth, depth, accuracy, precision, and promise [43]; in general, the results are novel and illuminating.

Chapter 6

Conclusion

Over the past decade the computational performance of temporal planners is *reported* to have steadily and considerably improved [36, 36, 37, 39, 54, 56, 57, 64, 84, 85, 86, 87, 93, 96, 100, 120, 129, 142, 146, 184, 198, 201, 202, 208]. *If* true then significant credit is surely due to the expedient of empirical analysis [60, 71, 76, 82, 105, 141]; "significant" because truly compelling empirical analysis of domain-independent planners is entirely nontrivial (*e.g.*, such analysis is not at all unlike designing an IQ test). Our skepticism is warranted. True, more than a few entirely practical and self-styled "temporal" planning systems perform well enough by all accounts [1, 12, 37, 110, 137, 154, 168, 169]. Still, 'any' kind of temporal reasoning is undeniably *hard* in theory [2, 4, 70, 88, 109, 112, 145, 148, 151, 165, 173, 175]. In particular, theory warrants investigation of:

Is the reported performance of temporal planners due merely to evaluating against temporally sugared classical planning problems?

Unfortunately we found that the short answer is "Yes" [42, 43]. For example, LPG, MIPS, and DAEYAHSP2 are all clearly minor modifications of classical planning cores [56, 57, 87]. The tragedy here is loss of faith in the data. Among other things, all the plots are missing a crucial line: as LAMA is presently performing best among classical planners, we ought to similarly 'generalize' it to time and compare [76, 174]. It is no stretch to suppose that said strategy will win every time, at least, so long as the competitions continue to prefer the temporally simple side of

the equation we can expect the baseline to win. Regardless, the baseline—dumb-as -possible application of classical state-of-the-art to temporal planning—*needs* to be present. How else can we have faith in conclusions reached by empirical analysis?

The issue is simple, dreadfully obvious even, but absolutely crucial, indeed, goes far beyond just the relationship of temporal planning to classical planning. McDermott's creation of a standard (appropriately called the *Planning Domain Definition Language*) around the turn of the millenia set apart *all* research into automated planning before and after [152]. After, both proofs and data are meaningful, before, only proofs. *Contribution* is too small a word.

With respect to specifically temporal planning, we note that "temporal" has been associated with many different facets of a planning problem: *durative actions*, *deadlines*, *concurrency*, *processes*, *trajectory constraints*, *exogenous events*, and *continuous change* for example. Lacking any particular standard, researchers, naturally, cherry-picked their favorite features, giving rise to many flavors of temporal planning. This was to have been resolved by the generalization of PDDL to time [71]. The effort has proven quite successful: a decade later we find a reasonable number of temporal planners for which pairwise empirical comparison *is* face -value meaningful. That said, "temporal PDDL" remains ambiguous: we also find a disturbing number of temporal planners purporting to support PDDL, yet implementing grossly distinct semantics.

6.1 KEY LESSONS

When *practice* refuses to conform, it is time for *theory* to change. Our meta-thesis—our explanation of the nonconformance—is that the standard went too far too fast. To get the definitions right, we need to precisely understand the computational relationship between Classical Planning and Temporal Planning. Slightly more specifically, we must, to have a hope of conformance, formulate Temporal Planning as *relative* to Classical Planning: that is, after all, how the engineering/research will proceed. In other words, an absolutely crucial aspect of our treatment throughout the dissertation is the pains we took to study Temporal Planning as 'orthogonal' to Classical Planning.

Such understanding—When is Temporal Planning *really* Temporal?—then, in so many words, has been our mission. The insights gleaned from practice that we have constructed greatest theoretical support for are:

The first two obstacles to generalizing classical planning technique to time are, in order, deadlines *and* decomposition.

Deadlines. The significance of *deadlines* is that *fastest* is computationally more complicated than *cheapest*: perhaps counter-intuitively, Single-Objective Search is not strong enough, theoretically speaking, to support finding fastest plans. So the first significant hurdle is Multi-Objective Search. To capture this lesson we precisely formulated and studied Conservative Temporal Planning.

Decomposition. The significance of *decomposition* is maddeningly simple: planning is combinatorial. Details are the enemy. Consider that it is a great art to abstract away the irrelevant (the art is to *correctly* identify the relevant): an art not

even close to mastered by present classical planning technique. To capture this lesson we precisely formulated and studied Interleaved Temporal Planning.

Durative Effects or Causally Primitive Actions. For a (related) specific, concrete, lesson: we consider the standard's lack of durative effects to be a mistake. That is, "(over all . . .)" effects ought to be legal (. . . and the semantics should be as for our all-parts). The reason is just that, otherwise, models will tend to contain twice as many primitives as necessary. (Without durative effects one cannot directly state causally primitive durative actions, which are the bread and butter of Conservative Temporal Planning.) The computational advantage had by disrespecting the intended semantics of *decomposition* has proven itself, in practice, to be too great to ignore.

Situations. More abstractly, a crucial principle that sets our investigation apart is that our formal semantics are precisely and explicitly reduced to state transition systems (and, for that matter, only slightly less explicitly, to situation calculus). In particular we explicitly define temporal situations for each form of temporal planning considered (and prove that the definitions are correct). The closest the standard comes is through a *one-way* mapping into a particular state transition system; were the mapping two-way, then working backwards from the vertices would give a correct definition of temporal situation. As-is, correctly defining the situations of temporal PDDL is far from trivial (meaning: nobody has done so). That our investigation possesses such *two-way* mappings+definitions+theorems is a significant improvement.

The Point. Chapter 2 gets every little detail *right*: for that value of "right" defined by "requires minimal reengineering of a classical planner". The key insight is that classical planners simply cannot understand *concurrency*. So, in order to understand the relationship of temporal planning to classical planning, the aspect to focus on is how the definition of concurrency ultimately gets twisted into reasoning about only sequences.

6.2 LIMITATIONS

The work is—*of course*—far from without flaws. Inasmuch as we have made progress in improving upon temporal PDDL: we can only expect future work will find fault here, too.

Indeed, we should hope that the distinctions made here eventually become insignificant as the state-of-the-art advances. *Eventually* we might hope that AI becomes solved, which would certainly suffice.

Perhaps a day will come (in the rather less distant future) when the present version of temporal PDDL will prove to be the 'right' perspective to take upon the then state-of-the-art. An alternative, and more likely, hypothesis takes PDDL+ as candidate [70].

Several of the high level points surely lack adequate technical support; for example, the correct form of Section 5.3.4 is obviously to 'just implement' and empirically evaluate. Which is no small task: there is much more to implementing an effective planner than merely setting up a reasonable search-tree!

⋮

Such aside: Several of the deliberate limitations are interesting to call attention to. Specifically all of the following are intimately intertwined:

- Our treatment cannot be understood independently of the standard [71].
- We skip formalizing syntax.
- There is no mechanism for permitting concurrency of additive effects.
- The intermediate language used in the proof of Theorem 4.12 is more compelling than ITP itself.

- "The Feature" from a domain modeling perspective is almost surely *exogenous/trajectory events+constraints*.

Consider that a particularly compelling source of real world temporal planning problems involve complex synchronization with numerous externalities. (Other agents, star visibility, solar/wind power availability, *et cetera*.) These can of course be compiled, quite unnaturally, into just 'voluntary' actions by exploiting required concurrency. However, there are two compelling reasons not to. For one thing, a heuristic should exploit the fact that such things are not actually voluntary. More importantly, to compile them out still needs a higher level language to have expressed them in the first place. As far as the syntax of a directly useful higher level language goes [187], note that, for example: the right way to describe a temporal planning problem is to state "permitted assumptions and desired constraints", both over whole timelines. That is in contrast to the tired "initial situation and goal expression" idiom inherited from classical planning.

Stepping back, the point is that we have deliberately and sharply curtailed our treatment of planning language. We fully expect that generalization is necessary to achieve direct relevance to real-world problems. So for example, consider additive effects. The right default, we argue, is to prohibit concurrency of such. But there are certainly real world problems where we *would* like to permit concurrency of additive effects. To do so requires two generalizations, firstly, some syntax would need to be cooked up for distinguishing. Secondly, and far more involved, the locking protocol would need to be extended to a third type of lock: a shareable write lock. Doing so in the right way calls for a lot of legwork: rebuild the impacted theory.

6.3 REVIEW OF THE DISSERTATION

The Thesis. Our broad aim was to develop a deeper understanding of the relationship between temporal and classical planning. The thesis is that *concurrency* is "The Feature" best characterizing that relationship. Then *the* lesson, in other words, is that we ought to distinguish between (at least):

- Sequential Planning: forbid concurrency.
- Conservative Temporal Planning: permit concurrency only for optimization.
- Interleaved Temporal Planning: permit causally required concurrency.

We constructed justification from two directions. From theory, each language is computationally more general than the preceding. From practice, each language captures a portion of the state-of-the-art.

Chapter 2. We gave thorough formal and intuitive accounts of these three kinds of temporal planning. Sequential Planning just emphasizes the form of the plans of classical planners: sequences of actions. In Conservative Temporal Planning (CTP) we extend the problem 'syntax' by durations and deadlines; the plan 'syntax' acquires dispatch-times. In Interleaved Temporal Planning (ITP) we extend the 'syntax' by having actions decompose into parts: one psuedo-part standing for the whole, a starting part, and an ending part. Each such primitive part acts as an action of CTP; so, for example, each has a strictly positive, constant, duration. Meanwhile, the plan 'syntax' remains the same in the sense that plans, called effect-schedules, consist of dispatches of primitives.

Chapter 3. We continued by building up foundational theory. Three particular intuitions are quite useful in practice.

Reduce We explicitly walked through the implementations of complete (and correct) brute-force approaches. (Which informs implementing more sophisticated approaches.)

Reschedule We proved that precise timing decisions are of no consequence. Such is crucial, otherwise there would be little hope to coming from classical planning; the right field would be *control theory*.

Reorder We proved that ordering nonmutex primitives is, as it should be, irrelevant. Such is crucial, as we lose effective forms of duplicate elimination in temporal planning; duplicate elimination has the effect of avoiding overly wasteful exploration of pointless ordering decisions.

Two results are especially significant so far as conclusions go.

Theorem 3.17, the formalization of reduction for CTP, establishes that Conservative Temporal Planning Problems and Sequential Planning Problems both reduce into the *same* underyling state-space. The difference is that finding *fastest* plans needs the techniques of Multi-Objective Search (rather than Single-Objective Search as for cheapest plans).

Theorem 3.21, the formalization of reordering for ITP, completes the lion's share of automatically exploiting absences of required concurrency. In Chapter 5 we completed that promise.

Next we turned our attention to applying the theory to analysis of the formal *languages* and *algorithms* underlying the state-of-the-art.

Chapter 4. In the end, we are able to conclude that: Taking Conservative Temporal Planning and Interleaved Temporal Planning as representatives of the temporal state-of-the-art is reasonable enough.

In particular it is useful to note that whether actions really are, in terms of causality, *compound* is enough to characterize expressibility of required concurrency. Firstly that is because such is the essential distinction between our suggested representative languages. Secondly, expressibility of required concurrency may be exploited to work around most other differences in syntax and semantics between actual systems.

Chapter 5. We demonstrated that Conservative Temporal Planning poses little difficulty to algorithm designers. In particular merely integrating First-Fit into a classical planner already attains completeness and systematicity for Conservative Temporal Planning *sans* deadlines. For deadlines/duration-optimality it is enough to disable duplicate state elimination; of course it is rather more promising to replace that with something reasonable. We covered such a replacement: an implementation of the reordering intuition formalized and proven in Chapter 3.

Then, as long as classical planning research keeps up its pace, those results close the book on CTP. So next we considered attempts at 'state-space' Interleaved Temporal Planning.

We (once more) covered the incompleteness result for Decision Epoch Planners [43, 147]. We extended that by a nonsystematicity result, in part to further undermine the algorithm style. In a roundabout fashion the nonsystematicity result also goes towards supporting (!) the style. Long story short we concluded that Decision Epoch Planners are a fundamentally local search approach to tempo-

ral planning; hence potentially quite reasonable as long as said insight is properly appreciated.

That said, it of course remains valuable to develop a theoretically pristine approach. The key insight is to change the design objective from 'state-space' to *forward-chaining*: the motivation from leveraging classical planning technique remains as compelling as ever.

Then the solution is just to forward-chain through effect-sequences: employ *temporal lifting* to delay choosing the missing dispatch-times. (From Chapter 3, it is possible to do so by building up just Simple Temporal Networks rather than, say, Mixed Integer Linear Programs; for generalizations of ITP one might very well end up with the latter instead, which has little impact on the promise of the approach [37].) So we easily proved completeness and systematicity for such Temporally Lifted Planners, Theorem 5.13.

With slightly more legwork, we also covered an implementation of the reordering intuition, Theorem 5.16. The last is quite promising towards the goal of automatically simplifying to CTP on any ismorphic image thereof; we sketched some practical approaches to narrowing the gap.

6.4 SUMMARY

A decade of temporal planning practice offers us several key lessons. The unifying theme is: *Leverage Classical Planning*. Piecing together, and expanding upon, those particulars is, in abstract, the technical work of the dissertation. To highlight a technical contribution: our Theorem 5.16 shows great promise towards automatically exploiting *absences* of causally required concurrency; the significance of doing so is under-appreciated by the state-of-the-art.

In carrying out our technical work, it became clear that the standard could benefit from revision; two forms of temporal planning have proven to be (in practice) too computationally distinct from one another, and too important in their own right, to be lumped together. Roughly, the dissertation is that revision. Accurately, the dissertation constitutes a detailed justification: changing a standard cannot be pursued lightly.

It is fascinating to observe that, from far enough away, every justification is:

Concurrency is "The Feature".

REFERENCES

[1] Mitchell Ai-Chang, John L. Bresina, Leonard Charest, Adam Chase, Jennifer Cheng jung Hsu, Ari K. Jónsson, Bob Kanefsky, Paul H. Morris, Kanna Rajan, Jeffrey Yglesias, Brian G. Chafin, William C. Dias, and Pierre F. Maldague. MAPGEN: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004. MAPGEN.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] James F. Allen. Planning as temporal reasoning. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *KR*, pages 3–14. Morgan Kaufmann, 1991.

[4] James F. Allen and Patrick J. Hayes. Moments, points in an interval-based temporal logic. *Computational Intelligence*, 5:225–238, 1989.

[5] Fahiem Bacchus. Subset of PDDL for the AIPS2000 planning competition draft 1. `http://www.cs.toronto.edu/aips2000/pddl-subset.ps`, 2000.

[6] Fahiem Bacchus and Michael Ady. Planning with resources and concurrency: A forward chaining approach. In Nebel [159], pages 417–424. TLPLAN.

[7] Fahiem Bacchus, Henry Kautz, David E. Smith, Derek Long, Hector Geffner, and Jana Koehler, editors. *Proceedings of the AIPS-00 Planning Competition*, June 2000. IPC 2000.

[8] Christer Bäckström. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research (JAIR)*, 9:99–137, 1998.

[9] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. In Ruzena Bajcsy, editor, *IJCAI*, pages 1430–1435. Morgan Kaufmann, 1993.

[10] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995.

[11] Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In Boddy et al. [18], pages 26–33.

[12] Javier Barreiro, Matthew Boyce, Minh Binh Do, Jeremy Frank, Michael Iatauro, Tatiana Kichkaylo, Paul Morrisand, James Ong, Emilio Remolina, Tristan Smith, and David E. Smith. EUROPA: A platform for ai planning, scheduling, constraint programming, and optimization. In *Proceedings of the 4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*, 2012. EUROPA.

[13] J. Benton, Kartik Talamadupula, Patrick Eyerich, Robert Mattmüller, and Subbarao Kambhampati. G-value plateaus: A challenge for planning. In Brafman et al. [22], pages 259–262.

[14] Sara Bernardini and David E. Smith. Automatic synthesis of temporal invariants. In Michael R. Genesereth and Peter Z. Revesz, editors, *SARA*. AAAI, 2011.

[15] Susanne Biundo and Maria Fox, editors. *Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, September 8–10, 1999, Proceedings*, volume 1809 of *Lecture Notes in Computer Science*. Springer, 2000.

