Honeywell Technology Center Technical Report SST-R97-030
Minneapolis, MN, October 1997
Authors: David J. Musliner and Robert P. Goldman and Kurt D. Krebsbach and Mark S. Boddy

Distributed CIRCA: Guaranteeing Coordinated Behavior in Distributed Real-Time Domains (D-CIRCA)

Data Item Number A003
Final Report

October 1997

Project Monitor: Mr. Henry Girolamo

Soldier Systems Command Acquisition Directorate
U.S. Army Natick Research, Development and Engineering Center
ATTN: SSCNC-P (Mr. Frank Scipione)
Kansas Street
Natick, MA 01760–5000

Contract Number DAAK60–94–C–0040

Honeywell Inc.
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

Technical Contact:
Dr. David J. Musliner
david.musliner@honeywell.com
(612) 951-7599

Administrative Contact:
Jim Nelson
Nelson_James@htc.honeywell.com
(612) 951-7490

ABSTRACT

This is the final report for DARPA contract DAAK60-94-C-0040-P0006 entitled "Distributed CIRCA: Guaranteeing Coordinated Behavior in Distributed Real-Time Domains." The goal of this contract effort was to begin extending the Cooperative Intelligent Real-Time Control Architecture (CIRCA) into distributed, multiagent operations. CIRCA is a coarse-grain architecture designed to control autonomous systems which require both intelligent, deliberative planning activity and highly reliable, hard-real-time reactions to safety threats. CIRCA allows systems to provide performance guarantees that ensure they will remain safe and accomplish mission-critical goals while also intelligently pursuing long-term, non-critical goals. The D-CIRCA project is a first step towards extending this type of intelligent and guaranteed behavior to multiagent systems. Major issues investigated during this project include planning with incomplete information, concurrent plan generation, and runtime plan synchronization. The new Dynamic Abstraction Planning (DAP) algorithm, which solves several of the key D-CIRCA planning problems, represents the primary scientific product of the D-CIRCA contract. This report describes how DAP was conceived, designed, and implemented. A detailed evaluation of DAP's performance on several different types of planning problems shows the dramatic improvement DAP provides over the original CIRCA state-space planner.

CONTENTS

LIST OF FIGURES v

PREFACE vii

SUMMARY 1

1. Introduction 3

2. Overview of CIRCA 3

3. Distributed CIRCA Issues 7

3.1. Thought Experiment 1 9

3.2. Thought Experiment 2 10

3.3. Thought Experiment 3 10

3.4. Local Planning with Simple Heuristics plus Plan Comparison 11

3.5. Alternative Distributed Planning Options 12

3.6. Planning in "Don't Care" Situations 12

4. Linking Distribution and Abstraction 13

5. Dynamic Abstraction Planning 13

5.1. DAP in Theory 15

5.2. DAP in Practice 16

5.2.1. Limiting the Set of Split Choices 18

5.2.2. Limiting the Set of Action Choices 20

5.2.3. Interaction with Temporal Reasoning 21

5.2.4. Intelligent Backtracking 21

5.2.5. Heuristic Control 24

5.2.6. Search Incompleteness 28

6. Evaluation 29

6.1. Eval-1 Domain: Benign Events 30

6.2. Eval-2 Domain: Uncertainty in Initial Conditions 32

6.3. Eval-3 Domain: Required Events 35

6.4. Eval-4 Domain: Complex Event Interleaving 40

6.5. Eval-5 Domain: Eval-4 with No Deleter Events 41
6.5.1. Implications of Heuristic Behavior on the DAP Planner 43
6.5.2. Implications of Heuristic Behavior on the Classic Planner 45
6.6. Eval-6 Domain: Expensive Splitting on Goals 46

7. Future Directions 47
7.1. Extending CIRCA's Expressiveness 48
7.2. More Flexible State Refinement 49
7.3. Improving the Heuristic Function 49

8. Conclusions 50

Appendix A. The CIRCA Temporal Model 53
A.1. Notation 54
A.2. Transition Timing 55
A.2.1. Events and Temporals 55
A.2.2. Actions 55
A.3. Definitions 56
A.4. Algorithm 58

Appendix B. Limitations of CIRCA 59

LIST OF FIGURES

1. The Cooperative Intelligent Real-Time Control Architecture. 4
2. The simulated Puma robot arm domain. 4
3. Example transition descriptions given to CIRCA's planner. 5
4. Sample output from the TAP compiler. 6
5. Summary of the CIRCA planning process. 7
6. Multiple D-CIRCA agents control a team of autonomous rotorcraft under supervisory control. 8
7. A partially-completed CIRCA plan. 15
8. A refinement of the NFA in Figure 7. 15
9. The DAP planning algorithm. 17
10. The three reasons to refine an abstract state. 18
11. Two successive splits, in order to enable an action with preconditions P, Q, R. 19
12. A more complete version of the DAP planning algorithm. 22
13. A sample operator-proposition graph. 26
14. A case where poor choice of next state causes failure to find a plan. 28
15. The 3-goal, 3-event Eval-1 domain. 31
16. DAP plan for the Eval-1 domain with 3 goals and 3 benign events: no state-space explosion. 32
17. Classic CIRCA plan for the Eval-1 domain: all combinations of events. . . . 33
18. DAP avoids state-space explosion on Eval-1 domains. 34
19. Classic CIRCA's runtime is exponential in the number of benign events. . . . 34
20. DAP's runtime on Eval-1 domains is linear in the number of goals and benign events. 35
21. Plans for Eval-2 domain with 3 goals and 3 initial states. 36
22. Plan size for Eval-2 domains with uncertainty in initial conditions. 36
23. DAP solves a 3-goal Eval-3 domain with sub-optimal split choices. 38
24. Plan size for Eval-3 domains with required events. 39
25. Runtime for Eval-3 domains with required events. 39
26. Detailed state space performance for a 4-goal Eval-3 domain. 40
27. Plans for Eval-4 domain with 3 goals. 42
28. Eval-4 domain shows DAP using non-homogeneous abstraction to advantage. 43
29. Plans for Eval-5 domain with 4 goals and no external predicate deleter events. 44
30. Eval-5 domains with no deleter events shows Classic CIRCA outperforming DAP planner. 45

31. A simple domain illustrating the difficulties of prepositioning for fortuitous events. 46
32. Eval-6 domains with many *goals* shows Classic outperforming DAP. . . . 47
B.1. A simple problem unsolvable by the CIRCA planner. 60
B.2. A trace of the execution of the plan given in Figure B.1. 61

PREFACE

The work reported here was conducted by the Honeywell Technology Center, Minneapolis, MN, during the period May 1996 through October 1997 under U.S. Army Soldier Systems Command, Natick Research, Development and Engineering Center contract DAAK60-94-C-0040-P0006. Henry Girolamo, a member of the Advanced Systems Concepts Directorate, was project officer for the contract.

DISTRIBUTED CIRCA: GUARANTEEING COORDINATED BEHAVIOR IN DISTRIBUTED REAL-TIME DOMAINS

SUMMARY

This is the final report for the Defense Advanced Research Projects Agency (DARPA) contract DAAK60-94-C-0040-P0006 entitled "Distributed CIRCA: Guaranteeing Coordinated Behavior in Distributed Real-Time Domains." The goal of this contract effort was to begin extending the Cooperative Intelligent Real-Time Control Architecture (CIRCA) into distributed, multiagent operations. CIRCA is a coarse-grain architecture designed to control autonomous systems which require both intelligent, deliberative planning activity and highly reliable, hard-real-time reactions to safety threats. CIRCA allows systems to provide performance guarantees that ensure they will remain safe and accomplish mission-critical goals while also intelligently pursuing long-term, non-critical goals. The D-CIRCA project is a first step towards extending this type of intelligent and guaranteed behavior to multiagent systems. Major issues investigated during this project include planning with incomplete information, concurrent plan generation, and runtime plan synchronization.

The new Dynamic Abstraction Planning (DAP) algorithm, which solves several of the key D-CIRCA planning problems, represents the primary scientific product of the D-CIRCA contract. The DAP technique provides major advantages, including:

- The selection of which features to "abstract away" is performed automatically during planning.
- The abstractions are *local*, in the sense that different parts of the state space may be abstracted to different degrees.
- The abstractions preserve guarantees of system safety.
- The planning system need not plan to the level of fully-elaborated states to construct a feasible, executable plan.