[16] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence (AIJ)*, 90(1-2):281–300, 1997. GraphPlan.

[17] Mark Boddy, Amedeo Cesta, and Stephen Smith, editors. *Proceedings of the Workshop on Integrating Planning into Scheduling (WIPIS, at ICAPS)*, Whistler, British Columbia, Canada, June 2004. AAAI.

[18] Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors. *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, Providence, Rhode Island, USA, September 2007. AAAI.

[19] Mark Boddy, Johnathan Gohde, Thomas Haigh, and Steven A. Harp. Course of action generation for cyber security using classical planning. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *ICAPS*, pages 12–21, Monterey, California, USA, June 2005. AAAI.

[20] Blai Bonet and Hector Geffner. Heuristic search planner 2.0. *AI Magazine*, 22(3):77–80, 2001.

[21] Craig Boutilier, editor. *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11–17, 2009*, 2009.

[22] Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors. *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, Toronto, Ontario, Canada, May 2010. AAAI.

[23] Wolfram Burgard and Dan Roth, editors. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7–11, 2011*. AAAI Press, 2011.

[24] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence (AIJ)*, 69(1-2):165–204, 1994.

[25] Franck Cassez, Thomas A. Henzinger, and Jean-François Raskin. A comparison of control problems for timed and hybrid systems. In Claire Tomlin and Mark R. Greenstreet, editors, *HSCC*, volume 2289 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2002.

[26] Amedeo Cesta and Cristiano Stella. A time and resource problem for planning architectures. In Steel and Alami [190], pages 117–129.

[27] Gregory J. Chaitin. On the simplicity and speed of programs for computing infinite sets of natural numbers. *JACM*, 16(3):407–422, 1969.

[28] David Chapman. Planning for conjunctive goals. *Artificial Intelligence (AIJ)*, 32(3):333–377, 1987.

[29] Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research (JAIR)*, 26:323–369, 2006.

[30] Yixin Chen, You Xu, and Guohui Yao. Stratified planning. In Boutilier [21], pages 1665–1670.

[31] Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors. *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS)*, Breckenridge, Colorado, USA, April 2000. AAAI.

[32] Steve A. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. ASPEN — automated planning and scheduling for space mission operations. In *in Space Ops*, 2000. ASPEN.

[33] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[34] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[35] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Extending the use of inference in temporal planning as forwards search. In Gerevini et al. [83].

[36] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In Brafman et al. [22], pages 42–49. POPF.

[37] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research (JAIR)*, 44:1–96, 2012. COLIN.

[38] Andrew Coles, Maria Fox, Keith Halsey, Derek Long, and Amanda Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence (AIJ)*, 173(1):1–44, 2009. CRIKEY-SHE.

[39] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In Fox and Gomes [68], pages 892–897. CRIKEY3.

[40] Ken Currie and Austin Tate. O-plan: The open planning architecture. *Artificial Intelligence (AIJ)*, 52(1):49–86, 1991. O-PLAN.

[41] William Cushing, J. Benton, and Subbarao Kambhampati. Cost based search considered harmful. In Ariel Felner and Nathan R. Sturtevant, editors, *SOCS*. AAAI Press, 2010.

[42] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. When is temporal planning really temporal? In Veloso [199], pages 1852–1859.

[43] William Cushing, Daniel S. Weld, Subbarao Kambhampati, Mausam, and Kartik Talamadupula. Evaluating temporal planning domains. In Boddy et al. [18], pages 105–112.

[44] Ramon López de Mántaras and Lorenza Saitta, editors. *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22–27, 2004*. IOS Press, 2004.

[45] Thomas Dean, editor. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 — August 6, 1999. 2 Volumes, 1450 pages*. Morgan Kaufmann, 1999.

[46] Thomas Dean, R. James Firby, and David P. Miller. Hierarchical planning involving deadlines, travel time, and resources. *Computational Intelligence*, 4:381–398, 1988. FORBIN.

[47] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence (AIJ)*, 49(1-3):61–95, 1991.

[48] Rina Dechter and Judea Pearl. The optimality of a* revisited. In Michael R. Genesereth, editor, *AAAI*, pages 95–99. AAAI Press, 1983.

[49] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[50] Edsger W. Dijkstra. On the teaching of programming, i. e. on the teaching of thinking. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 1–10, Marktoberdorf, Germany, July–August 1975. Springer.

[51] Edsger W. Dijkstra. On the cruelty of really teaching computing science. Circulated Privately, December 1988.

[52] Minh Binh Do and Subbarao Kambhampati. Solving planning-graph by compiling it into CSP. In Chien et al. [31], pages 82–91. GPCSP.

[53] Minh Binh Do and Subbarao Kambhampati. Improving temporal flexibility of position constrained metric temporal plans. In Giunchiglia et al. [91], pages 42–51.

[54] Minh Binh Do and Subbarao Kambhampati. SAPA: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research (JAIR)*, 20:155–194, 2003. SAPA.

[55] Brian Drabble and Austin Tate. The use of optimistic and pessimistic resource profiles to inform search in an activity based planner. In Hammond [94], pages 243–248. O-PLAN2.

[56] Johann Dréo, Pierre Savéant, Marc Schoenauer, and Vincent Vidal. Divide-and-Evolve: The marriage of Descartes and Darwin. In García-Olaya et al. [76]. IPC 2011.

[57] Stefan Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:195–238, 2003. MIPS-CONSERVATIVE.

[58] Stefan Edelkamp and Malte Helmert. MIPS: The model-checking integrated planning system. *AI Magazine*, 22(3):67–72, 2001. MIPS.

[59] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universitt Freiburg, Institut fr Informatik, 2004. PDDL 2.2.

[60] Stefan Edelkamp and Jörg Hoffmann, editors. *Proceedings of the 2004 Competition, Deterministic Part*, June 2004. IPC 2004.

[61] Stefan Edelkamp and Peter Kissmann. Optimal symbolic planning with action costs and preferences. In Boutilier [21], pages 1690–1695. GAMER.

[62] A. El-Kholy and Barry Richards. Temporal and resource reasoning in planning: The parcPLAN approach. In Wolfgang Wahlster, editor, *ECAI*, pages 614–618. John Wiley and Sons, Chichester, 1996. PARCPLAN.

[63] Kutluhan Erol, James A. Hendler, and Dana S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

[64] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In Gerevini et al. [83]. TFD.

[65] Eric Fabre, Loig Jezequel, Patrik Haslum, and Sylvie Thiébaux. Cost-optimal factored planning: Promises and pitfalls. In Brafman et al. [22], pages 65–72.

[66] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence (AIJ)*, 2(3/4):189–208, 1971. STRIPS.

[67] Sarah Finney, Natalia H. Gardiol, Leslie Pack Kaelbling, and Tim Oates. Learning with deictic representation, 2002.

[68] Dieter Fox and Carla P. Gomes, editors. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13–17, 2008*. AAAI Press, 2008.

[69] Maria Fox and Derek Long. Utilizing automatically inferred invariants in graph construction and search. In Chien et al. [31], pages 102–111.

[70] Maria Fox and Derek Long. PDDL+: Modelling continuous time-dependent effects. In *In Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002. PDDL+.

[71] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003. PDDL 2.1.

[72] Maria Fox, Derek Long, and Keith Halsey. An investigation into the expressive power of PDDL2.1. In de Mántaras and Saitta [44], pages 328–342.

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

[74] Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. ConGolog, sin trans: Compiling ConGolog into basic action theories for planning and beyond. In Gerhard Brewka and Jérôme Lang, editors, *KR*, pages 600–610. AAAI Press, 2008.

[75] Alfredo Gabaldon. Compiling control knowledge into preconditions for planning in the situation calculus. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1061–1066. Morgan Kaufmann, 2003.

[76] Angel García-Olaya, Sergio Jiménez, and Carlos Linares López, editors. *Proceedings of the 2011 International Planning Competition, Deterministic Track*, June 2011. IPC 2011.

[77] Antonio Garrido, Eva Onaindía, and Federico Barber. Time-optimal planning in temporal problems. In *ECP*, pages 397–402, 2001. TPSYS.

[78] B. Cenk Gazen and Craig A. Knoblock. Combining the expressivity of UCPOP with the efficiency of GraphPlan. In Steel and Alami [190], pages 221–233.

[79] Hector Geffner. Functional STRIPS: A more flexible language for planning and problem solving. In *Logic-Based Artificial Intelligence, Jack Minker (Ed.)*. Kluwer, 2000. Inspiration for PDDL 3.1.

[80] Hector Geffner. PDDL 2.1: Representation vs. computation. *Journal of Artificial Intelligence Research (JAIR)*, 20:139–144, 2003.

[81] Alfonso Gerevini, Yannis Dimopoulos, Patrik Haslum, and Alessandro Saetti, editors. *Proceedings of the Fifth International Planning Competition, Deterministic Part*, June 2006. IPC 2006.

[82] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence (AIJ)*, 173(5-6):619–668, 2009. PDDL 3.0, IPC 2006.

[83] Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS)*, Thessaloniki, Greece, September 2009. AAAI.

[84] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. An approach to temporal planning and scheduling in domains with predictable exogenous events. *Journal of Artificial Intelligence Research (JAIR)*, 25:187–231, 2006. LPG-TD.

[85] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Temporal planning with problems requiring concurrency through action graphs and local search. In Brafman et al. [22], pages 226–229. LPG-INTERLEAVED.

[86] Alfonso Gerevini and Ivan Serina. LPG: A planner based on local search for planning graphs with action costs. In Ghallab et al. [89], pages 13–22. LPG-SEQUENTIAL.

[87] Alfonso Gerevini, Ivan Serina, Alessandro Saetti, and Sergio Spinoni. Local search techniques for temporal planning in LPG. In Giunchiglia et al. [91], pages 62–72. LPG 1.2.

[88] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

[89] Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors. *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS)*, Toulouse, France, April 2002. AAAI.

[90] Malik Ghallab and Hervé Laruelle. Representation and control in IxTeT, a temporal planner. In Hammond [94], pages 61–67. IxTeT.

[91] Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors. *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS)*, Trento, Italy, June 2003. AAAI.

[92] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1993.

[93] Keith Halsey, Derek Long, and Maria Fox. Multiple relaxations in temporal planning. In de Mántaras and Saitta [44], pages 1029–1030. CRIKEY.

[94] Kristian J. Hammond, editor. *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS)*, University of Chicago, Chicago, Illinois, USA, June 1994. AAAI.

[95] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968. A^*: see Dechter 1983.

[96] Patrik Haslum. Improving heuristics through relaxed search — an analysis of TP4 and HSP^*_a in the 2004 planning competition. *Journal of Artificial Intelligence Research (JAIR)*, 25:233–267, 2006. HSP^*_a.

[97] Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to h^m. In Gerevini et al. [83].

[98] Patrik Haslum. An independent validator for PDDL, 2012.

[99] Patrik Haslum, Blai Bonet, and Hector Geffner. New admissible heuristics for domain-independent planning. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 1163–1168. AAAI Press / The MIT Press, 2005.

[100] Patrik Haslum and Hector Geffner. Heuristic planning with time and resources. In *ECP*, pages 121–132, 2001. TP4.

[101] Malte Helmert. Decidability and undecidability results for planning with numerical state variables. In Ghallab et al. [89], pages 44–53.

[102] Malte Helmert. A planning heuristic based on causal graph analysis. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 161–170. AAAI, 2004.

[103] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006. FD.

[104] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence (AIJ)*, 173(5-6):503–535, 2009.

[105] Malte Helmert, Minh Binh Do, and Ioannis Refanidis, editors. *Proceedings of the 2008 International Planning Competition, Deterministic Track*, May 2008. IPC 2008.

[106] Malte Helmert and Hector Geffner. Unifying the causal graph and additive heuristics. In Rintanen et al. [178], pages 140–147.

[107] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Boddy et al. [18], pages 176–183.

[108] Malte Helmert and Gabriele Röger. How good is almost perfect? In Fox and Gomes [68], pages 944–949.

[109] Thomas A. Henzinger. It's about time: Real-time logics reviewed. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 439–454. Springer, 1998.

[110] Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In Nancy A. Lynch and Bruce H. Krogh, editors, *HSCC*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2000.

[111] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *J. Comput. Syst. Sci.*, 57(1):94–124, 1998.

[112] Sarah L. Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Langford B. White. Planning via petri net unfolding. In Veloso [199], pages 1904–1911.

[113] Sarah L. Hickmott and Sebastian Sardiña. Optimality properties of planning via petri net unfolding: A formal analysis. In Gerevini et al. [83].

[114] Jörg Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In Ghallab et al. [89], pages 92–100.

[115] Jörg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)*, 20:291–341, 2003. MetricFF.

[116] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001. FF.

[117] Melvin Randall Holmes. *Elementary Set Theory with a Universal Set*, volume 10 of *Cahiers du Centre de Logique*. Academia, Grand'Place 29, 1348 Louvain-la-Neuve, Belgium, 1998. 2012 draft: `http://math.boisestate.edu/~holmes/holmes/head.pdf`.

[118] Werner Horn, editor. *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20–25, 2000*. IOS Press, 2000.

[119] Richard Howey, Derek Long, and Maria Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *ICTAI*, pages 294–301. IEEE Computer Society, 2004. VAL.

[120] Yanmei Hu, Minghao Yin, and Dunbo Cai. On the discovery and utility of precedence constraints in temporal planning. In Burgard and Roth [23]. LMTD.

[121] Ari K. Jónsson, Paul H. Morris, Nicola Muscettola, Kanna Rajan, and Benjamin D. Smith. Planning in interplanetary space: Theory and practice. In Chien et al. [31], pages 177–186. HSTS.

[122] Frederick P. Brooks Jr. The mythical man-month: After 20 years. *IEEE Software*, 12(5):57–60, 1995.

[123] Subbarao Kambhampati. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in graphplan. *Journal of Artificial Intelligence Research (JAIR)*, 12:1–34, 2000.

[124] Subbarao Kambhampati, Craig A. Knoblock, and Qiang Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence (AIJ)*, 76(1-2):167–238, 1995.

[125] Richard M. Karp. Combinatorics, complexity, and randomness. *Commun. ACM*, 29(2):97–109, 1986.

[126] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Boutilier [21], pages 1728–1733.

[127] Michael Katz, Jörg Hoffmann, and Malte Helmert. How to relax a bisimulation? In McCluskey et al. [150].

[128] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In Dean [45], pages 318–325. BlackBox.

[129] Bharat Ranjan Kavuluri. Extending temporal planning for the interval class. In García-Olaya et al. [76]. Sharaabi.

[130] Peter Kissmann and Stefan Edelkamp. Improving cost-optimal domain-independent symbolic planning. In Burgard and Roth [23]. GAMER.

[131] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence (AIJ)*, 68(2):243–302, 1994.

[132] Jana Koehler. Planning under resource constraints. In *ECAI*, pages 489–493, 1998. IPP with resources.

[133] Jana Koehler and Jörg Hoffmann. On the instantiation of ADL operators involving arbitrary first-order formulas. In *PuK*, 2000. adl2strips.

[134] Jonas Kvarnström, Patrick Doherty, and Patrik Haslum. Extending TALplanner with concurrency and resources. In Horn [118], pages 501–505. TALPLANNER.

[135] Jonas Kvarnström, Fredrik Heintz, and Patrick Doherty. A temporal logic-based planning and execution monitoring system. In Rintanen et al. [178], pages 198–205. TALPLANNER.

[136] Leslie Lamport. Buridan's principle. *Foundations of Physics*, 42:1056–1066, 2012. Written in 1984; self-published online.

[137] Hui X. Li. *Kongming: A Generative Planner for Hybrid Systems with Temporally Extended Goals*. PhD thesis, Massachusetts Institute of Technology, 2010. KONGMING.

[138] Hui X. Li and Brian C. Williams. Generative planning for hybrid systems based on flow tubes. In Rintanen et al. [178], pages 206–213. KONGMING applied to Aquatic Science.

[139] Vladimir Lifschitz. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, Timberline, Oregon, USA, 1987. Morgan Kaufmann.

[140] Iain Little and Sylvie Thiébaux. Concurrent probabilistic planning in the graphplan framework. In Long et al. [144], pages 263–273. PROTTLE.

[141] Derek Long and Maria Fox, editors. *Proceedings of the 2002 Competition, Deterministic Part*, June 2002. IPC 2002.

[142] Derek Long and Maria Fox. Exploiting a graphplan framework in temporal planning. In Giunchiglia et al. [91], pages 52–61. LPGP.

[143] Derek Long, Henry A. Kautz, Bart Selman, Blai Bonet, Hector Geffner, Jana Koehler, Michael Brenner, Jörg Hoffmann, Frank Rittinger, Corin R. Anderson, Daniel S. Weld, David E. Smith, and Maria Fox. The AIPS-98 planning competition. *AI Magazine*, 21(2):13–33, 2000. IPC 1998.

[144] Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors. *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS)*, Cumbria, UK, June 2006. AAAI.

[145] Peter Lynds. Zeno's paradoxes: A timely solution. *PhilSci Archive*, 2003. `http://philsci-archive.pitt.edu/1197/`.

[146] Frederic Maris and Pierre Régnier. TLP-gp: New results on temporally-expressive planning benchmarks. In *ICTAI*, pages 507–514. IEEE Computer Society, 2008. TLP-GP.

[147] Mausam and Daniel S. Weld. Challenges for temporal planning with uncertain durations. In Long et al. [144], pages 414–417.

[148] Mausam and Daniel S. Weld. Planning with durative actions in stochastic domains. *Journal of Artificial Intelligence Research (JAIR)*, 31:33–82, 2008.

[149] David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In Thomas L. Dean and Kathleen McKeown, editors, *AAAI*, pages 634–639. AAAI Press / The MIT Press, 1991. SNLP.

[150] Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors. *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS)*, Atibaia, São Paulo, Brazil, June 2012. AAAI.

[151] Drew McDermott. Reasoning about autonomous processes in an estimated-regression planner. In Giunchiglia et al. [91], pages 143–152. Optop.

[152] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel S. Weld, and David Wilkins. PDDL — the planning domain definition language — version 1.2. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998. PDDL 1.2.

[153] Steven Minton, John L. Bresina, and Mark Drummond. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research (JAIR)*, 2:227–262, 1994.

[154] David J. Musliner, Michael J. S. Pelican, Robert P. Goldman, Kurt D. Krebsbach, and Edmund H. Durfee. The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, pages 43–48. AAAI, 2008. CIRCA.

[155] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

[156] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In Dean [45], pages 968–975.

[157] Bernhard Nebel. Compilation schemes: A theoretical tool for assessing the expressive power of planning formalisms. In Wolfram Burgard, Thomas Christaller, and Armin B. Cremers, editors, *KI*, volume 1701 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 1999.

[158] Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research (JAIR)*, 12:271–315, 2000.

[159] Bernhard Nebel, editor. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4–10, 2001*. Morgan Kaufmann, 2001.

[160] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In Nebel [159], pages 459–466. RePOP.

[161] XuanLong Nguyen, Subbarao Kambhampati, and Romeo Sanchez Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence (AIJ)*, 135(1-2):73–123, 2002. AltAlt.

[162] Raz Nissim, Jörg Hoffmann, and Malte Helmert. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In Toby Walsh, editor, *IJCAI*, pages 1983–1990. IJCAI/AAAI, 2011.

[163] The on-line encyclopedia of integer sequences: Sequence A000262. `http://oeis.org/A000262`, 2012.

[164] Héctor Palacios and Hector Geffner. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research (JAIR)*, 35:623–675, 2009.

[165] Judea Pearl. The algorithmization of counterfactuals. *Annals of Mathematics and Artificial Intelligence*, 61(1):29–39, 2011.

[166] Edwin P. D. Pednault. ADL and the state-transition model of action. *J. Log. Comput.*, 4(5):467–512, 1994.

[167] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *KR*, pages 103–114. Morgan Kaufmann, 1992. UCPOP.

[168] J. Scott Penberthy and Daniel S. Weld. Temporal planning with continuous change. In Barbara Hayes-Roth and Richard E. Korf, editors, *AAAI*, pages 1010–1015. AAAI Press / The MIT Press, 1994. ZENO.

[169] Giuseppe Della Penna, Daniele Magazzeni, Fabio Mercorio, and Benedetto Intrigila. UPMurphi: A tool for universal planning on PDDL+ problems. In Gerevini et al. [83]. UPMURPHI.

[170] Willard V. Quine. New foundations for mathematical logic. *The American mathematical monthly*, 44(2):70–80, 1937.

[171] Ioannis Refanidis and Ioannis P. Vlahavas. Heuristic planning with resources. In Horn [118], pages 521–525. GRT.

[172] Ioannis Refanidis and Ioannis P. Vlahavas. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research (JAIR)*, 15:115–161, 2001. GRT.

[173] Raymond Reiter. Natural actions, concurrency and continuous time in the situation calculus. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *KR*, pages 2–13. Morgan Kaufmann, 1996.

[174] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research (JAIR)*, 39:127–177, 2010. LAMA.

[175] Jussi Rintanen. Complexity of concurrent temporal planning. In Boddy et al. [18], pages 280–287.

[176] Jussi Rintanen. Heuristics for planning with SAT and expressive action definitions. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *ICAPS*, Freiburg, Germany, June 2011. AAAI.

[177] Jussi Rintanen and Hartmut Jungholt. Numeric state variables in constraint-based planning. In Biundo and Fox [15], pages 109–121.

[178] Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS)*, Sydney, Australia, September 2008. AAAI.

[179] A. Robinson. *Non-standard analysis*. Princeton University Press, 1996.

[180] Bertrand Russell and Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, 1968.

[181] Bart Selman. Stochastic search and phase transitions: AI meets physics. In *IJCAI*, pages 998–1002. Morgan Kaufmann, 1995.

[182] Murray Shanahan. The event calculus explained. In Michael Wooldridge and Manuela Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430. Springer Berlin / Heidelberg, 1999.

[183] Daniel G. Shapiro and Markus P. J. Fromherz, editors. *Proceedings of the Twenty-Third Conference on Innovative Applications of Artificial Intelligence, August 9–11, 2011, San Francisco, California, USA*. AAAI, 2011.

[184] Ji-Ae Shin and Ernest Davis. Processes and continuous change in a SAT-based planner. *Artificial Intelligence (AIJ)*, 166(1-2):194–253, 2005. TM-LPSAT.

[185] Benjamin D. Smith, Martin S. Feather, and Nicola Muscettola. Challenges and methods in testing the remote agent planner. In Chien et al. [31], pages 254–263. RAX.

[186] David E. Smith. The case for durative actions: A commentary on PDDL2.1. *Journal of Artificial Intelligence Research (JAIR)*, 20:149–154, 2003.

[187] David E. Smith, Jeremy Frank, and William Cushing. The ANML language. In Rintanen et al. [178].

[188] David E. Smith and Daniel S. Weld. Temporal planning with mutual exclusion reasoning. In Dean [45], pages 326–337. TGP.

[189] Biplav Srivastava and Subbarao Kambhampati. Scaling up planning by teasing out resource scheduling. In Biundo and Fox [15], pages 172–186.

[190] Sam Steel and Rachid Alami, editors. *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24–26, 1997, Proceedings*, volume 1348 of *Lecture Notes in Computer Science*. Springer, 1997.

[191] Gerald J. Sussman and Guy L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[192] Gerald Jay Sussman. *A computational model of skill acquisition*. PhD thesis, MIT, 1973.

[193] Austin Tate, Jeff Dalton, and John Levine. O-plan: A web-based AI planning agent. In Henry A. Kautz and Bruce W. Porter, editors, *AAAI/IAAI*, pages 1131–1132. AAAI Press / The MIT Press, 2000. O-PLAN.

[194] Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In defense of PDDL axioms. *Artificial Intelligence (AIJ)*, 168(1-2):38–69, 2005. PDDL 2.2.

[195] Kevin Tierney, Amanda Jane Coles, Andrew Coles, Christian Kroer, Adam M. Britt, and Rune Møller Jensen. Automated planning for liner shipping fleet repositioning. In McCluskey et al. [150].

[196] Menkes van den Briel and Subbarao Kambhampati. OPTIPLAN: Unifying IP-based and graph-based planning. *Journal of Artificial Intelligence Research (JAIR)*, 24:919–931, 2005. OPTIPLAN.

[197] Menkes van den Briel, Subbarao Kambhampati, and Thomas Vossen. Fluent merging: A general technique to improve reachability heuristics and factored planning. In *ICAPS Workshop on Heuristics*, 2007.

[198] Menkes van den Briel, Romeo Sanchez Nigenda, Minh Binh Do, and Subbarao Kambhampati. Effective approaches for partial satisfaction (oversubscription) planning. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 562–569, San Jose, California, USA, July 2004. AAAI Press / The MIT Press. OPTIPLAN, ALTWLT, SAPA^{PS}.

[199] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6–12, 2007*, 2007.

[200] Steven A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Trans. Pattern Anal. Mach. Intell.*, 5(3):246–267, 1983.

[201] Vincent Vidal. YAHSP2: Keep it simple, stupid. In García-Olaya et al. [76]. IPC 2011.

[202] Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence (AIJ)*, 170(3):298–335, 2006. CPT.

[203] Martin Wehrle and Jussi Rintanen. Planning as satisfiability with relaxed $\exists$-step plans. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 244–253. Springer, 2007.

[204] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence (AIJ)*, 22(3):269–301, 1984.

[205] Steven A. Wolfman and Daniel S. Weld. The LPSAT engine & its application to resource planning. In Dean [45], pages 310–317. LPSAT.

[206] David Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evolutionary Computation*, 1(1):67–82, 1997.

[207] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6:12–24, 1990.

[208] Håkan L. S. Younes and Reid G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research (JAIR)*, 20:405–430, 2003. VHPOP.

[209] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, 24(2):73–96, 2003.

APPENDIX A

MATHEMATICS AND AUTOMATED PLANNING BACKGROUND

This appendix is a whirlwind treatment of some of the more unusual or important assumptions/notation underlying the entirety. It is intended only as a reference.

The Standard Temporal Planning Language. Version 2.1 of the Planning Domain Definition Language (PDDL 2.1) is proscriptive rather than descriptive [71], but otherwise our treatment (in Chapter 2) may be taken as roughly equivalent in purpose and, up to PDDL 2.1's third level of expressiveness, roughly equivalent in scope. Our treatment is (meant to be) complementary. Specifically we eliminate (by wholesale replacing its semantics) the obstacles to precisely understanding its syntax from the perspective of <u>state transition systems</u>.

Formatting, Conventions and Emphasis. Emphasis in general is formatted *thusly*. A potential connotation is for foreign language: *exempli gratia*. Another is for quotation: *we* means *you and I* (and hence "our contribution").

Notions whose precise technical meaning are being specifically called out are formatted as <u>foo</u>, chiefly when said definition has yet to appear. An opposing connotation is conveyed with single-quotes (*i.e.*, the technical meaning should be ignored): classical planning is 'feasible'.[1]

Definitions, particularly when less standard, may be highlighted as in: **foo** means nothing in particular. I employ (just for variety) all of the following as perfect synonyms for definition: *means*, *is*, *(precisely) when*, *denotes*, and *written as*. (In contrast to definitions/axioms, reserve *iff* and friends for propositions/theorems.) Plain $x + 0 = x$ could refer to *definition* or *proposition*; to emphasize equality-by-definition write: $x + 0 \coloneqq x$.

Assignment is (dubiously) denoted by $\coloneqq$ as well. Other texts write, for example, $\leftarrow$ to draw the subtle distinction. For natural language (equally dubiously), treat as synonymous all of: *let*, *(re)define*, *assign*, *with*, and *where*.

Variables/fields/accessors/slots are formatted as in: *State*(*Initial*) is the value of the field called *State* in the structure denoted by *Initial*. Any more complicated computation on a structure, for the sake of example consider computing the cost of a path, is formatted as in: $\text{cost}(P) \coloneqq \sum_{e \in E(P)} w(e)$. Of course we may also just *overload* $w(P)$ to mean $\sum_{e \in P} w(e)$ directly.

Context is progressively deemphasized by abbreviating, subscripting, or omitting arguments. For example, vertex i of path P for some index i is preferably written just v, followed closely by v_i. Less preferable are the increasingly formal expressions: $v_{P,i}$, $v_i \in V_P$, $v(P,i)$, and $v(P(i))$. For relevant subexpressions, abbreviation is by shortening to initials as in replacing *State* with S. Irrelevance is abbreviated further, with $\cdot$, as in: $\max(\cdot, \infty) = \infty$.

[1] 'Feasibility' of classical planning means: state of the art classical planners work surprisingly well in practice [76], despite the well-known theory to the contrary [24].

A.1 MATHEMATICAL UNDERPINNINGS

Standard Values and Sets. Booleans, Naturals, Integers, Rationals, and Reals are denoted by $\mathbb{B} = \{\texttt{False}, \texttt{True}\}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. Denote the first k Naturals by $[k] = [1, k] = \{1, 2, \ldots, k\} \subset \mathbb{N}$. The cartesian product is written as in: $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$.

Types. A type is a disambiguating, and optional, annotation that can be applied to expressions to clarify what values and operations are meant [33, 117, 170, 180]. We treat these as just (very well-behaved) sets of values. So we say that $\mathbb{B}$ denotes the set of truth-values (and $\mathbb{R}$ the set of Reals); we really mean though that $\mathbb{B}$ denotes a Boolean algebra (and $\mathbb{R}$ the Real field). For notation, write $\alpha^{\in X}$ to declare that α should be interpreted with respect to type X. Writing $\alpha \in X$ has intuitively the same meaning. A significant difference is that $\alpha^{\in X}$ is undefined when $\alpha \notin X$.

Functions, Sets, and Relations. We sometimes stylize functions using an adaptation of set-builder notation ("function-builder notation"). For example the Fibonnaci numbers are denoted by:

$$\text{Fib} := \left\{ \begin{array}{l} i^{\in\mathbb{N}} \mapsto i \mid i \in [0, 1]; \\ i^{\in\mathbb{N}} \mapsto \text{Fib}(i-2) + \text{Fib}(i-1) \end{array} \right\}.$$

Precise meaning may be had by transliterating into the Scheme adaptation of the λ-calculus [34, 191]. See Figure A.1.

```
(define Fib-Slowly (lambda (i) (cond
        ((and (natural? i) (member '(0 1))) i)
        ((natural? i) (+
                (Fib-Slowly (- i 2))
                (Fib-Slowly (- i 1)))))))
;; not built-in
(define natural? (lambda (i) (and
        (exact? i) (positive? i) (integer? i))))
```

Figure A.1: A terrible way to compute the Fibonnaci numbers.

The type of partial functions from type X to type Y is denoted by $X \to Y$, and the type of total functions by $X \xrightarrow{\text{total}} Y$. A single argument total truth-function, *i.e.*, type $X \xrightarrow{\text{total}} \mathbb{B}$, represents a set. A multiple argument total truth-function, *e.g.*, type $X \times Y \xrightarrow{\text{total}} \mathbb{B}$, represents a relation (*i.e.*, a set of tuples).

Any function $f^{\in X \to Y}$ may be regarded instead as a relation, as in:

$$(y = f(x)) \Rightarrow (f(x, y) = \texttt{True}),$$
$$(y \neq f(x)) \Rightarrow (f(x, y) = \texttt{False}).$$

(For totality, take $f(x, y)$ as false if otherwise undefined.)