In all but the most contrived domains, DAP dramatically outperforms the original CIRCA state-space planner, generating much smaller state spaces and final plans using much less computation time. DAP represents a significant new contribution to the planning field, bringing practical automated abstraction to bear on the complexity problems that have long prevented successful application of this technology. We anticipate that further development and refinement of the DAP concept will lead to major improvements in our ability to apply planning technology to practical, large-scale domains.

1. Introduction

This is the final report for DARPA contract DAAK60-94-C-0040-P0006 entitled "Distributed CIRCA: Guaranteeing Coordinated Behavior in Distributed Real-Time Domains." The goal of this contract effort was to begin extending the Cooperative Intelligent Real-Time Control Architecture (CIRCA) into distributed, multiagent operations. CIRCA is a coarse-grain architecture designed to control autonomous systems which require both intelligent, deliberative planning activity and highly reliable, hard-real-time reactions to safety threats. CIRCA allows systems to provide performance guarantees that ensure they will remain safe and accomplish mission-critical goals while also intelligently pursuing long-term, non-critical goals. The D-CIRCA project is a first step towards extending this type of intelligent and guaranteed behavior to multiagent systems.

Section 2 provides an overview of the single-agent CIRCA architecture [6], and Section 3 explores the issues involved in extending the CIRCA concept to multiagent domains. Section 4 shows how several of the key CIRCA planning problems introduced by distribution can be solved by the appropriate use of abstraction [10], and motivates our focus on the new Dynamic Abstraction Planning (DAP) algorithm that represents the primary scientific product of the D-CIRCA contract [7]. Section 5 describes how DAP was conceived, designed, and implemented during the latter half of the contract. Section 6 provides a detailed evaluation of DAP's performance on several different types of planning problems, showing the dramatic improvement DAP provides over the original "Classic" CIRCA state-space planner. Section 7 explores future directions for research on CIRCA, and Section 8 concludes with a brief summary of the contract's results.

Appendix A describes the detailed temporal model that CIRCA uses to reason about real-time behaviors. While understanding this model is critical to planning for real-time mission-critical domains, we have described and evaluated DAP in a broader planning context that does not require the temporal reasoning component. Appendix B discusses a very narrow class of plans that neither CIRCA planner can succesfully generate, although the plan itself is compatible with both the domain modeling and plan execution semantics.

2. Overview of CIRCA

CIRCA is designed to support both hard real-time response guarantees and unrestricted Artificial Intelligence (AI) methods that can guide those real-time responses. Figure 1 illustrates the architecture, in which an AI subsystem (AIS) reasons about high-level problems that require its powerful but potentially unbounded planning methods, while a separate real-time subsystem (RTS) reactively executes the AIS-generated plans and enforces guaranteed response times. The AIS and Scheduler modules cooperate to develop executable reaction plans that will assure system safety and attempt to achieve system goals when interpreted by the RTS.

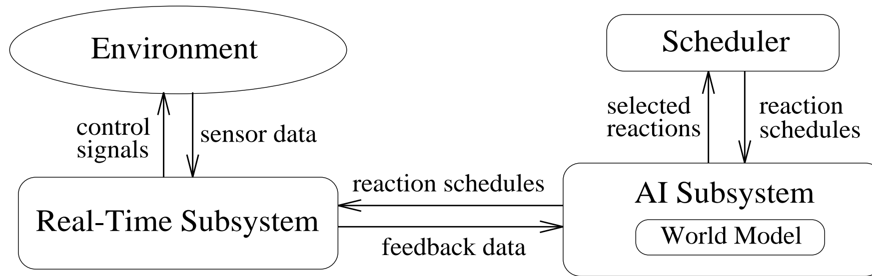


Figure 1. The Cooperative Intelligent Real-Time Control Architecture.

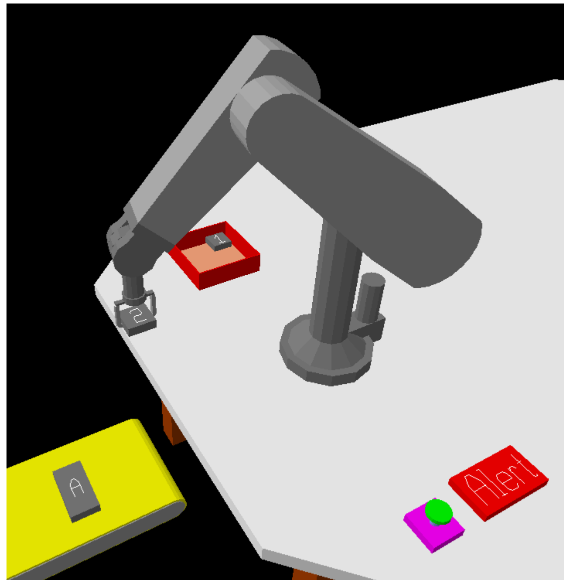


Figure 2. The simulated Puma robot arm domain.

CIRCA has been applied to real-time planning and control problems in several domains including mobile robotics and simulated autonomous aircraft. In this paper we draw examples from the domain illustrated by Figure 2, in which CIRCA controls a simulated Puma robot arm that must pack parts arriving on a conveyor belt into a nearby box. The parts can have several shapes (e.g., square, rectangle, triangle), each of which requires a different packing strategy. The control system may not initially know how to pack all of the possible types of parts— it may have to perform some computation to derive an appropriate box-packing strategy. The robot arm is also responsible for reacting to an emergency alert light. If the light goes on, the system must push the button next to the light before a fixed deadline.

In this domain, CIRCA's planning and execution subsystems operate in parallel. The AIS reasons about an internal model of the world and dynamically programs the RTS with a planned set of reactions. While the RTS is executing those reactions, ensuring that the system avoids failure, the AIS is able to continue executing heuristic planning methods to find the next appropriate set of reactions. For example, the AIS may derive a new box-packing algorithm that can handle a new type of arriving part. The derivation of this new algorithm does not need to meet a hard deadline, because the reactions concurrently

```
EVENT emergency-alert                            ;; Emergency light goes on
      PRECONDS: ((emergency nil))
      POSTCONDS: ((emergency T))

TEMPORAL emergency-failure                       ;; Fail if don't attend to
      PRECONDS: ((emergency T))                  ;; light by deadline
      POSTCONDS: ((failure T))
      MIN-DELAY: 30 [seconds]

ACTION push-emergency-button
      PRECONDS: ((part-in-gripper nil))
      POSTCONDS: ((emergency nil) (robot-position over-button))
      WORST-CASE-EXEC-TIME: 2.0 [seconds]
```

Figure 3. Example transition descriptions given to CIRCA's planner.

executing on the RTS will continue handling all arriving parts, just stacking unfamiliar ones on a nearby table temporarily. When the new box-packing algorithm has been developed and integrated with additional reactions that prevent failure, the new schedule of reactions can be downloaded to the RTS.

CIRCA's planning system builds reaction plans based on a world model and a set of formally-defined safety conditions that must be satisfied by feasible plans [11]. To describe a domain to CIRCA, the user inputs a set of transition descriptions that implicitly define the set of reachable states. For example, Figure 3 illustrates several transitions used in the Puma domain. These transitions are of three types:

Action transitions represent actions performed by the RTS.
Temporal transitions represent the progression of time and continuous processes.
Event transitions represent world occurrences as instantaneous state changes.

The AIS plans by generating a nondeterministic finite automaton (NFA) from these transition descriptions. The AIS assigns to each reachable state either an action transition or `no-op`. Actions are selected to *preempt* transitions that lead to failure states and to drive the system towards states that satisfy as many goal propositions as possible. A planned action preempts a temporal transition when the action will definitely occur before the temporal transition could possibly occur. The assignment of actions determines the topology of the NFA (and so the set of reachable states): preemption of temporal transitions removes edges and assignment of actions adds them. System safety is guaranteed by planning action transitions that preempt *all* transitions to failure, making the failure state unreachable [11]. It is this ability to build plans that guarantee the correctness and timeliness of safety-preserving reactions that makes CIRCA suited to mission-critical applications in hard real-time domains.

```
#<TAP 10>
        Tests   : (AND (PART_IN_GRIPPER NIL) (EMERGENCY T))
        Acts    : push_emergency_button
        Max-per : 9984774
        Runtime : 2520010
#<TAP 9>
        Tests   : (AND
                    (PART_IN_GRIPPER NIL)
                    (EMERGENCY NIL)
                    (PART_ON_CONVEYOR T)
                    (NOT (TYPE_OF_CONVEYOR_PART SQUARE)))
        Acts    : pickup_unknown_part_from_conveyor
        Max-per : 12029856
        Runtime : 3540010
#<TAP 8>
        Tests   : (AND
                   (TYPE_OF_CONVEYOR_PART SQUARE)
                   (PART_IN_GRIPPER NIL)
                   (EMERGENCY NIL))
        Acts    : pickup_known_part_from_conveyor
        Max-per : 12029856
        Runtime : 3520010
```

Figure 4. Sample output from the TAP compiler.