A relation is a function when the reverse mapping is possible. Abusively, take the range of a relation to be the image in its last argument (rather than $\mathbb{B}$). *E.g.*: $\mathrm{Rng}(f) \coloneqq \{y \mid f(x, y)\}$. The range of the subrelation $f{\restriction}_Z$ of f domain-restricted to $Z \subseteq \mathrm{Dom}(f)$, *i.e.*, loosely the image of Z under function f, may be written:

$$\begin{aligned} \mathrm{Rng}(f{\restriction}_Z) &= \{y \mid f(x, y) \text{ for some } x \in Z\}, && \text{if } f \text{ is a function then:} \\ &= \{f(x) \mid x \in Z\}, && \text{also written as just:} \\ &= f\,(Z). \end{aligned}$$

The result of remapping (a.k.a. overriding, overwriting, updating, assigning, extending-by) the function $f^{\in X \to Y}$ by the function $g^{\in X' \to Y'}$ is the function that is the extension of g obtained by defaulting to f wherever g is undefined:

$$\begin{aligned} (f \oplus g)^{\in (X \cup X' \to Y \cup Y')} &= \left\{x^{\in \mathrm{Dom}(g)} \mapsto g(x); x^{\in \mathrm{Dom}(f)} \mapsto f(x)\right\}, \\ &= f{\restriction}_{\overline{\mathrm{Dom}(g)}} \cup g, \\ &= f \setminus f{\restriction}_{\mathrm{Dom}(g)} \cup g. \end{aligned}$$

Prototypes and Structures. Structure types are traditionally declared in terms of prototypes, with instances given by unification/substitution. Treat field names of prototypes as accessor functions. For example, “A **simple graph** $G = (V, E)$ consists of its vertices V, a set, and edges E, a symmetric irreflexive binary relation on V.”, declares the type simple graph in terms of the prototypical instance $G = (V, E)$. So declaring some other instance, “$H = (A, B)$ is a simple graph.”, implicitly asserts $V(H) = A$, $E(H) = B$, $B \in \binom{A}{2} \xrightarrow{\text{total}} \mathbb{B}$, and so forth.

Collections. Collections are especially useful for grouping related structures together and are treated similarly in notation. Formally say that collections are total functions, but, write subscripts in preference to function applications. For example, with $i^{\in I}$ the index of a graph in some collection $G = (V, E)_I$ of (simple) graphs indexed by I: preferably write G_i, V_i and E_i for the graph, its vertices, and its edges, rather than writing $G(i)$, $V(G(i))$, and $E(G(i))$. If helpful we can name the iterator, for example: $G = (V_i, E_i)_i$. To disambiguate, write both explicitly: $G = (V_i, E_i)_{i \in I}$.

Sequences. A finite sequence is more or less a collection indexed by a finite total-order; except that sequences are regarded as identical if the only difference is the representation of the total-order. So the canonical index set is just $[n]$ for any n-length sequence. The prototypical X-sequence of length n is written either $\left(S^{\in X}\right)_{[n]}$ or $(S_1, S_2, \ldots, S_n)^{\in X^n}$. The Kleene star of X, $X^* = \bigcup_{k=0} X^k = \left\{\left(S^{\in X}\right)_{[k]} \mid k^{\in \mathbb{Z}} \geq 0\right\}$, denotes the type of arbitrary, but not infinite, length X-sequences. A tuple is just a sequence in the context of fixed length/size. A structure is just a tuple giving better names (than 1st, 2nd, ...) to its elements.

Families and Tuple Types. Collections of sets are also called families and serve, by their cartesian products, to precisely define very particular subtypes of tuples. (In application to planning the set of possible <u>states</u> is conveniently expressed as such.) The cartesian product of a family is given by the set of all tuples whose elements are each of the right type, *i.e.*, each belonging to the same-indexed set of the family. In notation, with $\left(C^{\in \text{Set}}\right)_I$ a family, its cartesian product is written $\bigtimes C = \{(c)_I \mid c_i \in C_i \text{ for all } i \in I\}$, or equivalently $\bigtimes C = \{(c_i \in C_i)_{i \in I} \mapsto \texttt{True}\}$.

Graphs. All graphs actually of interest are at a minimum directed and theoretically could have multiple edges, even loops, between vertices. So say: A **graph** $G = (V, E, R)$ consists of its vertices V, edges E, and single-step transition relation $R^{\in E \xrightarrow{\text{total}} V^2} = \left(u^{\in V}, v^{\in V}\right)_E$, which is an edge-indexed relation on vertices. The transition relation R denotes that each edge in E connects some vertex u to some vertex v, in that order, perhaps redundantly, perhaps even in a loop. That is, asserting $R(e) = (a, b)$ means also that $e \in E$, $u(e) = a$, $v(e) = b$, and denotes that edge e connects vertex a to vertex b. (One thinks of R as a relation proper by writing $R(a, b)$ in preference to $R^{-1}(a, b) \neq \emptyset$.)

A.2 DOMAIN-INDEPENDENT AUTOMATED PLANNING BACKGROUND

In case of dispute between planners: The ultimate notion of *correct* is always by reduction to state transition systems. It is then, perhaps, worthwhile to be extremely clear concerning the relationship to and the nature of the state transition systems that we require. We also examine some of the key inference techniques employed by the state-of-the-art in classical planning.

A.2.1 Syntax, Semantics, and State Transition Systems

(Lack of) Formal Syntax for Examples. Our examples of planning problems usually employ *psuedo*-syntax. In part that is just because employing PDDL-syntax would be (even more) ambiguous. Also we do not wish to encourage implementation, which certainly needs to be explained. Somewhat abstractly, having planners all willing to parse identical syntax is better than ensuring that syntax accurately reflects the semantics ascribed. That is, we consider a plethora of incompatible dialects to be worse than semantic ambiguity. Ultimately we should indeed extend/modify from PDDL-syntax so as to capture the semantic issues investigated here. However, we consider design of syntax to be a serious endeavor in its own right, and so leave the task to the future. Such work should take seriously those issues considered and formalized in at least: (temporal) PDDL, PDDL+, Optop, and ANML [70, 71, 151, 187]. It certainly wouldn't hurt to also draw from real-world systems, *e.g.*: NDDL for EUROPA [12, 73].

It is easy to underestimate the significance of the task. So we note that the very first specification of PDDL—a defining moment in the literature—formalized *only* syntax [152]. McDermott could get away with skipping semantics chiefly because those issues—*i.e.*, for starters, the connections between STRIPS, SAS, and ADL—had already been resolved [9, 66, 78, 139, 166].

(Lack of) Formal Semantics. It is more lucid to be a touch imprecise regarding the several levels of interpretation involved in automated planning. In general it is quite worthwhile to distinguish between them, say for the purpose of successfully automating planning in practice, especially: *(Level 1)* the (meta-)language of planner engineering, *(Level 2)* the formal language(s) of planning systems themselves, and *(Level 3)* the physical interpretation of plans as implemented by executives. But for our purpose it is preferable to conflate these together. For example, we say: "*ActionDefs*(a) is the definition of action a.", rather than:

(Level 1) *ActionDefs*(a) denotes

(Level 2) those axioms of the planner's theory that constrain

(Level 3) the permitted interpretations as a physical action by the executive of

(Level 2) that particular constant action-symbol of the planner's theory referred to

(Level 1) by us, at the moment, with the variable symbol a of the meta-language.

Note that the less precise statements are normally no less accurate, theoretically speaking. Consider the following.

There are several assumptions virtually always made by domain-independent planners that ensure the apparent lack of precision is nonetheless unambiguous. It is beyond commonplace to assert the Unique Names Axiom: (i) distinct names identify distinct objects and distinct objects are distinctly named. That licenses conflations such as "the action a" and "the fluent f" to abbreviate "the action named by $a^{\in \mathrm{Dom}(\mathit{ActionDefs})}$" and "the fluent named by $f^{\in \mathrm{Dom}(\mathit{FluentDefs})}$". The other two axioms that go almost hand in hand are that: (ii) every name is known, and (iii) there are only finitely many names. Taking all three guarantees that planning theories ground into the narrow confines of (the situation calculus embedded within) propositional logic [71]. At that point the theoretical differences between the model-theories and proof-theories of the languages of *domain modeling*, *planning*, and *execution* are all slight enough.

In short, as far as our aims are concerned, our somewhat loose treatment of the various levels of interpretation is of little significance. In general the distinctions are rather more important. For example, real systems must be robust to plan failure. As a trivial prequisite to such, clearly we would have to distinguish the planner's notion of executable from the executive's notion of executable.

State Transition Systems. A state transition system is a model of a machine in terms of an edge-labeled graph, where the vertices denote all the possible states of the machine, and the edges, along with their labels, denote all the possible transitions the machine can make between its states. Somewhat more precisely: At each time the behavior of a state transition system is to traverse some transition (i) labeled by the current input, (ii) leaving the current state, and (iii) arriving at the next state. If no such transition exists then the input is immediately rejected. (We assume determinism, so otherwise the choice is unique.) The input is accepted only if the final state is one of a designated set of accepting states (and otherwise rejected). The set of all accepted input is called the language of the system. The remainder elaborates and ascribes notation; throughout let $M = (V, E, \Sigma, R, \ell, s_0, T)$ denote the state transition system in question.

Definition A.1 (State Transition System)**.** A **state transition system** is the extension of its underlying graph $G = (V, E, R)$ to include: a set of **(transition-)labels** Σ, a **(transition-)labeling** $\left(\ell^{\in\Sigma}\right)_E$, which is a total function ascribing a label to every transition, an **initial state** $s_0{}^{\in V}$, and a set of **accepting states** $T \subseteq V$. A **state** is just a vertex, and a **transition** an edge, of the underlying graph.

A state transition system is **finite**, or may as well be, if its transition relation is finite. Our definitions do not require finiteness.

For simplicity we do require determinism. A state transition system is deterministic when the labels induce transition functions:

Definition A.2 (Induced Transition Functions)**.** Recall that the (indexed) transition relation is the collection specifying the endpoints of each transition: $R = \left(u^{\in V}, v^{\in V}\right)_E$. Define the transition relation R'_a **induced** by any transition-label $a^{\in\Sigma}$ as the restriction of the full transition relation to just those transitions labeled by a, *i.e.*: $R'_a := \mathrm{Rng}(R\restriction_{\ell^{-1}(a)})$. Then M is **deterministic** if each induced transition relation is moreover a function. In notation, M is deterministic if, for all transition-labels $a^{\in\Sigma}$, the transition relation induced by a is equal to its functional part:

$$R'_a = \{x \mapsto y \mid \text{ for all } e \text{ such that } (\ell_e = a \text{ and } u_e = x) \text{ then } (v_e = y)\}. \tag{A.1}$$

If so, then each R'_a is called the a-induced **transition function**, a.k.a. the transition function of a.

Definition A.3 (Behavior)**.** Consider any **input** $(w)_{[n]}{}^{\in\Sigma^*}$, a label-sequence, and write $(s)_{[0,m]}{}^{\in V^*}$ for the behavior, a state-sequence, of M given w. The **behavior** of M given w is defined by iteratively applying the induced transition functions, as follows. For each **time** $i^{\in[m]}$ with $m \leq n$ the point after which time stops, let $a = w_i$ be the **current label**, and $s = s_{i-1}$ the **current state**. Define the **result** thus far as just $\mathrm{Result}(w\restriction_{[i-1]}, s_0) := s$ the current state. The system M immediately **rejects** if none of the edges leaving s are labeled by a, *i.e.*, if $R'_a(s)$ is undefined. Time stops whenever M rejects or accepts. If otherwise, then define the **next state**:

$$s_i := R'_a(s_{i-1}). \tag{A.2}$$

Regard the result function as encompassing all choices of initial state: say $\mathrm{Result}_M = \bigcup \mathrm{Result}_{M'}$ for M' only differing in s_0. In particular observe that the full reachability relation, as it is called when viewed as a relation, is transitive. That is, a determination of $\mathrm{Result}(a, s)$ for all labels a and states s likewise determines $\mathrm{Result}(w, s)$ for all label-sequences w. Equivalently a determination of R'_a for each a determines all of Result().

Definition A.4 (Language)**.** A state transition system M **accepts** input w, and w is then called a **word**, when the final result $\mathrm{Result}(w, s_0)$ is one of the accepting states: $\mathrm{Result}(w, s_0) \in T$ holds (and, firstly, is defined).

The **language** L of M is its set of words:

$$L := \{w \mid \mathrm{Result}(w, s_0) \in T\}. \tag{A.3}$$

The differences between a state transition system and its underlying graph, besides vocabulary, are relatively minor. For example its language is just the set of sequences of edge-labels one obtains by walking through the underlying graph between the initial and target vertices. In practice usually at most the minimal walks —the paths—are of interest. So computing the 'interesting' walks/words is intimately related to the Single-Source Cheapest-Path Problem [49].

Similarly the connection to *planning* is by and large a shift in vocabulary (labels are actions, paths are plans, *etc.*). There is one absolutely crucial computational difference though: planning problems are exponentially smaller than their corresponding transition systems. In particular the graphs that serve to arbitrate the formal semantics of variations on planning are *strictly figurative*; we assume, with excellent cause, that to write out such graphs is to 'die' (computationally speaking).

A.2.2 Inference: Compilation, Reachability Heuristics, and Landmarks

The aim here is to formalize 'efficient' compilation between forms of planning along the lines that the target planner carries out 'enough' in the way of sophisticated inference so as to render tolerable the otherwise atrocious computational consequences (of taking a compilation approach to planning). So, in slight contrast with related work, we make explicit the point that whether a compilation is 'good' fundamentally depends on the target 'platform' being compiled to [10, 157]. An improvement is that theory of the following kind can, for example, support the observation that both the exponential and polynomial approaches to compiling out conditional effects can be effective in practice. Nebel's theory, in particular, cannot distinguish between meaningful and meaningless increases in plan/solution size.

A.2.2.1 Compilation

For the moment let notation be as follows:

- all plans in problem P of source language S and problem $Q = \sigma(P)$ of target language T: $X = \mathit{Actions}(P)^*$ and $Y = \mathit{Actions}(Q)^*$,
- the total map from source to target: $\sigma^{\in S \overset{\text{total}}{\longrightarrow} T}$,
- the partial map from target to source: $\rho^{\in T \to S}$,
- the expressions/sets presently defining "interesting plans" (*e.g.*, goal-achieving, executable, cost-optimal): α and $\beta = \sigma(\alpha)$,
- the plans of interest: $A = \alpha^{-1}(\mathtt{True}) \subseteq X$ and $B = \beta^{-1}(\mathtt{True}) \subseteq Y$, finally
- the uninteresting plans: $\overline{A} = \alpha^{-1}(\mathtt{False}) \subseteq X$ and $\overline{B} = \beta^{-1}(\mathtt{False}) \subseteq Y$.

Assume/demand that the empty plan always represents itself: $\sigma(\emptyset) = \emptyset$ (and $\rho(\emptyset) = \emptyset$). An **interpretation** stipulates that the two directions really agree: $\rho(\sigma(x)) = x$ for all x, that is, the reverse map is a left-inverse. A **relaxation** guarantees uninterestingness: σ and ρ are such that $\rho(\overline{B}) \subseteq \overline{A}$. A **restriction** guarantees interestingness: σ and ρ are such that $\rho(B) \subseteq A$. A **compilation** does all three:

$$\begin{aligned} \sigma^{-1} &\subseteq \rho, \\ \rho(\overline{B}) &\subseteq \overline{A}, \qquad\qquad \text{and} \\ \rho(B) &\subseteq A. \end{aligned}$$

When the target language is smaller then we say **equivalence-reduction** rather than compilation: compilations and equivalence-reductions are two sides of a coin. So for example (taking ρ as the reduction direction), with $y =_\rho y'$ whenever $\rho(y) =$

$\rho(y')$, then we say $\sigma(\rho(y))$ is a choice of canonical representative for the ρ equivalence class of y. In other words a compilation is an equivalence-reduction in reverse: an anti-optimization, theoretically speaking.

Forcing: Not Inefficient Compilation. Not all compilation approaches are anti-optimizations in practice. Clever compilation can ensure the target planner does more or less just as well as a direct implementation. To motivate our definition of a forcing compilation, consider the following.

Fix some tree over the plans of the target, rooted at the empty plan. One plan **extends** another with respect to the tree if it is weakly a descendent: plans extend themselves. The **ill-formed** plans fail to extend to canonical plans: the plans in $\{y \mid \text{ there exists no } y' \in \sigma(X) \text{ extending } y\}$ are the ill-formed plans. The **partially-formed** plans succeed:

$$\{y \mid y \notin \sigma(X) \text{ and there exists } y' \in \sigma(X) \text{ extending } y\}$$

are the partially-formed plans. The **well-formed** plans in the target are just the canonical representatives: $\sigma(X)$.

Suppose the target manages to prune all and only the ill-formed plans. So every leaf would be canonical. Further suppose every vertex is a leaf or branch: no choices are forced. (Or just directly assume that total work is proportional to branches+leaves.) Then the number of internal vertices are bounded by the leaves. Hence total work is bounded by at most twice the number of canonical plans, which is already promising.