The NFA is translated into a memoryless controller for downloading to the RTS. This is done through a two-step process. First, the action assignments in the NFA are compiled into a set of *Test-Action Pairs* (TAPs). The tests are a set of boolean expressions that distinguish between states where a particular action is and is not to be executed. Each TAP's test expression is derived by examining all of the planned actions and finding a logical expression that distinguishes between the states in which the current TAP's action is planned and the states in which other actions are planned. Some sample TAPs for the Puma domain are given in Figure 4.

Eventually, the TAPs will be downloaded to the RTS to be executed. The RTS will loop over the set of TAPs, checking each test expression and executing the corresponding action if the test is satisfied. The tests consist only of sensing the agent's environment, rather than checking any internal memory, so the RTS is asynchronous and memoryless.

However, before the TAPs can be downloaded, they must be assembled into a loop that will meet all of the planned deadlines, captured as constraints on the maximum period of the TAPs (see Figure 4). This second phase of the translation process is done by the scheduler. In this phase, CIRCA's scheduler verifies that all actions in the TAP loop will be executed quickly enough to preempt the transitions that the planner has determined need preempting. The tests and actions that the RTS can execute as part of its TAPs have

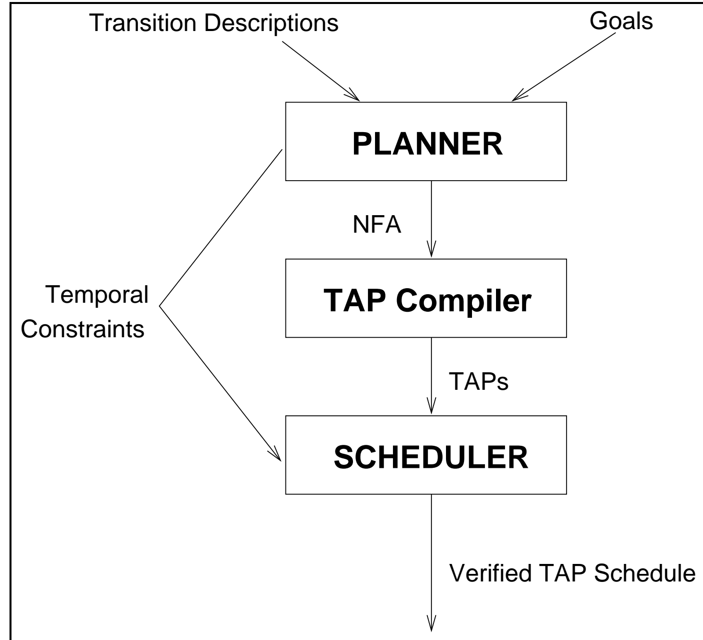


Figure 5. Summary of the CIRCA planning process.

associated worst-case execution times that are used to verify the schedule. If the scheduling does not succeed, the AIS will backtrack and force the state-space planner to revise the NFA, leading to a new set of TAPs and another scheduling attempt. The planning process is summarized in Figure 5.

3. Distributed CIRCA Issues

Early work on CIRCA focused on building an intelligent control system for a single agent, allowing that agent to provide real-time response guarantees while also using complex planning algorithms. The Distributed CIRCA (D-CIRCA) project extends the concepts of guaranteed safety and predictable performance into multiagent domains such as cooperating teams of autonomous aircraft (see Figure 6). D-CIRCA agents will communicate to allocate tasks and build executable real-time plans that achieve overall team goals. D-CIRCA will enforce both the *logical correctness* of coordinated multiagent behaviors and the *timeliness* of those behaviors, ensuring that coordinated actions achieve their goals and preserve overall system safety. While executing their plans, D-CIRCA agents will respond to ongoing events in real-time, invoking safety-preserving reactions and/or triggering dynamic replanning tailored to the current context.

This dual capability is distinctly different from typical distributed AI systems. Most DAI research evaluates collaboration and coordination methods based primarily on logical correctness and solution efficiency, ignoring the issues of behavioral synchronization and reaction timing required for *guaranteed* performance by a multiagent system. Systems based on the D-CIRCA architecture can be applied to mission-critical distributed domains with confidence, and will provide plan-time feedback when the available multiagent

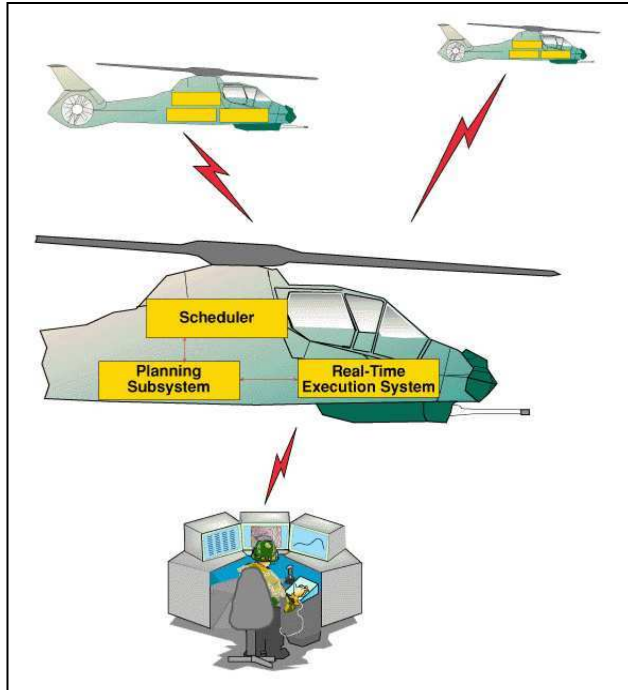


Figure 6. Multiple D-CIRCA agents control a team of autonomous rotorcraft under supervisory control.

resources are insufficient to deal with the anticipated behavior of the domain.

Extending the single-agent CIRCA model to multiagent applications raises many challenging issues, some common to all distributed AI applications and some uniquely the result of CIRCA's commitment to predictable, guaranteed real-time performance. For example, the problems associated with cooperating concurrent planners have been investigated in other work (e.g., Partial Global Planning [3]), but issues of predictable asynchronous plan execution and performance guarantees across team behaviors have not.

This section discusses important issues involving the D-CIRCA concept. Some of the more challenging aspects of these issues include the following:

- **Scalability:** State space explosion problems are exacerbated in distributed systems.
- **Action reflection:** When multiple agents collaborate, mechanisms are required to support group knowledge and action reflection (e.g., one agent may have to consider the possible ramifications of another's actions).
- **Guarantees with distributed scope:** Plans made up of the actions of multiple agents are difficult to guarantee as safe, particularly if the guarantor cannot know all of the actions of other agents, even those on the same team.
- **Runtime communication:** Agents must communicate about actions to keep plan execution synchronized.

We began exploring the D-CIRCA concept and the associated problems by considering a new domain, derived from the original Puma domain, in which two robot arms cooperate. While investigating this domain and the D-CIRCA concept, we considered several key questions, including:

- What domain characteristics make distributed planning with performance guarantees hard?
- What types of interagent communication can increase functionality within a domain without overwhelming the agents with constant (mostly irrelevant) message-passing?
- What types of assumptions can an agent make about other cooperating agents' behavior, without unduly constraining its own behavior?
- How useful is the concept of the pre-assignment of "roles" between agents.
- To what extent is it necessary or desirable to model world features of other agents?
- What is the tradeoff between world modeling, sensing, and communication?

To make our consideration of these questions less abstract, we performed some fairly small-scale but rigorous "thought" experiments. These experiments consisted of positing the agents with certain capabilities and the domain with certain constraints, and then manually performing the state-space planning to discover what further assumptions were involved, and what hidden problems might exist, in this form of distributed collaborative planning.