Leaves do not, in principle, require heuristic evaluation (and in some sense, no substantially expensive computation whatsoever need happen at leaves). Hence the number of heuristic evaluations could be bounded by the number of canonical plans. It is only a small stretch to further conclude that total work would be bounded directly by the number of canonical plans.

In other words, by supposing/proving either/both, we may conclude that the target planner would be performing more or less just as well as a direct implementation against the source language could possibly be expected to. Or well, to be accurate, at least we would have that the worst-cases would be close enough. Naturally we may easily suppose that the target planner would have less appropriate search-control.

That, however, is a question to be settled empirically. Then the significance is that it would be worth the while to in fact implement the compilation in question: it certainly matters whether a compilation approach is even remotely practical.

So say a compilation is **forcing** with respect to a target planner when:

1. at least every ill-formed plan is pruned, and ideally

2. only negligible work is performed on partially-formed plans.

Regarding the latter it would typically be enough to guarantee that at most one heuristic evaluation occurs per canonical plan; the cost of computing heuristics tends to dominate the runtime of domain-independent planners. Achieving either calls for heavy-duty unreachability analysis by the target planner.

A.2.2.2 Unreachability Analysis

"Maybe reachable" is a vague notion: good for intuition/insight [16, 102, 162]. For analysis/implementation it is better to work with "definitely unreachable" [20, 106, 127]. In particular, here the right perspective to take on planning graphs is that of HSP [20].

Let $\text{Regress}(a, q)$ denote a necessary and sufficient condition in normal form (*i.e.*, CNF) for action a to result in satisfaction of the condition q: $\text{Regress}(a, q) = \text{And}_i \text{ Or}_j \ \ell_{i,j}$ is such that $\text{Regress}(a, q)$ is satisfied by S iff $\text{Result}(a, S)$ satisfies q. An m-**literal** is a consistent conjunction of up to m literals, *i.e.*, an assignment on up to m fluents. We can relax these preconditions for conditional effects by choosing to satisfy only some of the constraints, *i.e.*:

$$\text{Relax}^m(\text{And}_{i\in[a]} \text{Or}_{j\in[b_i]} f_{i,j} = v_{i,j}) \coloneqq \text{And}_{I\in[a]^m} \text{Or}_{q\in Q(I)} q, \qquad \text{and} \tag{A.4}$$

$$Q(I) = \left\{q^{\in m\text{-literals}} \mid q \text{ satisfies } \text{And}_{i\in I} \text{Or}_{j\in[b_i]} f_{i,j} = v_{i,j}\right\} \tag{A.5}$$

make every choice of m clauses and every choice of how to satisfy each such set.

Then the heuristic h^m computes bounded-depth unreachability in the relaxation that ignores inconsistencies between m-literals:

$$h^m_{\mathcal{P},B}(\text{And}\,\text{Or}\, q) \coloneqq \max\min h^m_{\mathcal{P},B}(q). \tag{A.6}$$

$$h^m_{\mathcal{P},B}(q) \coloneqq 0 \text{ when } q \in B, \text{otherwise} \tag{A.7}$$

$$h^m_{\mathcal{P},B}(q) \coloneqq \min_{a\in Actions(\mathcal{P})} 1 + h^m_{\mathcal{P},B}(\text{Relax}^m(\text{Regress}(a, q))). \tag{A.8}$$

For self-contradictions and infinite recursions then naturally:

$$h^m(\cdot) \coloneqq \infty. \tag{A.9}$$

Some useful special cases for STRIPS-style problems are (with standard abuses):

$$h^m(q) = \min_{a: q\cap\text{Add}(a)\neq\emptyset \text{ and } q\cap\text{Del}(a)=\emptyset} 1 + h^m(\{p \mid q \setminus \text{Add}(a) \subset p \subseteq \text{Pre}(a)\}).$$

$$h^1(q) = \min_{a: q\in\text{Add}(a)} 1 + h^1(\text{Pre}(a)).$$

$$h^m(P) = \max_{p\in P} h^m(p).$$

- That h^m is well-defined, computable, and so forth are all standard results (*e.g.*, detecting infinite recursion is, here, legitimate).
- Technically the definition treats every action as mutex with every other action and so corresponds to the relatively uncommon serial planning graph. (The only way the definition permits an action to establish both of x and y, *e.g.*, in h^2, should it lack y as a direct effect, is to persist y and give x.) However, we

are primarily interested in just $m = 1$ and $h^m(\cdot) = \infty$, either of which renders the distinction meaningless.

- Similarly the definition differs from typical implementations in its treatment of disjunctions (*i.e.*, conditional effects). In STRIPS-style disjunctions are forbidden in the first place.

- It can be useful to note that m is only used to define the relaxed problem. Any algorithm for computing h^1 can be used to compute h^m as well if one is willing to write down the relaxed problem explicitly [97]. The relaxed problems, however, are exponentially large in m.

The intuition, for $m = 1$, is to drop every fundamental contradiction: permit both $f = v$ and $f = v'$ to be simultaneously true for every fluent and every pair of values. (In terms of formal problem relaxation: setup new boolean fluents whose names are (f, v) so that it is notationally possible to write down the impossibility without actual contradiction, *i.e.*, by writing $S((f, v)) = \texttt{True}$ and $S((f, v')) = \texttt{True}$ instead.) So the set of literals taken to be true only ever enlarges (because the relaxation of $f := v$ is $(f, v) := \texttt{True}$—there are no assignments to $\texttt{False}$), thereby earning the title: *delete-relaxation*. In particular satisfaction of every expression in the relaxation is monotonic: if ever true, then forevermore true. Then every relaxed plan remains executable if appended to any other. Meaning (un)solvability is polynomial.[2] That is, to solve the relaxed problem it suffices to generate as many witnesses of individual fluent-value pairs as possible (just one witness per pair, hence poly-time in the pairs); if that includes all top-level goals then any concatenation of all the witnesses is a (very sub-optimal) solution to the relaxed problem. Otherwise the relaxed problem is unsolvable—*as is the real problem*.

Example Inference Using h^m in Blocksworld. Increasing m drops fewer constraints, yielding a more informed heuristic. For example, h^2 recognizes that a single fluent can only have one value at a time ($h^2(f = v$ and $f = v') = \infty$ always), unlike h^1. Other, but not all, 2-size impossibilities can also be caught, for example:

Proposition A.1. *Stacking two blocks upon each other is 2-delete-relaxed unsolvable; having a block clear and yet below another block is also impossible even in the relaxation; moreover, stacking two blocks upon a common block is likewise deemed, in poly-time, unreachable:*

$$h^2((\text{below } a) = b \text{ and } (\text{below } b) = a) = \infty. \tag{A.10}$$

$$h^2((\text{clear } a) = \texttt{True} \text{ and } (\text{below } b) = a) = \infty. \tag{A.11}$$

$$h^2((\text{below } b) = a \text{ and } (\text{below } c) = a) = \infty. \tag{A.12}$$

[2] *Optimally* solving the relaxation is NP-complete.

For concreteness consider the specific model of Blocksworld on Page 39. The details should still work out for any vaguely normal encoding, but little can stop someone determined to defeat a heuristic [69].

Proof. Let q = ((below a) = b and (below b) = a); so the first aim is to show $h^2(q) = \infty$. There are 2 types of actions that can establish the expression: moving a to ontop of b, or vice-versa. They do so if applied in a situation where the other half of the expression is already true. Renaming objects has no meta-effect, *i.e.*, by symmetry: it suffices to show that moving a onto b in a situation where b is already on a is known to be impossible. In notation, the aim is to show that:

$$x = 1 + \max\left\{h^2(p'^{\ \in 2\text{-literals}}) \mid \text{Result}((\text{move } a\ b), S) \text{ satisfies } q \text{ only if } S \text{ satisfies } p'\right\}$$

is undefined/infinite. Which suffices, as x is a straightforward under-estimate of h^2, *i.e.*, $h^2(q) \geq x$. If it were possible the movement would have to be executable—in particular a would have to be clear above and beyond persisting (below b) = a.

(P). Consider p = ((clear a) = `True` and (below b) = a): p is a necessary condition for the movement to be both executable and result in q. Therefore $h^2(q) > h^2(p)$. Should we happen to know that "clear" literally means the absence of every covering block then we are done but for the moreover. Otherwise it suffices to show $h^2(p) = \infty$ is anyways inferred (from the moreover). No action has p as its direct effect, so again one half must be persisted. Case: persist clear. The only way to establish (below b) = a is to move b onto a, but doing so also has the effect (clear a) := `False`. So p can never be a post-condition of moving b onto a. The regression of an apparent contradiction through an action can be assumed/demanded to produce syntax remaining clearly unsatisfiable, *e.g.*, by just copying the already apparent contradiction. For h^2, "apparent" means containing contradictory unit clauses, which "(clear a) = `False` and (clear a) = `True`" meets. (So it is indeed formally correct to drop such obviously impossible actions from the relaxed problem: doing so does not alter h^2 but for speeding up its computation.) Then the remaining case is to attempt to establish the other half of p. Case: persist below. The only actions establishing (clear a) = `True` are (i) moving a anywhere, and (ii) moving a block on top of a elsewhere. (i) Moving a has as precondition that it already be clear, so all of p would have to have held before. Which is circular. For such circularity to win the minimization over actions means that the relation inferred is $h^2(p) = 1 + h^2(p)$. That recurrence relation is unsolvable, denoted by writing $h^2(p) = \infty$, *i.e.*, p would be deemed unreachable (as desired). (ii.1) Moving b off of a would directly contradict p (by setting (below b) to something besides a), so is deemed impossible by virtue of its regression containing contradictory unit clauses. (ii.2) So some other block, say c, must move off of a if $h^2(p)$ is to be finite.

(O). Consider o = ((below c) = a and (below b) = a). Then moving c off of a will establish p—but only if o. That is, $h^2(o)$ is a candidate in the maximization for the sole remaining conceivable way of establishing p and thus: $h^2(p) > h^2(o)$. Therefore it suffices to show $h^2(o) = \infty$. Again no single action could establish both halves at once. Case: persist below b. Consider moving c onto a from a state where b is already on a. Then p itself is a candidate in the maximization: o and p cannot both be bigger than each other (so are deemed infinite). Case: persist below c. Consider moving b onto a from a state where c is already on a. So p' = ((clear a) = `True` and (below c) = a) is a necessary condition (and thus $h^2(o) > h^2(p')$).

Then return to **(P)** with $p := p'$ and $q := o$. That is, ping-pong back and forth until exhausting every variation on: **(P)** a block is on a yet a is clear, **(O)** two blocks are on a.[3] Hence all $\left\{h^2(r) \mid r \in P \cup O\right\}$ are h^2-provably greater than each other: all are infinite. □

[3]Being clear is much like being below only the hand; for such an encoding of Blocksworld the latter two claims of the proposition may be combined together as "at most one thing is directly above a block". The proof could then consider all minimal counterexamples in one fell swoop.

A.2.2.3 Landmark Analysis

A "landmark" is in principle a necessary and sufficient condition for reachability of the final destination. In planning, consideration of sufficient conditions for goal -reachability are considered only rarely, for the reason that a general enough treatment subsumes the whole task [61]. That is, landmarks, in planning, are only necessary conditions [174], albeit sometimes exploited in a manner befitting sufficient conditions [116].

An h^m**-detectable landmark** (α, γ, β) satisfies: for any B satisfying (α and $\neg\beta$) and B' satisfying (β and $\neg\alpha$) there exists at least one witness q, satisfied by B', such that $h^m_{\mathcal{P}-\gamma,B}(q) = \infty$. The intuition is that satisfaction of β is unreachable from satisfaction of α in the problem suppressing γ. (An action a appearing in α or β stands for all the situations in which it is executable, *i.e.*, Regress(a, `True`).) Usually α and β name sets of vertices comprising a separation of a Domain Transition Graph (or for $m > 1$, separations of several Domain Transition Graphs merged together).[4] That is, these would usually have the form q_1 or q_2 or $q_3 \cdots$ for a set of m-literals q_i known to be pairwise mutually exclusive in the real problem.

The inefficient way to find such separations would be to hypothesize cuts, do them, and detect unreachability of certain literals explicitly. (Which is how the notation reads.) Some of the intuition from that perspective is useful nonetheless. For example, consider the final layer of a special planning graph built from the initial situation, suppressing γ. Say some situation satisfies no literal considered unreachable by suppressing γ. (So it satisfies $\neg\beta$ for β the disjunction over all such conditionally unreachable literals.) Then the literals it does satisfy are a subset of the final layer of the special planning graph. Hence it continues to fail to reach any situation satisfying any of those conditionally unreachable literals. Barring the use of γ, of course.

In other words, landmarks name (directed) cuts in the *real* problem: highly exploitable knowledge. Better yet landmarks name (directed) separations—the planner can know which side of a cut it is presently considering. At least, that is how we formulate them here. It is also possible to consider landmarks as mere cuts. (Which is more general, but awkward, as it calls for additional machinery to track whether or not a landmark has taken place sometime in the past.)

Example Landmark Analysis. Unqualified, a landmark means specifically a separation of the initial situation from every goal situation, in truth (*i.e.*, with respect to h^*). For example, in 4-operator Blocksworld unstacking block b from c is a landmark with respect to (i) block b starting on top of c, and (ii) block c not starting on top of whatever the goal demands. Such is certainly true with respect to the real domain physics. It is moreover the case that said landmark is detectable

[4] A Domain Transition Graph is equivalent to the result of contracting together all states agreeing upon the value of a given fluent.

by merely h^1. Say block d is what is to be below c in the goal. Then in our notation, ((i) and (ii), (unstack b c), (on c d)) is an h^1-detectable landmark. (Meta -Proof) Each inference in the following chains through just one literal at a time as precondition/effect: (Paraphrasing of h^1-detection) block c does not start where it needs to finish, so needs to move; any movement requires that it be clear; so it suffices to show that block c is never clear; the only actions giving that c is clear do so by moving a block off of c; moving a block off of c requires that a block is already on c; the only actions resulting in their being a block on c require it already be clear; which is circular; so only moving specifically the block already on c, block b, can witness achieving that c is clear for the first time; suppressing said movement then ensures c is never clear for a first time; hence block c is never clear at all in the modified problem, which suffices.

Notation for Suppressing Propositions and Actions. Say $\mathcal{P} - a$ denotes the planning problem with action a removed. To address minor technicalities it is easier to make it impossible than to outright remove it. Define **removal** of actions as making their definitions empty: $\mathit{eff}_a := \{\}$ in $\mathcal{P} - a$ is the sole change. So syntactically it remains possible to include such actions in plans, but such plans are never executable, because said actions are nowhere executable. Similarly treat removal of sets of actions.

Let $\mathcal{P} - (f = v)$ denote adding the negation of the condition, $f \neq v$, as a precondition to every action a (syntactically) dependent upon the fluent f. Then any remaining executable plan never exploits $f = v$. Moreover, if $f = v$ should indeed ever become true then it remains so thereafter—the more interesting aspect of which is that no action thereafter depends upon f in any way. Furthermore for any action already requiring $f = v$ as a precondition the manipulation is tantamount to making its definition be empty. At least, intuitively speaking that is so; in STRIPS-style one might very well define $\mathcal{P} - (f = v)$ as equivalent to removing all actions explicitly requiring $f = v$. In general, however, the two definitions are not equivalent; adding $f \neq v$ as a precondition is more constraining than just removing actions requiring $f = v$.

Let $\mathcal{P} - (f := v)$ denote adding $\neg\text{Regress}(a, f = v)$ as a precondition to each action a. Then no remaining executable plan can ever re-establish $f = v$; it is only true if initially true, and has remained so up to the time in question. Which is much like removing all actions that, if executable, certainly carry out the effect $f := v$; for STRIPS-style that equivalence is provable, because, in that setting, there are no such things as indirect or conditional effects.

APPENDIX B

FORMAL PROOF OF THE DOMINANCE OF LEFT-SHIFTING

Here we rigorously prove that left-shifted defines a dominance reduction. For reference, the theorem may be stated precisely:

Theorem (Left-Shifting is Complete)**.** *Left-shifted schedules dominate their action -sequence equivalence classes.*

Conceptually the theorem is saying: "Fastest is Best". We break its proof down into two lemmas: "Faster is Locally Better" and "Locally Better is Globally Better". So the first lemma creates small local improvements. Then the second lemma converts those into small global improvements by induction. Finally the proof of the theorem converts those into largest global improvements, by another induction.

We begin by precisely defining dominance between situations, prove that shifting a single action earlier in time immediately results in dominance, and follow that up with a demonstration that the dominance persists through the tail of the altered schedule. Rescheduling a second action raises issues. In short the precise values of earliest start-times are moving targets as we engage in rescheduling. To address this we appeal to universally left-shifted schedules as a mechanism for referring to earliest start-times by name (rather than value). Then it is no longer an issue should the precise value later change. Finally we take our amassed tools, restate the theorem, and prove it.

B.1 FASTER IS LOCALLY BETTER

To make the proof go through we need to generalize dominance to cover the mutual exclusions between action effects; the notion from Chapter 3 is only strong enough to cover satisfaction of goals and subgoals as soon as possible. Recall that situations consist of just states and vaults and that the theorem concerns action -sequence equivalent schedules. Meaning the underlying state-sequence is fixed during rescheduling. So only the vault-sequences of executions really matter.

Then the idea is: One vault dominates another if every hypothetical dispatch has an at least as early, earliest, start-time. Earliest start-times are just minimizations over subsets of read-times and write-times, so:

Definition B.1 (Dominance)**.** A vault $X^{\in Vaults}$ **dominates** another $Y^{\in Vaults}$ when, for all fluents $f^{\in Fluents}$:

$$\text{Read-Time}(X_f) \leq \text{Read-Time}(Y_f), \qquad \text{and} \tag{B.1}$$

$$\text{Write-Time}(X_f) \leq \text{Write-Time}(Y_f); \tag{B.2}$$

say vaults X and Y are of the same **vault-subtype**[1] if furthermore:

$$Readable(X_f) = Readable(Y_f). \tag{B.3}$$

If all three conditions hold, then say the vault X **dominates with type** the vault Y. Observe that the relation is a partial-order on vaults: write $X \preceq Y$.

Say a vault transition function **dominates** another when, on *all* inputs, its result dominates (with type) the other's result. Then we can show that starting actions sooner results in dominance:

Lemma B.1. *The earliest vault transition function (V'_a) dominates every non-earliest vault transition function ($V'_{a,t}$). In notation, for arbitrary actions $a^{\in Actions}$ and requested start-times $t^{\in \mathbb{Q}}$:*

$$\text{for all } V^{\in Vaults}, V'_a(V) \preceq V'_{a,t}(V). \tag{B.4}$$

Proof. Let vault $X = V'_a(V)$ and vault $Y = V'_{a,t}(V)$ denote the vaults resulting from respectively dispatching a earliest and at some other time t. Demonstrating dominance and vault-subtype equality may be pursued for each fluent separately: let fluent $f^{\in Fluents}$ be arbitrary. Denote the lock on f resulting from applying the action earliest by $\ell' = (Acquired', Released', Readable') = X_f$. Likewise let $\ell = (Acquired, Released, Readable) = Y_f$. Then it suffices to show, with

[1] The third condition, vault-subtype, is just to streamline the proofs. Conceptually only the first two conditions should matter.

Read-Time′ = Read-Time(ℓ'), Read-Time = Read-Time(ℓ), and so forth:

$$\text{Read-Time}' \leq \text{Read-Time}, \tag{a}$$
$$\text{Write-Time}' \leq \text{Write-Time}, \qquad \text{and} \tag{b}$$
$$\mathit{Readable}' = \mathit{Readable}. \tag{c}$$

(Claim) The start-times and release times of ℓ' are monotonically sooner than those of ℓ: (i) $\mathit{Acquired}' \leq \mathit{Acquired}$, and (ii) $\mathit{Released}' \leq \mathit{Released}$.

Read-time dominance, *i.e.*, (a), follows from (c), (i), and (ii) by the definition of read-times, namely: Read-Time = if*Readable* then *Acquired* else *Released*. Write-time dominance, *i.e.*, (b), follows from just (ii), since write-time is just a synonym of release time. Then it suffices to show (c), (i), and (ii).

Conduct a case-analysis over the manner in which the action effects the lock on fluent f. The arguments are all by inspection of the definition of acquired-locks, recall:

$$\begin{aligned}
\text{Acquired-Read-Locks} &= (\text{Read-Time}, \max(\mathit{Released}, \text{AFT}), \texttt{True})_{\mathit{Reads}}, \\
\text{Acquired-Write-Locks} &= (\text{Write-Time}, \text{AFT}, \texttt{False})_{\mathit{Writes}}, \\
\text{AFT} &= \text{AST} + \mathit{dur}, \text{and} \\
\text{AST} &= \max(\text{EST}, t).
\end{aligned}$$

Case $f \notin \mathit{Depends}$. Then the fluent is neither read from nor written to, ergo, its lock is uneffected. So $\ell' = \ell = V_f$ holds trivially. Then (c), (i), and (ii) are confirmed for uneffected fluents.

Case $f \in \mathit{Depends}$. Confirm (c), (i), and (ii) in turn.

For (c): By the definition of Conservative Temporal Planning, whether actions write to, or only read from, any given fluent is fixed. So $\mathit{Readable}' = \mathit{Readable}$ holds trivially, confirming (c).

For (i): By the definition of acquired-locks, locks are always acquired as soon as possible, *i.e.*, independently of actual start-time. Then, because also (c) $\mathit{Readable}' = \mathit{Readable}$ holds, either both lock acquisition times are the read-time of the pre-existing lock, Read-Time(V_f), or both are the write-time of the pre-existing lock, Write-Time(V_f). Either way they are equal. Therefore $\mathit{Acquired}' = \mathit{Acquired}$ holds, confirming (i).

Then it remains only to confirm (ii). Consider separately the (exhaustive and disjoint) sub-cases of write-locks and read-locks: $f \in \mathit{Writes}$ and $f \in \mathit{Reads} = \mathit{Depends} \setminus \mathit{Writes}$.

Case $f \in \mathit{Writes}$. From the definition of acquired write-locks: The competing write-locks, ℓ' and ℓ, are held until respectively the earliest finish-time and the actual finish-time with respect to t: $\mathit{Released}' = \text{EFT}$ and $\mathit{Released} = \text{AFT}$. These really do obey the constraint $\text{EFT} \leq \text{AFT}$, as the action's duration is a constant,

and both are just offset from the start-times by said duration: EFT = EST + $dur \le$ AST + dur = AFT by EST $\le$ AST. So (ii) is confirmed for fluents written to.

Case $f \in Reads$. From the definition of acquired read-locks: The release time of a read-lock on f is the greater of the pre-existing release time $Released(V_f)$ and the actual finish-time of the action. So $Released' = \max(Released(V_f), \mathrm{EFT})$ and $Released = \max(Released(V_f), \mathrm{AFT})$. Since EFT $\le$ AFT (see above), then either possibility for $Released'$ is sooner than at least one possibility for $Released$. That is, the desired relationship holds: $Released' = \max(Released(V_f), \mathrm{EFT}) \le \max(Released(V_f), \mathrm{AFT}) = Released$. Then (ii) is confirmed for fluents only read from. □

So we have a relatively intuitive notion of dominance and a suitable place to apply it for a local benefit. Greed, though, is not always good.

For Conservative Temporal Planning, greed *is* good:

Lemma B.2. *If vault X dominates with type a vault Y, then any successor of X dominates with type the corresponding successor of Y. Precisely, for any dispatch (a, t), vault X, and vault Y:*

$$\text{if } X \preceq Y \tag{B.5}$$
$$\text{then } V'_{a,t}(X) \preceq V'_{a,t}(Y). \tag{B.6}$$

Moreover type-dominance is preserved through whole dispatch-sequences.

The structure of the proof is nigh identical to the last; it is perhaps straightforward to abstract over both lemmas.

Proof. Let $X' = V'_{a,t}(X)$, $Y' = V'_{a,t}(Y)$ denote the successors of X and Y. So the aim is to show that $X' \preceq Y'$ follows from $X \preceq Y$; note that the moreover follows by induction. It suffices to consider one fluent at a time: let $f^{\in Fluents}$ be arbitrary. There are four concerned locks. Let $x = X_f$ and $x' = X'_f$ be the locks before and after along the dominating sequence. Likewise let $y = Y_f$ and $y' = Y'_f$ be with respect to the dominated sequence. Then the task is to demonstrate, with respect to the new locks, the desired properties:

$$\text{Read-Time}(x') \le \text{Read-Time}(y'), \tag{a$'$}$$
$$\text{Write-Time}(x') \le \text{Write-Time}(y'), \qquad \text{and} \tag{b$'$}$$
$$\mathit{Readable}(x') = \mathit{Readable}(y'). \tag{c$'$}$$

With respect to the old locks we are given those properties by hypothesis:

$$\text{Read-Time}(x) \le \text{Read-Time}(y), \tag{a}$$
$$\text{Write-Time}(x) \le \text{Write-Time}(y), \qquad \text{and} \tag{b}$$
$$\mathit{Readable}(x) = \mathit{Readable}(y) \qquad \text{(which is unnecessary).} \tag{c}$$

Argue by a case analysis over the manner in which the action effects the fluent f.

Case $f \notin \mathit{Depends}$. There is nothing to show: the locks simply persist ($x = x'$ and $y = y'$). *I.e.*, (a′), (b′), and (c′) follow immediately from (a), (b), and (c).

Case $f \in \mathit{Depends}$. (Claim) It suffices to show (i) $\mathit{Acquired}(x') \le \mathit{Acquired}(y')$, (ii) $\mathit{Released}(x') \le \mathit{Released}(y')$, and (c′) all hold. Write-times are just release times: (b′) follows from (ii). Read-times are either acquisition times or release times depending on the lock-types, which are equal by (c′). So (a′) follows from (i) in the case that f becomes read-locked, and from (ii) in the case that f becomes write-locked. Then the claim is shown—it remains to confirm (i), (ii) and (c′). First establish (c′) and (i), completing the trivial cases. Finally argue for (ii).

For (c′), *i.e.*, lock-type equality: Fluents read by an action are read-locked, and fluents written by an action are write-locked, independently of their former status. So $Readable(x') = Readable(y')$ holds trivially as both are equal to $f \in Reads$.

For (i), *i.e.*, acquisition time dominance: The aquisition-times select the former read-times or write-times, as appropriate for the new lock-types, which are equal by (c′). So in the case of reading (i) follows from (a): $Acquired(x') = \text{Read-Time}(x)$ and $Acquired(y') = \text{Read-Time}(y)$. Likewise in the case of writing (i) follows from (b): $Acquired(x') = \text{Write-Time}(x)$ and $Acquired(y') = \text{Write-Time}(y)$.

Only release time dominance (ii) remains. Here the start-times and finish-times of the action dispatch actually matter. First, by the dominance of X over Y, the earliest start-time in X is monotonically sooner than in Y: $\text{EST}(X) \leq \text{EST}(Y)$. As actual start-times are just the maximum of earliest and requested start-times then also the actual start-time in X is no later than in Y: $\text{AST}(X) \leq \text{AST}(Y)$. Then since actual finish-times are just actual start-times plus the action duration, which is fixed, again X has the advantage:

$$\text{AFT}(X) \leq \text{AFT}(Y). \tag{f}$$

Then finish by considering separately whether the fluent becomes write-locked or read-locked.

Case $f \in Writes$. The release times are the actual finish-times: $Released(x') = \text{AFT}(X)$ and $Released(y') = \text{AFT}(Y)$. So (ii) follows by (f).

Case $f \in Reads$. The release times are the maximizations over the former release times and the actual finish-times: $Released(x') = \max(Released(x), \text{AFT}(X))$ and $Released(y') = \max(Released(y), \text{AFT}(Y))$. Then (ii) is confirmed, as there is always a larger possibility on the right, respectively by (b) and (f):

$$\max(Released(x), \text{AFT}(X)) \leq \max(Released(y), \text{AFT}(Y)). \qquad \square$$

The current status is as follows. We have defined “Better” and “Faster”, shown that “Faster” gives “Locally Better”, and shown that “Locally Better” is also “Globally Better”. So it remains to define “Best”, define “Fastest”, and show that the two coincide.

B.3 FASTEST IS BEST

We distinguish two particular left-shifted schedules: actual and universal. The requested and actual start-times in an **actual** schedule coincide. The actual and earliest start-times in a **left-shifted** schedule coincide. So the **actual left-shifted schedule** requests (and receives) the precise values of its earliest start-times.

Definition B.2 (Left-Shifted)**.** Let Left-Shift$(X, V_0) \coloneqq (a, t)_{[n]}$ denote, with respect to vault V_0, the **actual left-shifted schedule** action-sequence equivalent to schedule $X = (a, \cdot)_{[n]}$ given by setting every requested start-time to the earliest start-time. Precisely, set t_i for each $i^{\in[n]}$ as:

$$t_i \coloneqq \mathrm{EST}_{a_i}(V_{i-1}), \quad \text{and} \tag{B.7}$$

$$V_i \coloneqq V'_{a_i}(V_{i-1}). \tag{B.8}$$

Note that the vault-sequence in the definition is identical to the result of attempting to execute the schedule. (It would be poor form though to refer to executions of the schedule during its own definition.) Style aside, with respect to just the vault V_0: the actual start-times, earliest start-times, and requested start-times are all clearly identical. Which is fine for the statement of the theorem, but the dependency on V_0 is inconvenient for its proof.

The **universal left-shifted schedule** instead asks for earliest start-times 'by name'. For notation just use the beginning-of-time $t_{-\infty}$ as a sentinel value to denote a request for earliest start-time. Note that any action scheduled for $t_{-\infty} < t_0$, or any other such 'negative' value, automatically waits until its earliest start-time: $V'_{a,t_{-\infty}} = V'_a$. So the **universally left-shifted schedule** $X^\dagger \coloneqq (a_i, t_{-\infty})_{i\in[n]}$, action-sequence equivalent to schedule $X = (a, \cdot)_{[n]}$, is given by just setting every request to the beginning-of-time. Then finally we are ready:

Theorem B.3 (Left-Shifting is Complete)**.** *Left-shifted schedules dominate their action-sequence equivalence classes.*

Proof. Suppose $X = (a, t)_{[n]}$ is a solution; the task is to prove that its action-sequence equivalent universally left-shifted schedule $X^\dagger$ is a solution as well. (As, by Proposition 3.6, the executions of $X^\dagger$ and Left-Shift$(X, \mathit{Vault}_{\mathit{Initial}})$ are identical, hence the two are result-equivalent and thus also solution-equivalent.) The argument proper is an induction through increasingly long suffixes of the given solution, demonstrating at each step that the current suffix is dominated by its left-shifted version.

First we extend dominance to schedule fragments (*i.e.*, the suffixes), so that we may apply the lemmas more directly. For any subsequence $Y = (a, t)_{[i,j]}$ of some larger schedule let $V'_Y \coloneqq V'_{a_j,t_j} \circ V'_{a_{j-1},t_{j-1}} \circ \cdots \circ V'_{a_i,t_i}$ denote the composition of its vault transition functions. In the event that these are all equal to earliest vault transition functions, then omit the irrelevant requested start-times; *i.e.*, for $Y^\dagger$ write:

$V'_{Y^\dagger} := V'_{a_j} \circ V'_{a_{j-1}} \circ \cdots \circ V'_{a_i}$. As with single dispatches, say Y **dominates** Z when, for all possible input vaults, the output from V'_Y dominates with type the output from V'_Z. For notation, define $Y \leq Z$ when:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_Y(V) \leq V'_Z(V). \tag{B.9}$$

Then argue by induction on a suffix $Y = X\restriction_{[k,n]}$ of X. That is, assume we know $Y^\dagger$ dominates Y and conclude that $Z^\dagger$ dominates $Z = X\restriction_{[k-1,n]}$. For notation, assume:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Y^\dagger}(V) \leq V'_Y(V). \tag{IH:k}$$

From which, conclude:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Z^\dagger}(V) \leq V'_Z(V). \tag{IH:$k-1$}$$

Which suffices, as then, by induction, (IH:1) would hold, in particular:

$$V'_{X^\dagger}(\mathit{Vault}_{\mathit{Initial}}) \leq V'_X(\mathit{Vault}_{\mathit{Initial}}). \tag{B.10}$$

That such is strong enough follows by Proposition 3.5 and Proposition 3.7; goal satisfaction is monotone in decreasing read-times, which dominance is stronger than. Therefore $X^\dagger$ is a solution whenever X is.