3.1. Thought Experiment 1

Imagine a Puma domain in which two CIRCA-controlled robots have the original domain's goals: they must attempt to keep objects from falling off the conveyor (safety goal), turn off the emergency light in time (another safety goal), and, if possible, pack the parts in a box (optional goal). There is a table near the conveyor where parts can be placed safely, but they can no longer be retrieved for packing once placed there. There are only three locations for the agents: conveyor, box, and light. Either agent can perform any task, with the constraint that it takes both of them to pack a part in the box (i.e., they must co-locate and cooperate to achieve the optional goal). Finally, they both share all world information (including each other's locations), except that they do not know if the other agent is holding something.

This experiment revealed that a very limited form of partial information (not knowing if the other agent is holding something) came at a great cost. In hand-simulating the planning process, we found that the agents had to either 1) assume the worst and make a lot of ultra-conservative moves, or 2) "get inside the other agent's head" in very domain-specific ways. An example of the first is to assume the other agent has a part when you don't want him to (e.g., when you want him to help you pack a part), and doesn't have one when you're in an advantageous location to help (e.g., at the box without a part). This assumption that the other, supposedly cooperating agent will always be doing things that

place the most severe constraints on your own behavior leads to extremely inefficient plans, and, in some cases, effectively negates any advantage gained by having multiple agents.

An example of the second, more insidious problem is an instance of the well-known, extremely difficult problem of intent inference. For example, based on some domain assumptions or prior agreements, Agent-A might reason that Agent-B would never be over the box if Agent-B hadn't just picked up a part, and Agent-B couldn't have just dropped one off without Agent-A's help, so Agent-A can infer that Agent-B must be holding a part. Therefore, when Agent-A sees Agent-B at the box, Agent-A should move there as soon as possible to help. All of this introspection is a means of inferring the world features that we don't want to bother representing, or, more likely, can't know by observation or communication, but nonetheless need in order to do an efficient job of planning. The obvious disadvantage is that to do the inference, potentially a great deal more information about the other agent's capabilities, tendencies, and agreements, in addition to further domain dynamics, must be available and represented to derive the missing information. Furthermore, unless this relevant information can be effectively represented declaratively, the assumptions will be implicit, and must be re-implemented for each new domain variation.

Our investigation of the issues involved in the multiagent Puma domain led us to consider several design alternatives, discussed in the following subsections.

3.2. Thought Experiment 2

In this experiment, we attempted to cut down the complexity of the state space by assigning loose "roles" to each agent (e.g., Agent-A *mainly* handles the alert light and Agent-B *mainly* handles the conveyor, unless they get in a jam). Unfortunately, when we consider this domain and try to make performance guarantees (as CIRCA always does), it quickly becomes obvious that this domain degenerates into the original domain without roles. Worst-case assumptions are the great equalizer: it's hard in the first place to find a domain where the decisions are difficult enough to be interesting and constrained enough to be computable, without having to worry about the worst-case assumptions being variable enough between agents/actions/domain dynamics that they don't completely flatten out that precarious balance.

3.3. Thought Experiment 3

We constrained the third experiment even further. Again, it takes both agents to pack a part (so there's some notion of synchronization, although no communication at present), and neither agent knows the other's holding status (so there's some partial world information to contend with). The new element is a set of fixed roles: one agent will *only* handle conveyor parts, and the other will *only* handle the emergency light. These stronger roles make planning considerably easier and more analyzable. For example, even though each agent is still physically capable of being in one of three locations, each will actually only choose to go to the two locations that are consistent with their roles. This type of

restriction can dramatically decrease the domain's state-space size. The challenge, however, is finding a way of deriving the restriction without examining the portions of the state-space that would be pruned out. Section 4 discusses how this type of observation motivates our work on Dynamic Abstraction Planning (DAP). The remainder of this section will concentrate on some general results, good ideas, and lessons learned from our thought experiments.

3.4. Local Planning with Simple Heuristics plus Plan Comparison

To build plans for the multiagent Puma domains, we assumed that each agent performed its own local planning in isolation. Then these plans were compared, state by state, to determine if there were harmful interactions (clobberers) between what one agent was planning to do and what another was planning. A manual conflict resolution strategy was applied to modify the plans when necessary. None of the plan comparison or modification steps have been automated; these experiments were designed, in part, to see how effective and efficient this "local planning with comparison" approach would be.

When performing the planning manually for each agent, we found it necessary to make some simple heuristic assumptions about what the other agent would do in certain states. For example, when planning for Agent-A in the state where both Agent-A and Agent-B were at the box and Agent-A was holding a part, we assumed that Agent-B would take the action of helping Agent-A pack the part. To be truly justified in making this assumption, Agent-A would have to know how soon the emergency light might go on (immediately, in the worst case), how long the synchronized packing takes (known), and how long the sequence of actions for Agent-B to turn off the light would take (which might be very hard to derive, since a sequence of Agent-B's states is involved). Using the "local planning with comparison" approach, Agent-A adopts some simple assumptions about Agent-B's behavior, rather than attempting to perform this very difficult prediction of the other agent's precise behaviors. Then, if those assumptions turn out to be wrong at plan-comparison time, the agents must deal with them at that point.

Similarly, when Agent-A is waiting at the box with a part, and a part shows up at the conveyor, Agent-A assumes that Agent-B cannot, in general, be trusted to come over and help Agent-A pack its part before Agent-A has to move back to the conveyor for the newly-arrived part. Ideally, Agent-A should wait as long as possible for Agent-B, only giving up and heading back to the conveyor when there is barely enough time to guarantee picking up the new part before a failure occurs. This is probably possible just with local knowledge, since Agent-A knows how long each of its actions should take in the worst case, and can simply add them up to figure out how long to wait. However, there is currently no way to express this in the CIRCA plan semantics without introducing an arbitrary, intermediate feature.

3.5. Alternative Distributed Planning Options

There are many alternatives to the "local planning with assumptions plus compare" approach to distributed planning. One alternative is to constantly trade partial plans, queries, constraints, and negotiations throughout the distributed planning process. This appears to have the disadvantage that it will deteriorate into a lock-step process, in which parallel planners must reason at the same time (synchronously) about each shared state, thus negating many of the advantages of multiagent planning. A second alternative is to have one agent construct its plan, then send it down the line to the next agent as a set of constraints on the second agent's planning process. This, of course, is less efficient, because the distributed planning is not in parallel, and also forces an ordering on which agents "get their way" first. Perhaps worst of all, this approach introduces the possibility of having to backtrack over multiple agents' plans.

3.6. Planning in "Don't Care" Situations

The thought experiments made clear that D-CIRCA's planner should select actions that satisfy the following criteria, in order of priority:

1. Preempt failure (strictly necessary).
2. Help achieve a goal (desirable but not strictly necessary).
3. Remain independent of what other agents plan.

Classic single-agent CIRCA already incorporates the first two criteria; the third is a simple, if incompletely defined, way to avoid the expense of multiagent conflict resolution. Criteria 2 and 3 are related in an interesting way: if an agent's choice of action does not depend on what others are doing, he can choose the action that is heuristically best for him, namely the one that is most goal-achieving. However, in cases where another agent *might* do something that affects the choice of action, the first agent must choose his own action based on a worst-case assumption, which is usually too conservative to be goal-achieving. This is similar to how single-agent CIRCA makes worst-case assumptions about nonvolitional transitions like events. However unlikely they are, they must be considered and safety must be guaranteed, but this can lead to highly sub-optimal, minimally-goal-achieving plans. In many ways, other agent's actions are just like nonvolitional transitions at the state-space modeling level. Our intuition tells us that we should have more powerful ways of reasoning about them via cooperative interagent negotiation. One challenge of the D-CIRCA concept, then, is to find ways of recognizing patterns in the state-space representation of multiagent interactions that can be resolved or improved by negotiation with the source of some of the transitions: other agents.

Again, in some cases this reduces to finding domains in which there is enough interaction between agents for some intelligence to be required, but not so much that the agents are checking with each other on every single decision. By removing one difficult aspect in Experiment 3 (overlap of capabilities), planning with fairly weak assumptions was possible,

but plan comparison and conflict resolution have yet to be tackled, and can certainly be expected to provide another raft of insights and problems as yet undiscovered.