Hence it remains only to in fact show the induction. By Lemma B.1 we know—which for $k = n$ addresses the base case (IH:n)—that starting the last action before the current suffix as soon as possible locally dominates:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{a_{k-1}}(V) \leq V'_{a_{k-1},\, t_{k-1}}(V).$$

By Lemma B.2 we have that this dominance is maintained throughout the suffix:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Y^\dagger}(V'_{a_{k-1}}(V)) \leq V'_{Y^\dagger}(V'_{a_{k-1},\, t_{k-1}}(V)).$$

This can be simplified to the following statement. The left-shifting of the longer suffix dominates left-shifting all but its first action:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Z^\dagger}(V) \leq V'_{Y^\dagger}(V'_{a_{k-1},\, t_{k-1}}(V)). \tag{IH:$k-1$:i}$$

To finish, first reword the induction hypothesis: Left-shifting all but the first action of the longer suffix dominates no rescheduling whatsoever. Then conclude by transitivity that left-shifting all of the longer suffix dominates. In notation, first substitute $V'_{a_{k-1},t_{k-1}}(V)$ for V into (IH:k):

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Y^\dagger}(V'_{a_{k-1},t_{k-1}}(V)) \leq V'_Y(V'_{a_{k-1},t_{k-1}}(V)).$$

Then simplify to Z as intended:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Y^\dagger}(V'_{a_{k-1},t_{k-1}}(V)) \leq V'_Z(V). \tag{IH:$k-1$:ii}$$

So finally, (IH:$k-1$) follows from (IH:$k-1$:i) and (IH:$k-1$:ii) by transitivity:

$$\text{for all } V^{\in \mathit{Vaults}}, V'_{Z^\dagger}(V) \leq V'_Z(V). \qquad \square$$

APPENDIX C

THE DEFERRED CASE ANALYSIS FOR COMPILING TO THE MINIMAL TEMPORALLY EXPRESSIVE SUBLANGUAGES OF INTERLEAVED TEMPORAL PLANNING

Here we complete the proof of Lemma 4.17, which is missing the technical details of its case analysis. The general idea of the overall proof is to:

- split up any given action (α) into a chain-decomposition (β),
- compile each piece into its own action ($\hat{\beta}$ for all of them),
- setup an envelope to contain them all (*do*), and
- use two matched pairs of virtual actions to help control the relationship between the envelope and its contents (*setup*/*reset* and *before*/*after*).

The last, which is all that remains to be proven, is needed to circumvent the extreme syntactic limitations of the minimal cases.

Recall the regular expression inspired notation. The notion of the regular language denoted by a regular expression is entirely standard. The notational liberties taken (which documentation is previously omitted, incidentally) are documented in the following rundown of technical details.

- The tokens (*i.e.*, the alphabet symbols) are the names of the parts of compound actions.
- Parentheses and commas are for sequencing tokens.
- "+" is for one-or-more.
- "$*$" is for zero-or-more.
- Alternation/disjunction is unused (presumably write "|").
- Compounds stand for the sequence through their parts, for example, with α a compound: the regular expression "α" rewrites to "(`all`-α, `bgn`-α, `fin`-α)".
- We write such regular expressions as assertions capturing much of our state of knowledge regarding the properties of the compilations. Elaborating, the meaning of asserting "(regular expression) *foo* (holds)"—which is always written just "*foo*", *sans* quotes, in the following—is:
  - all solutions to the compiled problems (viewed as token-sequences),
  - up to operationalizing (*e.g.*, by completness-preserving pruning) any context-specific equivalence/dominance reductions,
  - projected onto the alphabet of *foo* (*i.e.*, throw away tokens not in *foo*),
  - lie within the regular language denoted by *foo*.

The intent here is to concisely state certain meta-properties about the reasoning capabilities had within the general framework of merging Domain Transition Graphs together (but stated from the perspective of the edge labels rather than the vertex labels, which latter vocabulary is the norm for discussion of Domain Transition Graphs) [106]. Elaborating, the reason to demonstrate correctness of the compilations through that particular set of techniques is to strongly substantiate forcing. For example, the twist of considering the alphabet underlying *foo* to be just the tokens appearing within *foo* points to (presumably rather nonobviously) the (hard in practice) problem of *fluent merging*. In other words, I leave those ambiguities as pointers towards promising future work.

Recall also the strategy for completing the case analysis (Page 288), reproduced momentarily, which consists of demonstrating four key properties—labeled throughout the following by (1) through (4)—for each case. Finally recall the overarching per-action compilation strategy and attendant machinery developed: the envelope *do*, the pair *setup*/*reset* wrapping the compiled view of the action α, said compiled view $\hat{\beta}$ of the action α, all of which thus far comprise the envelope and its intended contents, and finally the pair *before*/*after* (which are supposed to occur outside the envelope) serving to separate multiple simulations of the action α.[1] Then, to complete support of Theorem 4.12, it suffices to prove:

Lemma C.1 (The Case Analysis of Lemma 4.17)**.** *The four guarantees needed by Lemma 4.17 per minimal temporally expressive sublanguage hold. Namely:*

1. $(\mathit{before}+, \mathit{setup}+, \mathit{reset}+, \mathit{after}+)*$.

2. $(\mathit{before}+, \mathit{do}+, \mathit{after}+)*$.

3. *Every envelope starts between the last* before_α *and first* setup_α.

   That is, with (1) and (2): $(\mathit{before}+, \texttt{bgn}\text{-}\mathit{do}+, \mathit{setup}+, \mathit{reset}+, \mathit{after}+)*$.

4. *Every envelope ends between the last* reset_α *and first* after_α.

   That is, with (1) and (2): $(\mathit{before}+, \mathit{setup}+, \mathit{reset}+, \texttt{fin}\text{-}\mathit{do}+, \mathit{after}+)*$.

[1] Achieving repeatability of the compilation of each action is perhaps the greatest aspect forcing technical complexity well beyond conceptually necessary. (The severely limited syntax of the minimal cases is also a strong contender.) The significance follows from the observation that many benchmarks can be strongly attacked in practice from the direction of assuming that actions need to be repeated at most only a handful of times [202]. (An easy counterexample is the Towers of Hanoi puzzle.) From that assumption we can just make copies in order to implement a guarantee that every action need occur at most once. Here that would permit notably simplifying the bookkeeping needed. In practice the (intimately related) observation would be that automating the (landmark-style) inference is notably simpler when we can ignore the distinction between the/first/last/all instance(s) of an action.

Proof. Every case shares the following book-keeping.

- For reference, action *do* has duration $dur_{\alpha,0} + 4\hat{\mu}$.
- For reference, actions *before* and *after* have unit-duration (written $\hat{\mu}$ to distinguish from the original unit of time).
- Create a boolean fluent *before-possible*, initially and finally true.
- Create a boolean fluent *setup-possible*, initially and finally false.
- Create a boolean fluent *reset-possible*, initially and finally false.
- Create a boolean fluent *after-possible*, initially and finally true.
- Add *before-possible* ≔ `True` to *after*.
- Add *before-possible* = `True` to *before*.
- Add *before-possible* ≔ `False` to *setup*.
- Add *setup-possible* ≔ `True` to *before*.
- Add *setup-possible* = `True` to *setup*.
- Add *setup-possible* ≔ `False` to *reset*.
- Add *reset-possible* ≔ `True` to *setup*.
- Add *reset-possible* = `True` to *reset*.
- Add *reset-possible* ≔ `False` to *after*.
- Add *after-possible* ≔ `True` to *reset*.
- Add *after-possible* = `True` to *after*.
- Add *after-possible* ≔ `False` to *before*.

The machinery is just an instance of token-passing (with descriptive names), in support of (1). That is, by the same kind of analysis by which we already know:

$$(\mathit{setup}{+}, \hat{\beta}_{i,1}{+}, \hat{\beta}_{i,2}{+}, \ldots, \mathit{reset}{+})*,$$

likewise infer, from the book-keeping just above, satisfaction of (1):

$$(\mathit{before}{+}, \mathit{setup}{+}, \mathit{reset}{+}, \mathit{after}{+})*, \tag{C.1}$$

and so moreover, combining both, infer:

$$(before+, setup+, \hat{\beta}_{i,1}+, \ldots, reset+, after+) * . \tag{C.2}$$

Then it remains to show (2–4) for each case. We could easily implement (2), which is roughly that *before*/*after* wrap each instance of the envelope, in case-independent fashion (*i.e.*, by applying more token-passing). However, implementing (3) and (4) —for a rough statement, these mean that the envelope is to wrap *setup*/*reset*— anyways does most/all of the work of (2) as a side-effect.

Then consider each of the following four cases in turn (which are exhaustive by Lemma 4.14 and Theorem 4.1). In the first case we shall proceed most carefully and thoroughly. All subsequent cases will typically, for example, elide qualifying "repetition" by "internal", omit "instance of", and so forth. Even in the first case, that "first/last/all/only/the" are meant per repetition of the action α is not always spelled out in so many words. The notation, in contrast, is reasonably accurate and precise throughout (albeit, subscript-litter such as do_α everywhere is still omitted).

Case: `L(eff; pre; pre)`. For (3) and (4), we ensure the envelope can only start and end in the right places using the conditions we are allowed in this case to place at its start and end. We ensure it must occur per instance of *before*, *i.e.*, for (2), by making it responsible for achieving a new condition for carrying out *after*. The case-specific book-keeping:

- Create a boolean fluent *do-occurred*, initially and finally true.
- Add *do-occurred* := `True` to the all-part of *do*.
- Add *do-occurred* = `True` to *after*.
- Add *do-occurred* := `False` to *before*.
- Add *before-possible* = `True` to the start-part of *do*.
- Add *setup-possible* = `True` to the start-part of *do*.
- Add *reset-possible* = `True` to the end-part of *do*.
- Add *after-possible* = `True` to the end-part of *do*.

We already have, from before the case analysis, that *before*, *setup*, $\hat{\beta}_i$, *reset*, and *after* all occur (hypothetically with undesirable repeats), in that order, cyclically throughout every solution, for every i. Concisely, equation (C.2) describes all solutions:

$$(before+, setup+, \hat{\beta}_{i,1}+, \ldots, reset+, after+) * .$$

By the mechanics of *do-occurred*, combined with our prior knowledge that *before* and *after* are governed by a loopy cycle throughout solutions, we have that (2) holds, *i.e.*:

$$(\mathit{before}+, \mathit{do}+, \mathit{after}+)*$$

describes all solutions.

Note that *setup-possible* is false from the first instance of *setup* until the first instance of *after*. That it be true is a condition for starting the envelope. So every internal repetition of the envelope starts between the first instance of *before* and the first instance of *setup*. Because of the mutual exclusion concerning *do-occurred* between the all-part of the envelope and *before*, we furthermore know that every internal instance of the envelope begins after the *last* internal repetition of *before*. Therefore we know (3). Then we may write, regarding (by Proposition 5.18) the all -part and start-part of the envelope as one primitive:

$$(\mathit{before}+, \mathsf{bgn}\text{-}\mathit{do}+, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \mathit{after}+) * .$$

Note that *after-possible* is false from the first instance of *before* until the first instance of *reset*. That it be true is a condition for ending the envelope. So every internal repetition of the envelope ends after the first instance of *reset*. Recall that *after* deletes *reset-possible*, which is a condition for ending the envelope. So every internal repetition of the envelope ends before the first instance of *after*. Then we have most of (4): we are missing that the envelope always ends after the *last* internal repetition of *reset*. We could write that out as a regular expression, but let us just eliminate (useless) repetitions of *reset* directly (next).

In any state subsequent to executing a *reset* and preceding its corresponding *after*, it (*i.e.*, *reset*) is a no-op. That is because (a) its corresponding *after* is a landmark for everything mutex with *reset* besides itself and ending the envelope, and (b) ending the envelope changes nothing.[2] Thus *reset* is effectively unique up

[2]The first to follow among that which changes what *reset* changes, besides itself, is just *after*. Hence internal repetitions of *reset* alter no fluent by inspection. Ergo such produce strictly dominated situations with respect to locks. By temporal reasoning all of its parts are contiguous in canonical dispatch-sequences, *i.e.*, it may be treated as a primitive despite technically consisting of multiple precisely temporally coinciding parts. In short, such internal repetitions are effectively no-ops with respect to promises. Therefore pruning them incurs no loss (*i.e.*, is completeness-preserving).

to independent simulations of the original action α:[3]

$$(\mathit{before}{+}, \mathit{setup}{+}, \hat{\beta}_{i,1}{+}, \ldots, \mathit{reset}, \mathit{after}{+}) * .$$

Hence (4), *i.e.*, furthermore infer and write (which is stronger than (4)):

$$(\mathit{before}{+}, \mathit{setup}{+}, \hat{\beta}_{i,1}{+}, \ldots, \mathit{reset}, \texttt{fin}\text{-}\mathit{do}{+}, \mathit{after}{+}) * .$$

From this point forward (up till the next case) we demonstrate, for reference, the significance of (1–4) to the over-arching argument.

(Uniqueness in Detail: Stepping up to Lemma 4.17). So we desire here to show correctness of the compilation in this case, in gory detail, for sake of example with respect to the remaining cases. (Which is technically unnecessary, as the common argument already given suffices.) Then forget, for example and for the moment, that we know *(uniqueness)* of *reset* already; assume only (1–4), *(existence)*, and *(ordering)*. So, from such we need to conclude *(uniqueness)* and *(duration)*. For the contents $\hat{\beta}$ the former is a side-effect of the latter. For *(uniqueness)* of the rest (the book-keeping actions) we employ arguably *ad hoc* arguments. For *(duration)* of the book-keeping, we rely on finesse applied long ago (namely, we ensured that repeats of real actions must occur far enough apart in time). Incidentally, (3) and (4) directly subsume (1) and (2); the properties (1) and (2) are useful as steps towards establishing (3) and (4).

Note that the first instance of *reset* always follows the first instance of *setup*. It follows that every internal repetition of the envelope begins before (by (3)) and ends after (by (4)) the first instance of *setup*. So all internal repetitions are concurrent (as we may freely assume time-sortedness), meaning: there are no internal repetitions of the envelope. Then write:

$$(\mathit{before}{+}, \texttt{bgn}\text{-}\mathit{do}, \mathit{before}{*}, \mathit{setup}{+}, \hat{\beta}_{i,1}{+}, \ldots, \mathit{reset}{+}, \texttt{fin}\text{-}\mathit{do}, \mathit{reset}{*}, \mathit{after}{+}) * .$$

The only primitives mutex with a *before* are: itself, the start-part (and all-part) of the envelope, *setup*, and *after*. Consider internal repetitions of a *before* in the two intervals on either side of the start-part of the envelope (which is indeed unique per simulation of α as just shown).

[3]We will repeat this argument for any other purely virtual pieces of the compilation ("Local repetitions of primitives engaged in only token-passing are pointless and hence prunable without loss."), namely: the parts of the envelope, the four wrapping actions, and the struts inside of $\hat{\beta}$ can 'freely' be assumed to occur 'uniquely'. More accurately: non-dominated solutions provably exclude localized repetition.

From the landmarks: only itself could possibly occur in the interval from the first instance of *before* until the envelope starts. That is deordered-equivalent to an immediate repetition: hence a no-op, thus dominated, ergo prunable.

Then consider after the start of the envelope: so concurrent with the all-part. All instances of *before* are mutex with the all-part of the envelope (because they write to *do-occurred*). So no such instance occurs—in time—until after the all-part ends. As we may freely assume that solutions are time-sorted, no internal repetition of a *before* may occur—in dispatch-sequence—until after the end-part of the envelope. We already know the end-part is forced between a *reset* and an *after*, where we also know that instances of *before* cannot occur.