4. Linking Distribution and Abstraction

Consider the following aspects of a multiagent system, when viewed from the perspective of a single agent within that system:

Partial Information — The agent cannot know everything about its environment or about the internal state of other active agents.

Incomplete Control — The agent cannot perfectly control all aspects of the domain or the actions of other agents.

Limited Time — The agent has a limited amount of time to reason about and react to various situations to maintain safety and achieve its goals.

The original CIRCA system was designed to address the latter two issues explicitly, but made strong assumptions about the world being "fully observable." To make guarantees that CIRCA would detect and react to all environmental hazards, it was assumed that the system could sense and deliberate about all modeled world features. In a distributed environment, there are several reasons to use a partially-observable model, including:

Local Views— Distributed systems are often motivated by the need for different agents to have heterogeneous sensors, locations (and hence fields of view), and other distinguishing capabilities.

Limited Communication— Agents cannot share all of their knowledge.

Bounded Rationality — Even if they could share all of their knowledge, agents cannot afford the state-space explosion associated with reasoning about the complete domain model.

An important observation motivating our research on Dynamic Abstraction Planning is that *a single agent capable of making performance guarantees based on an abstracted world model can successfully address each of the above distribution issues.*

By "abstraction," we mean the deliberate omission of certain pieces of detailed information from an agent's world model, and hence from its consideration. An abstracted world model is essentially indistinguishable from an incomplete world model, a local view, a partially-shared model, or a "partially considered" model. Distributed observability reduces to partial observability for a single agent, so our first challenge is to build a new CIRCA that retains its unique guaranteed performance characteristics while using abstract world models.

5. Dynamic Abstraction Planning

In a state-space model like CIRCA's, one of the most straightforward ways of using abstraction is to simply remove a feature from the description of the world. This

corresponds closely to the methods used in early work on abstraction planning systems to generate abstract operators by omitting less-critical elements of operator precondition lists (cf. ABSTRIPS [14]). ABSTRIPS planned at an abstract level that then restricted the extent of the detailed planning required to build a final plan. The Dynamic Abstraction Planning (DAP) technique is significantly different in that:

- The selection of which features to "abstract away" is performed automatically during planning.
- The abstractions are *local*, in the sense that different parts of the state space may be abstracted to different degrees.
- The abstractions preserve guarantees of system safety.
- The planning system need not plan to the level of fully-elaborated states to construct a feasible, executable plan.

The DAP concept is simple: rather than always using all of the available features to describe world states, we let the planner dynamically decide, for each new world state, the level of description that is necessary and desirable. By ignoring certain features, the planner can reason about *abstract states* that correspond to *sets* of "base-level" states, and thus can avoid enumerating the individual base-level states.

Of course, during the planning process the system might realize that an abstract state that has already been reasoned about is not sufficiently detailed. For example, the planner knows that an action can only be executed if all of its preconditions are known to hold in the state. Thus a state description may not be sufficiently refined to indicate whether a desirable action can, in fact, be executed. If an abstract state is insufficiently refined because its state description does not specify values for all of the features in a desired action's preconditions, the planner can dynamically increase the precision of that abstract state description by including one or more of the omitted features. We call this process of adding detail a "split" or "refinement."

In the language of finite automata, DAP starts with a very crude nondeterministic finite automaton (NFA) and dynamically adds more detail. DAP refines the NFA when it is unable to generate a satisfactory plan[1] at the current level of detail. DAP refines the NFA by taking an existing state and splitting it into a number of more specific states, one for each possible value of a particular feature, F_i.

For example, consider the partially-completed plan in Figure 7, which is based on the domain model given in Figure 3. Here there are three states: the failure state and two non-failure states, one for each value of `emergency`, a boolean proposition. We assume that `emergency` is `nil` when the system begins operation.

The NFA in Figure 7 is not safe, because there is a reachable state, S_1, from which there is a transition to the failure state (`emergency-failure`) that has not been preempted. One way to fix this problem would be to choose an action for S_1 that will preempt

[1]We will be more clear about what is "satisfactory" below.

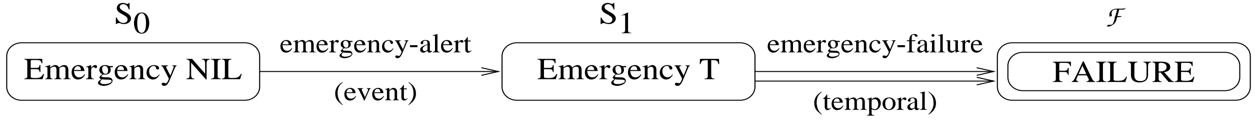


Figure 7. A partially-completed CIRCA plan.

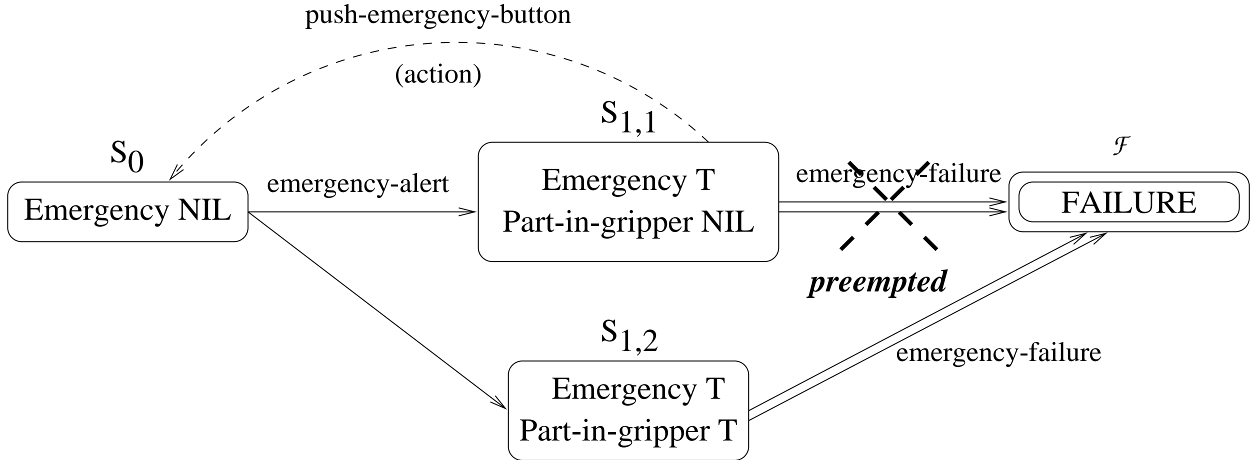


Figure 8. A refinement of the NFA in Figure 7.

`emergency-failure`. The domain description contains such an action, `push-emergency-button`. Unfortunately, one of `push-emergency-button`'s preconditions is `part-in-gripper`= `nil` and S_1 is too abstract to specify a value for `part-in-gripper`. We can only consider applying `push-emergency-button` by splitting S_1 into a set of states, one for each value of `part-in-gripper`. The resulting NFA is given in Figure 8. We can now assign `push-emergency-button` to solve the problem posed by state $S_{1,1}$. Further planning is required to resolve the problem posed by $S_{1,2}$, either by finding a preempting action that does not require `part-in-gripper` = `nil` or by making $S_{1,2}$ unreachable.

Note that DAP adds detail to the NFA only *locally*. In our example above, we only added the feature `part-in-gripper` to the state where the `emergency` feature took on the value `true`, rather than refining all of the states of the NFA symmetrically. This introduces new nondeterminism: because we do not have a complete model of the initial state, we cannot say whether the `emergency-alert` transition will send the system to state $S_{1,1}$ or $S_{1,2}$.

5.1. DAP in Theory

During its operation, DAP manipulates NFAs of a particular type. An NFA, $\mathcal{N} = \langle v(\mathcal{N}), e(\mathcal{N})\rangle$, will have a number of states (or vertices), $S_i \in v(\mathcal{N})$, each of which corresponds to a set of feature-value pairs; we will refer to these as $f(S_i)$. A state S_i *necessarily* satisfies a proposition, P ($S_i \models \Box P$) if $P \in f(S_i)$; it *possibly* satisfies P ($S_i \models \Diamond P$) if $\neg P \notin f(S_i)$ (these boolean definitions may be straightforwardly extended to non-boolean features).

The transitions in the NFA are generated by the transition descriptions, which are nondeterministic STRIPS operators. A transition t is possibly (respectively, necessarily)

executable in a state when the transition's preconditions are all possibly (necessarily) satisfied by that state: $S_i \models \Diamond pre(t)(\Box pre(t))$. With some abuse of notation, for each transition t we define a function $t(S)$ from a state to a formula (in the general case, a disjunction), describing the state(s) that result from executing t in S. DAP manipulates NFAs that contain edges for all possibly executable non-preempted event and temporal transitions (we refer to these collectively as "non-volitional transitions") and for all currently-assigned actions.

The refinement (or splitting) operation r on an NFA $\mathcal{N}$ with respect to a state S_i and a feature F_j, notated as $r(\mathcal{N}, S_i, F_j) = \mathcal{N}'$, is defined as follows:

$$\begin{aligned} S' &= \{S | f(S) = f(S_i) \cup \{(F_j, z)\} \text{ for } z \in val(F_j)\} \\ v(\mathcal{N}') &= (v(\mathcal{N}) - S_i) \cup S' \end{aligned}$$

where S' is the set of newly-added states. New transitions must be added into and out of the replacement states:

$$\begin{aligned} e(\mathcal{N}') = & \ (e(\mathcal{N}) - \{v_1 \rightarrow v_2 | v_1 = S_i \text{ or } v_2 = S_i\}) \\ & \cup \{v \xrightarrow{t} S | v \models \Diamond pre(t), S \in S', S \models \Diamond t(v)\} \\ & \cup \{S \xrightarrow{t} v | S \in S', S \models \Diamond pre(t), v \models \Diamond t(S)\} \end{aligned}$$

To build safe plans for CIRCA, DAP must construct NFAs in which there are no chains of non-preempted, possibly-executable transitions that lead to a failure state. To preempt a temporal transition in a state, DAP assigns to that state a *necessarily* executable action that can be executed before the preempted transition.

5.2. DAP in Practice

The prototype DAP planner takes as input a domain model in the form of transition descriptions, a description of a set of initial states, and a conjunctive goal expression. The planner returns an NFA containing only reachable states. Each state of the NFA will be labeled with either an action or `no-op`, indicating to CIRCA how the RTS should react in that situation. Failure states will not be reachable in this NFA and the system will move towards states satisfying the goal expression whenever possible.

The planning problem may be very concisely described as a nondeterministic algorithm, given in Figure 9. In this presentation, **choose** and **oneof** are nondeterministic choice operators. An action is applicable if the state necessarily satisfies its preconditions and if the action preempts all transitions to failure from the state. Note that it is not sufficient to preempt transitions directly to the distinguished failure state. For example, if there is a state s with an event transition (i.e., a transition with a zero delay) to the failure state, then any edges into s must also be considered as transitions to failure.

In practice, we implement this algorithm through search, with choice points corresponding to the nondeterministic choice operators. The search engine backjumps when it encounters

```
abstract-plan (isd);
    isd is initial state description
    let 𝒩 = ∅; The graph
        openlist = ∅;
        is = make-initial-state(isd);
    𝒩 := 𝒩 ∪ {is};
    push(is, openlist);
    loop
        if there are no more reachable states in the openlist then
          we are done
          break;
        else
          let s = choose a reachable state from openlist;
              openlist := openlist − {s};
              oneof
                  split-state :
                      choose a proposition p and split s into |val(p)| states;
                      remove s from 𝒩 and insert the new states;
                      add the new states to the open list;
                  assign-action :
                      choose an action (or no-op) that is applicable for s;
                  fail
```

Figure 9. The DAP planning algorithm.

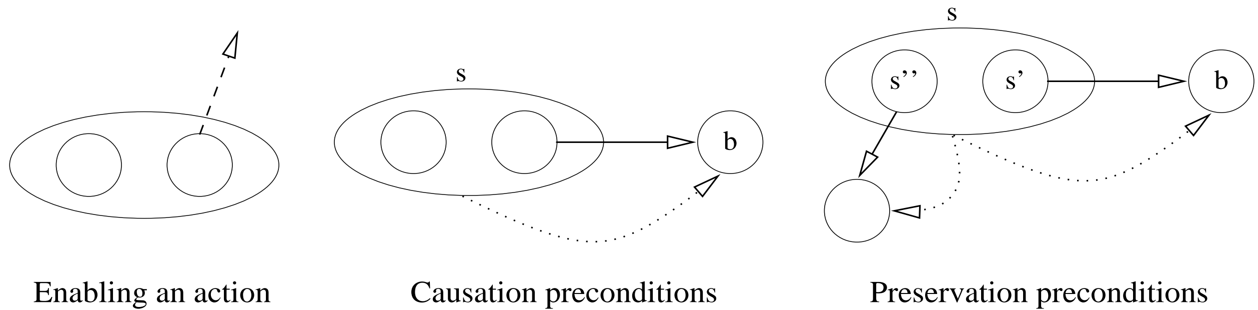


Figure 10. The three reasons to refine an abstract state. Ovals represent abstract states that are refined into two component states (circles). Dotted arrows are edges out of the abstract states, while solid arrows are edges out of the refined, less-abstract states.

a state for which there is no acceptable action and for which there is no proposition on which to split. We may not be able to split the state productively even if the state is only partially specified. No further splitting will be productive if we can determine that some bad transition *must* occur in the state, that the state is reachable, and that there are no available actions with which to preempt the bad transition.

5.2.1. Limiting the Set of Split Choices

We do not need to consider all possible propositions when deciding how (and whether to) split a state. Figure 10 illustrates the three reasons to add a proposition to a state description:

1. To enable an action;
2. To avoid a state by means of causation preconditions;
3. To avoid a state by means of preservation preconditions.

Enabling an Action We may add a proposition to a state in order to enable us to use an action. In this case, we may refine the state by a proposition, P, when P is the precondition of an action that is *possibly*, but not *necessarily* enabled: that is, an action none of whose preconditions is negated by the current state description. Note that while we have said that we do this to "enable us to use an action," in fact, what we are doing is applying a divide and conquer strategy: we have divided the previously abstract state into two new states, in one of which the action is possibly enabled and in one of which it is not, as Figure 10 shows. Also note that we still say "possibly" enabled, because it may take multiple splits before we have a state in which an action is necessarily enabled. For example, Figure 11 illustrates the case where we would like to use an action whose preconditions are P, Q, R in a state whose features are only P, and we have to split twice, once on Q and once on R (not necessarily in that order).

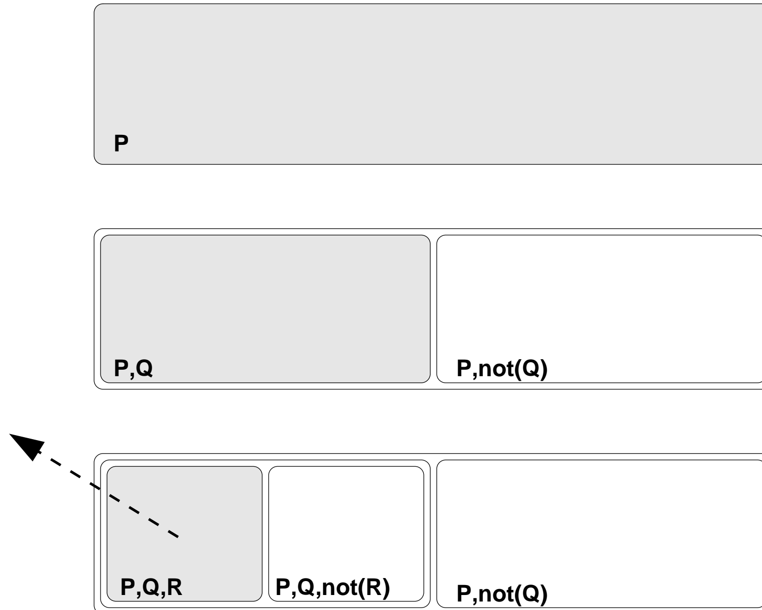


Figure 11. Two successive splits, in order to enable an action with preconditions P, Q, R.

We need not consider all possibly-enabled actions in this process: some actions can be identified as futile *a priori*. For example, if any temporal transitions must be preempted in the state, the distinguished `no-op` action is futile.[2] Likewise, if we have a temporal transition that must be preempted, then any action that is slower than that transition can be discarded.