So, either way, effectively (*i.e.*, after pruning), only one instance of a *before* occurs (per simulation of α). Then say just "the *before*". For notation, write:

$$(\mathit{before}, \texttt{bgn-}\mathit{do}, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \texttt{fin-}\mathit{do}, \mathit{reset}*, \mathit{after}+) * .$$

By similar reasoning, the action *after* is effectively unique per simulation of α:

$$(\mathit{before}, \texttt{bgn-}\mathit{do}, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \texttt{fin-}\mathit{do}, \mathit{reset}*, \mathit{after}) * .$$

We already demonstrated that *reset* occurs effectively uniquely, but for contrast with the argument just given concerning internal repetitions of *before*, let us demonstrate *(uniqueness)* of *reset* again. Our second argument differs by starting from *(uniqueness)* of the envelope (which was demonstrated independently of *(uniqueness)* of *reset*, *i.e.*, the following is noncircular). The only primitives mutex with a *reset* are: itself, $\hat{\beta}_{i,k_i}$, the end-part of the envelope, *setup*, and *after*. Consider hypothetical repetitions on either side of the end-part of the envelope; in contrast with the argument concerning internal repetitions of *before* is the justification for the second case.

From the landmarks: only itself could possibly occur in the interval from the first instance of *reset* until the envelope ends. That is deordered-equivalent to an immediate repetition: hence a no-op, thus dominated, ergo prunable.

Then consider the interval after the envelope ends until the *after* occurs (both of which are unique by above). (Unlike above, a copy of *reset* could occur.) The *only* primitive that could occur in that interval, mutex with a *reset*, is itself. So the only primitive that could intervene between copies of a *reset* is the end-part of the envelope. The end-part of the envelope alters no fluents that *reset* writes to. Hence any copy of a *reset* in this latter interval is a no-op: thus dominated, ergo prunable.

In either case, *reset* is effectively unique. So write:

$$(\mathit{before}, \texttt{bgn-}\mathit{do}, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}, \texttt{fin-}\mathit{do}, \mathit{after}) * .$$

We could furthermore demonstrate that all internal repetitions of a *setup* are singleton by virtue of pruning no-ops. Such argument only works for virtual actions: actions that touch only book-keeping. (That is because we may identify all of their causal interactions and so carry out the analysis as above.) For *(uniqueness)* of the *real* parts of α we need to invoke temporal reasoning, which in particular grants *(uniqueness)* to all of the contents (real or virtual).

Specifically, by temporal reasoning, given the preceding: the contents of the envelope are unique per simulation of α. That is because repetitions increase the minimum duration between the start-part and end-part of the envelope beyond its maximum value (the duration of α plus two units for *setup* and *reset*). So write, for each i:

$$(\textit{before}, \texttt{bgn-}\textit{do}, \textit{setup}, \hat{\beta}_{i,1}, \ldots, \textit{reset}, \texttt{fin-}\textit{do}, \textit{after}) * .$$

In other words, we have *(uniqueness)* for the whole compilation of α.

Note that, long ago, we ensured that excess book-keeping lying outside the duration of α is nonproblematic (by ensuring that copies of α must occur sufficiently far enough apart in time). So *(duration)*—that everything starts when it is supposed to—is true of the excess book-keeping by triviality. Meaning: there is no specific time at which *before* and *after* are forced/supposed to take place. Naturally it is important to ensure that there exists at least one legal time to dispatch them (hence this note). In fact there exists a plethora of such legal times; regarding the slightly undesirable aspect of which, consider the following points.

A sufficiently clever—and little cleverness is called for—implementation of slackless-equivalence will make up a canonical time to dispatch the two. A human, for example, would likely right-shift *before* and left-shift *after*. That yields the tightest packing of the compilation of the original action α, taking up a total of $6\hat{\mu}$ excess duration (*i.e.*, before and after the start-time and finish-time of α itself). An automated planner might very well prefer to left-shift both or to right-shift both (or just think in purely lifted terms, and leave the variables largely unconstrained). We may also observe that accepting these degrees of freedom is necessary supposing we need the excess book-keeping at all. Specifically, attempting to compile in a mechanism for forcing the start-times of *before* and *after* falls into a recursion trap: they exist precisely to allow forcing of the start-times of *setup* and *reset*. So conversely, if one can circumvent the trap, then one can presumably circumvent the excess book-keeping altogether (which is certainly possible in non-minimal languages).

Anyways, returning to the main line: recall that *(uniqueness)* of the contents was just a side-effect of all the parts being forced to occur at the right times. In particular also *(duration)* holds of the contents, and thus of the whole compilation. Then we

have all of *(existence)*, *(ordering)*, *(duration)*, and *(uniqueness)*: correctness of the compilation is shown, in great detail, for this particular case of Lemma 4.17.

So for the remaining three cases we shall walk through only the bare bones.

Case: `L(pre; eff; eff)`. For (2), we ensure, by the all-part condition on *between-before-and-after*, that the envelope never occurs in any other context than with a surrounding pair of *before/after*. The general storyline is to use (3) and (4) to conclude *(uniqueness)* of the envelope; in fact, in this and every following case, we establish and use *(uniqueness)* of the envelope to conclude the otherwise challenging bits of (3) and (4). (The common argument will then reprove the property, which is unnecessary rather than circular.) Specifically, by using tracking effects at the envelope's endpoints, we ensure that (a) one instance occurs, and (b') the first occurs where desired. The rest of (3) and (4), that (b) *all* instances occur where desired, then follows trivially from at-most-onceness. Elaborating, we directly establish *(uniqueness)* of the envelope (at-most-onceness subject to completeness-preserving pruning) by way of the destructive effect on *between-before-and-end* of its end-part. The book-keeping:

- Create a boolean fluent *do-start-occurred*, initially and finally true.
- Create a boolean fluent *do-end-occurred*, initially and finally true.
- Create a boolean fluent *between-before-and-after*, initially and finally false.
- Create a boolean fluent *between-before-and-end*, initially and finally false.
- Add *do-start-occurred* := `True` to the start-part of *do*.
- Add *do-start-occurred* = `True` to *setup*.
- Add *do-start-occurred* := `False` to *before*.
- Add *do-end-occurred* := `True` to the end-part of *do*.
- Add *do-end-occurred* = `True` to *after*.
- Add *do-end-occurred* := `False` to *reset*.
- Add *between-before-and-after* = `True` to the all-part of *do*.
- Add *between-before-and-after* := `True` to *before*.
- Add *between-before-and-after* := `False` to *after*.
- Add *between-before-and-end* = `True` to *setup*, every $\hat{\beta}$, and most importantly to *reset*.

- Add *between-before-and-end* ≔ `True` to *before*.
- Add *between-before-and-end* ≔ `False` to the end-part of *do*.

From before the case analysis:

$$(\mathit{before}+, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \mathit{after}+) * .$$

Particularly, when any of those occur, then they all occur (in that order).

If an envelope occurs, then so does a prior instance of *before* by the dynamics of *between-before-and-after* (*i.e.*, given by *before* and needed by the all-part of the envelope). The corresponding *after* must follow the envelope in question, again by the dynamics of *between-before-and-after* (*i.e.*, taken away by *after* and needed by the all-part of the envelope).

Conversely, whenever an *after* occurs, then an envelope precedes it by the dynamics of *do-end-occurred* (*i.e.*, given by the end-part of the envelope and needed by *after*).

Hence solutions conform to (2):

$$(\mathit{before}+, \mathit{do}+, \mathit{after}+) * .$$

So (3) and (4) remain.

Consider (further) the tracking effects at the endpoints of the envelope, *i.e.*, consider the dynamics of *do-start-occurred* and *do-end-occurred*. Specifically, we may separately infer first that an envelope must start between *before* and *setup*, and secondly that an envelope must end between *reset* and *after*. As noted, such are most of (3) and (4) but for forcing *all* instances of the envelope to occur where desired. In fact we cannot force that aspect of (3) and (4) except by first ensuring the envelope occurs, effectively, uniquely.

We can be sure that the first instance of the envelope to start does so before *setup*, simply because there is no earlier than earliest. We cannot be so easily sure that the first instance of the envelope to end does so where we desire it to. To see the desired guarantee consider the dynamics of *between-before-and-end*. As the first envelope to end takes it away (which only *before* gives/restores), and all the intended contents require it, then specifically we have that the first envelope to end does so following the last *reset*. Then all later instances of the envelope start after the last *reset*—recall that actions are self-mutex—and so in particular all later envelopes are empty. Putting it all together, for notation:

$$(\mathit{before}+, \texttt{bgn-}\mathit{do}, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \texttt{fin-}\mathit{do}, \mathit{do}+, \mathit{after}+) * .$$

Note that all of the repetitions of the envelope are no-ops, hence prunable. So simplify:

$$(\mathit{before}+, \texttt{bgn-}\mathit{do}, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \texttt{fin-}\mathit{do}, \mathit{after}+) * .$$

Such is stronger than (2–4), thus this case is complete.

Case: `L(∅; pre; eff)`. For the ordering constraints of (3), we re-use a mechanism from the first case: we constrain where any instance of the envelope could possibly begin by use of conditions in its start-part (*i.e.*, on *before-possible* and *setup-possible*). For at-least-onceness of all of (2–4) we reuse the mechanism of a tracking effect in the end-part of the envelope. Said dynamics of *do-end-occurred* considered more generally also give most of the ordering constraints of (4). As in the second case, we finish off (4) by establishing at-most-onceness through the mechanism of *between-before-and-end*. The machinery:

- Create a boolean fluent *do-end-occurred*, initially and finally true.
- Add *do-end-occurred* := `True` to the end-part of *do*.
- Add *do-end-occurred* = `True` to *after*.
- Add *do-end-occurred* := `False` to *reset*.
- Add *before-possible* = `True` to the start-part of *do*.
- Add *setup-possible* = `True` to the start-part of *do*.
- Create a boolean fluent *between-before-and-end*, initially and finally false.
- Add *between-before-and-end* = `True` to *setup*, every $\hat{\beta}$, and most importantly to *reset*.
- Add *between-before-and-end* := `True` to *before*.
- Add *between-before-and-end* := `False` to the end-part of *do*.

From before the case analysis:

$$(\mathit{before}+, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \mathit{after}+) * .$$

Particularly, when any of those occur, then they all occur (in that order).

By the condition *do-end-occurred* = `True` of *after* we know the envelope occurs at least once between corresponding instances of *before* and *after*. In the other direction, whenever the envelope occurs: its start-part forces a previous instance of

before by the condition on *setup-possible* (which is given only by *before*). The rest of (2) follows (because a *before* forces a corresponding *after*), so write:

$$(\textit{before}+, \textit{do}+, \textit{after}+) * .$$

Then (3) and (4) remain.

Note:

- The condition *setup-possible* = `True` holds only between the firsts of *before* and *reset*.
- The condition *before-possible* = `True` holds only between the firsts of the previous iteration's *after* (or the initial state) and *setup*.

So every start-part of an envelope occurs between the firsts of *before* and *setup*. As similarly argued above, instances of *before* following the start-part of the envelope and preceding *setup* are no-ops (as are immediate repetitions), hence prunable. Then in particular we have, by pruning, that `bgn`-*do* occurs between the last *before* and first *setup*. In other words we have (3), so write:

$$(\textit{before}+, \texttt{bgn}\text{-}\textit{do}+, \textit{setup}+, \hat{\beta}_{i,1}+, \ldots, \textit{reset}+, \textit{after}+) * .$$

For (4), consider *do-end-occurred* in greater depth. Specifically, note that at least one such end-part occurs between *reset* and *after*. We do not yet, but will very shortly, know that such holds of the first (and all) envelope(s) to end.

Recall the mechanism of *between-before-and-end*: all contents fall within the first envelope, because the first envelope takes it away (which only *before* gives back), and all contents need it. In this case there cannot be later instances of the envelope as we already have all of (3), in particular, the start-part cannot follow a *reset*. We could also argue that later instances of the envelope, by virtue of containing nothing, would be no-ops and hence prunable.

Regardless, combining both points: the only (and so trivially first and all) instance(s) of the envelope end where desired, establishing (a result stronger than) all of (4). Specifically write:

$$(\textit{before}+, \textit{setup}+, \hat{\beta}_{i,1}+, \ldots, \textit{reset}+, \texttt{fin}\text{-}\textit{do}, \textit{after}+) * .$$

Moreover, combining with (3), write:

$$(\textit{before}+, \texttt{bgn}\text{-}\textit{do}, \textit{setup}+, \hat{\beta}_{i,1}+, \ldots, \textit{reset}+, \texttt{fin}\text{-}\textit{do}, \textit{after}+) * .$$

Such is enough to complete the case.

Case: `L(∅; eff; pre)`. Conceptually the case ought to be simply converse to the preceding. However, the mechanisms actually employed differ somewhat;[4] specifically, while playing the same role (ensuring at-most-onceness), the dynamics of *setup-occurred* and *between-before-and-end* do not seem to be precisely converse. Said mechanisms:

- Create a boolean fluent *do-start-occurred*, initially and finally true.
- Add *do-start-occurred* := `False` to *before*.
- Add *do-start-occurred* := `True` to the start-part of *do*.
- Add *do-start-occurred* = `True` to *setup*.
- Create a boolean fluent *setup-occurred*, initially and finally true.
- Add *setup-occurred* := `False` to the start-part of *do*.
- Add *setup-occurred* := `True` to *setup*.
- Add *setup-occurred* = `True` to the end-part of *do*.
- Add *setup-occurred* = `True` to *after* as well.
- Add *reset-possible* = `True` to the end-part of *do*.
- Add *after-possible* = `True` to the end-part of *do*.

Excluding the envelope, from before the case analysis, if anything, then everything: $(\mathit{before}+, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \mathit{after}+)*$. Specifically, if anything else besides the envelope occurs, then a *setup* occurs. From which, infer that also the envelope must start (hence occur as a whole); such is by the dynamics of *do-start-occurred* (*i.e.*, `bgn`-*do* gives, *setup* needs, and *before* takes).

Conversely, if the envelope occurs, and so ends, then (because *reset-possible* and *after-possible* can only be simultaneously true between the firsts of *reset* and *after*) note that everything else occurs as well (in particular *before* and *after* are forced).

Hence (2):

$$(\mathit{before}+, \mathit{do}+, \mathit{after}+) * .$$

So (3) and (4) remain.

[4]Perhaps the implied proof-improvement is impossible? After all, *causality* is not a wholly symmetric notion.

Note that by digging deeper into the preceding we can already extract most of (3) and (4). As in all but the first case, it suffices to ensure *(uniqueness)* of the envelope to obtain the rest of (3) and (4).

Elaborating, begin by observing that every envelope to end does so in a fairly specific spot in the dispatch-sequence: between the firsts of *reset* and *after*. Specifically the first envelope to end does so there; so any hypothetical repeat of the envelope follows. Then note that from the dynamics of *setup-occurred*—which the envelope sets to false—any hypothetical repeat of the envelope is a contradiction of solutionness. (Doing so is executable, but, leads to a dead-end.) More specifically that is because *setup* is never executable (with respect to the reachable situations of course) between the firsts of *reset* and *after*; the latter of which requires that *setup* take place.

So the envelope occurs at most once per repetition of the action α, from which (3) and (4) in particular easily follow. Slower, the following stronger statement is easily had from the notes thus far made concerning the case:

$$(\mathit{before}+, \texttt{bgn-}\mathit{do}, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \texttt{fin-}\mathit{do}, \mathit{after}+) * .$$

In contrast, (3) and (4) together only guarantee the nominally weaker assertion (which nominally permits repeats of the envelope):

$$(\mathit{before}+, \texttt{bgn-}\mathit{do}+, \mathit{setup}+, \hat{\beta}_{i,1}+, \ldots, \mathit{reset}+, \texttt{fin-}\mathit{do}+, \mathit{after}+) * .$$

Regardless (*i.e.*, starting from either assertion), the common argument for establishing correctness of the compilation proceeds readily. (For reference, *correctness* is written as, for each i: $(\mathit{before}, \texttt{bgn-}\mathit{do}, \mathit{setup}, \hat{\beta}_{i,1}, \ldots, \mathit{reset}, \texttt{fin-}\mathit{do}, \mathit{after})*$.) In any event, correctness follows, which suffices for the case.

So we are done, as all cases are complete. □