Avoiding a State via Preconditions We may also refine a state in order to avoid another, undesirable state. Consider the case of a state s that has an outgoing edge to a bad state, b. This "out" edge is the result of a transition that is not under the planner's control: a temporal transition that it has decided not to preempt,[3] or an event. The successor state is known to be bad, either because it can be identified *a priori* as one that cannot be solved or because we have backtracked from that state (see Section 5.2.4). So the planner would like to make state b unreachable.

There are two ways the planner can use refinement of state s to make b unreachable. The first is to split on one of the causation preconditions for reaching b. The causation preconditions for b from s are the preconditions for the transition t that carries the NFA from state s to state b. If any of these preconditions are not *necessarily* satisfied by state s, then these are candidates for splitting. For example, if transition t's preconditions are P, Q, and R, and state s's description is $\{P\}$, then the remaining causation preconditions are Q and R, either of which is a candidate for use in refining s.

The second way to make b unreachable is to refine state s by including preservation

[2] Although it is possible that we could split the state into a set of more specific states, in some of which the `no-op` action would be useful.

[3] Of course, preemption decisions can be changed via backtracking.

preconditions for b. Preservation preconditions for b are those features of b that are not established by the transition t. For example, consider a state b, whose features are $\{P, S, T\}$. The transition t establishes S. The preservation preconditions for b with respect to t are P and T. Suppose P is among the set of features of state s, but we could refine state s by feature T. Assuming T is a boolean feature, that would refine state s into two states, s' and s''. The features of state s' would be $\{P, T\}$ and from it b would still be reachable. But s'' would have the features $\{P, \neg T\}$, and b would *not* be reachable from s'' by means of t. Note that in this case of refinement, unlike the previous case, transition t is still enabled in *both* successor states; however in one of the two successor states, the transition is now benign instead of malignant (cf. bottom of Figure 10).

Note that our use of the expression "refine state s to make state b unreachable" is actually a shorthand.[4] We refine state s into a set of states, from some of which b is not reachable and from some of which b is reachable. The states from which b is not reachable now present the planner with an easier problem to solve. Those states from which b is still reachable are solved by recursively applying these techniques, or are themselves made unreachable.

These methods for selecting which splits are useful may still leave a large number of choices. Our current implementation of DAP uses several heuristics to choose among actions and propositions for refinement. We discuss these in greater detail below (see Section 5.2.5).

5.2.2. Limiting the Set of Action Choices

The algorithm described in Figure 9 specifies that we must choose an "applicable" action. There are three considerations that go into determining whether an action is applicable:

1. The action's preconditions must be *necessarily* satisfied by the state;
2. The action must not lead to a failure state; and
3. The action must be fast enough to make all the required preemptions.

Case 1: Because actions will be planned to preempt failures, CIRCA insists that they be completely reliable and guaranteed. Therefore, DAP will not consider selecting an action at a state where it is not certain that the action can actually be executed.

Case 2: Actions may be found to lead to failure states during the planning process. For example, an action a executed in a state s might (possibly) lead to a state t, from which there is an event to failure. Events cannot be preempted, so state t is a failure state, and a will not be applicable in s.[5]

Case 3: We use local information to estimate whether or not an action will be fast enough to make all the required preemptions (we discuss below how this set of preemptions is computed). We can only *estimate* whether an action will be fast enough, because the speed of the action is a property of the plan as a whole, not a local property. The speed of the

[4] As with "refine state s to enable action a."

[5] In cases when a *possibly* leads to t, but does not *necessarily* lead to t, further planning may make it possible to use a after all.

action is a global property because the plan will be compiled into a schedule of test-action pairs. The maximum delay before an action will occur is a function of this schedule as a whole, rather than any individual action assignment.

5.2.3. Interaction with Temporal Reasoning

The algorithm given in Figure 9 is actually a simplification of the search performed by our implemented planner. In order to construct a correct and safe plan, the DAP planner must take temporal information into account, as noted in Case 3 above. The temporal model underlying CIRCA's plans is described in detail in Appendix A. Essentially, the model allows the planner to determine whether a planned action can be executed quickly enough to preempt an undesirable temporal transition.

To give the planner flexibility in deciding which temporals it wants to try to preempt, we add a new set of explicit search decisions that specify the set of temporals to be preempted. See Figure 12 for a revised version of the algorithm. Some of these temporals are easy to identify — the ones we call "TTFs" (temporal transitions to failure), that explicitly mention FAILURE as a postcondition. However, there are other temporals we might choose to preempt. For example, if we have a state S for which we cannot identify an applicable action, it may be necessary to render S unreachable. In that case, we will prefer to preempt any temporal that leads from a reachable state to S, even when these transitions are not explicitly TTFs.

Accordingly, before we resolve a state by splitting or assigning an action, we choose whether or not to preempt the temporals that are possibly applicable in that state. When computing whether an action is fast enough, the set of required preemptions is the set of TTFs applicable in the state together with the set of transitions T such that preempt(s,t) = true.

As discussed in Appendix A, we have developed a recursive algorithm, based on depth-first search, to determine how fast an action must be executed in order to make all required preemptions. Using the set of temporal transitions that must be preempted, we search the NFA to determine bounds on how much time there will be before those transitions are enabled in the current state. These bounds are computed based on worst-case estimates of how long the agent will remain in the predecessor states. These computations are efficient, but unfortunately, not sound, so that there are pathological cases when the planner believes an action is not fast enough when, in fact, it is. Appendix B discusses one type of plan illustrating this problem.

5.2.4. Intelligent Backtracking

The DAP search algorithm uses a version of backjumping that takes into account both temporal information and the topology of the NFA. Backtracking is also used to provide an anytime plan refinement capability.

Recall that, for each state, we have two kinds of choice points: there is a choice point for

```
abstract-plan (isd);
    isd is initial state description
    let 𝒩 = ∅; The graph
        openlist = ∅;
        is = make-initial-state(isd);
    𝒩 := 𝒩 ∪ {is};
    push(is, openlist);
    loop
        if there are no more reachable states in the openlist then
          we are done
          break;
        else
        † let s = choose a reachable state from openlist;
              openlist := openlist − {s};
              foreach t in the possibly applicable temporals for s
                ‡ oneof
                      preempt(s, t) = true; or
                      preempt(s, t) = false;
          †† oneof
                split-state :
                     choose a proposition p and split s into |val(p)| states;
                     remove s from 𝒩 and insert the new states;
                     add the new states to the open list;
                assign-action :
                     choose an action (or no-op) that is applicable for s;
                fail
```

Figure 12. A more complete version of the DAP planning algorithm.

each temporal that may optionally be preempted[6] and there is a choice point for resolving the state (refining it or assigning an action to the state). For each of these decisions, we keep a list of *eliminations* (we use Ginsberg's terminology [5]). These eliminations are sets of values that are not consistent with earlier decisions, together with a set of decisions that is sufficient to rule the value out.

For example, it may be impossible to make a preemption because a set of prior decisions creates a dependent chain such that the lower bound on the latency for the temporal is zero. In that case, the eliminating explanation will be the set of decisions (action assignments and preemption decisions) that make up the dependent chain.

Similarly, when attempting to assign an action to a state, we accumulate eliminating explanations for the action choices. These are generally in the form of preemptions that cannot be made. In this case, we find the sets of decisions that assembled a dependent chain the action cannot preempt. Sometimes, however, a state is simply unsolvable (no action choices are available, there is an event leading to failure, etc.). In this case, we backjump to the earliest solved state that has an edge into the failed state. Because the state is reachable, there must be a state with an edge into it, unless the state is the starting state. If we fail on the starting state, the search as a whole has failed.

When all possible values for a given decision have been eliminated, we backjump to the most recent decision that is contained in an elimination for the current decision. We also resolve together the eliminations for the failed decision to derive a new elimination for the state to which we are backjumping.

The backjumping search strategy is essential to the solution of CIRCA planning problems. When the preemption decisions are taken into account, even modestly-sized CIRCA planning problems become too large for solution using simple chronological backtracking. However, backjumping is not sufficient for satisfactory search performance. With backjumping alone, the DAP planner often gets lost, repeatedly straying into states that cannot be solved (or are very difficult to solve). Accordingly, when we backtrack from a state, we tag it to indicate that it should be avoided. The other search heuristics, discussed below, are then biased to prefer actions and preemptions that avoid such states.

Note that we do not backtrack over state refinements. Backtracking over these refinements is never necessary: for every plan that can be found at a low level of detail, there is a corresponding plan at every higher level. Our experience suggests that the cost of "coarsening" an NFA (and the additional bookkeeping necessary to provide this option) is not worth the small savings in graph size.

Through additional backtracking, we provide a simple anytime behavior. CIRCA's AI System (AIS) caches plans as they are produced (recall that all plans are safety-preserving). Through backtracking, the AIS can generate plans that satisfy more of the goal propositions. Thus, once a first safety-producing plan is generated, the AIS may invest more time in generating better plans, or it may allocate its effort to other problems.

[6]TTFs must be preempted, so there is no choice point for TTFs.

5.2.5. Heuristic Control

There are three points in the DAP search where heuristic guidance is required: when we choose a state on which to operate, when we choose whether or not to preempt a temporal transition, and when we choose how to resolve the state. These three decisions are marked †, ‡ and ††, respectively, in Figure 12.

Choosing the Next State The choice of the next state to operate on († in Figure 12), is dictated by a simple heuristic. First, we prefer to examine an initial state, should any such state be available in the openlist. Second, we try to find a state that is "threatened" — i.e., the state is either known to be a failure state, or is a state at which we failed earlier and from which we have backtracked (cf. Section 5.2.4). Finally, we prefer a state that has an edge in from the most recently closed state. If all of these fail, then we just take the first state off the openlist.

What is the intent behind this heuristic choice? First, we prefer to expand initial states because, in the case that the initial state cannot be solved, the whole problem is unsolvable, and we would prefer to determine this as soon as possible. Furthermore, we prefer to search the NFA in a quasi-depth first way (we have more to say about this below).

The reason we prefer to choose the predecessor to a threatened state is a kind of "first-fail" or "most-constrained first" heuristic. When we are attempting to solve a state that has a known failure state as a successor, we must either make the failure state unreachable by the actions we take at the predecessor state, or fail and backtrack; hence this state preference acts as a surrogate for "first-fail." When we are using the heuristic for a state that has as a successor a state from which we have backtracked, the fact that there is a successor state that is threatened provides more guidance to the heuristic choices of actions and preemption decisions. Hence we can think of this as a kind of "most-constrained first" (or perhaps "most informed first") heuristic.

When the other heuristics fail, we prefer to choose a state that has an edge into it from the previously solved state. There are two reasons we prefer this:

1. we avoid squandering effort on irrelevant states; and
2. we fail sooner.

Preferring states that have edges into them from previously-solved states has the advantage of avoiding unnecessary computation. If we did not have this preference, we could examine states that appear reachable only because there is a chain of temporal transitions from an initial state, through unexamined states, to the state in question. Later search actions could render this state unreachable: the chain of intermediate transitions could be broken by preemption decisions. We also fail sooner with this heuristic choice, and backtrack more directly from unsolvable states (see Section 5.2.4).

Preemption Decisions Once a state is popped off the openlist, the first thing DAP does is make the preemption decisions (see the line marked ‡ in Figure 12). Here we use two heuristics: first, if a successor state is threatened (we have backtracked from it previously), then we prefer to preempt a temporal to that state. Second, all else being equal, we prefer *not* to preempt temporal transitions, because preemptions make action selection (and eventually, scheduling) more difficult.

Refinement and Action Decisions Finally, we use heuristic guidance in choosing the "resolution" of the state, either the decision to split on a particular proposition, or the decision to resolve the state by choosing a particular action. This is the choice point marked by †† in Figure 12. We treat this as a single choice point, rather than two, because that makes it easier for us to, for example, prefer an action when one is available.

The heuristic we use for directing the choice of actions and refinements is a modified version of McDermott's heuristic estimator for state-based ADL[7] planning [9]. To construct the heuristic, we construct a set of paths (an "operator-proposition graph") from the current state to the goal state, ignoring the interactions between individual actions. The heuristic function prefers those actions (or splits) that appear as first steps on one of these paths.

When choosing how to handle a state, the planner constructs an operator-proposition graph connecting the current state description to the goal state description. This is a layered graph, with alternating layers containing nodes that represent propositions to be achieved and operators that can establish those propositions. Despite using full lookahead, this approach is heuristic and efficient because it ignores details such as interactions between operators.

A sample heuristic graph is shown in Figure 13. In this example, the current state is $\{\neg P, \neg Q, T, U\}$ and the goal description is $\{P, Q\}$. The node marked `top` designates the goal state and has two descendents, for the two literals in the goal state proposition. In this domain, the planner has at its disposal three operators. The operator `establish-P` has as its preconditions R, S and its sole postcondition is P. `establish-Q`'s preconditions are $\{S, T\}$ and its postcondition is Q. In the operator-proposition graph we draw edges from unachieved propositions (here P and Q) labeled with actions that can achieve those propositions (`establish-P` and `establish-Q`). The base of the edge is labeled with a node with the conjunction of unrealized preconditions, here $\{R, S\}$ and $\{S\}$ (because T holds in the current state). Finally, we repeat the process, adding a new layer with nodes for individual literals (R and S). The S is marked specially: because the current state gives no value to the feature S, it is possible that S is true in the current state. R can be established by an operator whose sole precondition is U, a literal that is (necessarily) true in the current state. Once the graph-building is complete, we examine the graph for suggested search actions. In this case, the two actions suggested by the graph would be:

1. execute action `establish-R`, or

[7]ADL is an action notation that is related to STRIPS, but has slightly greater expressive power [13].

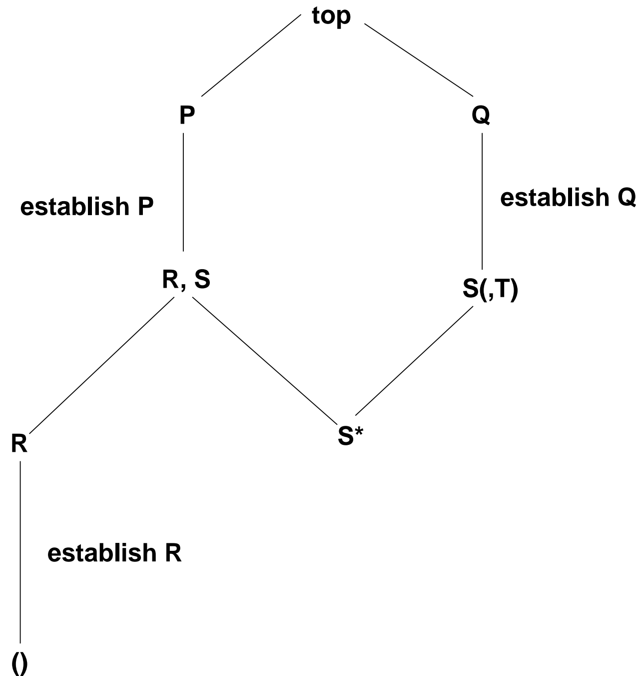


Figure 13. A sample operator-proposition graph.

2. split the state on proposition S.

This graph also illustrates what we mean when we say that the graph ignores interactions. The graph would look exactly the same even if the operator `establish-Q` had $\{\neg P, Q\}$ as its postconditions. The heuristic is meant as a simple, efficient filter on action selection, not a foolproof oracle. In this case, it would be up to the search engine to determine that `establish-Q` should be done before `establish-P`, rather than vice versa.

Our version of the operator-proposition graph differs from McDermott's because our actions are simple STRIPS operators; his approach covers schemas as well and must consider variable binding. Another difference is that McDermott's is a more traditional state-space planner, so state descriptions are complete and the only way to establish a proposition is to apply an operator with the appropriate postconditions. Our state descriptions are partial, and one way for the DAP planner to establish a proposition is to refine a partial state description to include that proposition. We showed this in our example, where S may be "achieved" by state refinement. There is no search operation like this in McDermott's planner.

Note that the operator-proposition graph may propose several choices: different actions and propositions on which to split. McDermott's heuristic evaluation function scores these alternatives according to the cost (roughly speaking, the length) of the path to the goal. Currently, we do not do this, but are actively investigating appropriate metrics for the way DAP uses the heuristic graph. In the prototype, we prefer action assignments over state splits, because state splits cause the problem size to grow. The choice of actions is filtered by feasibility, since we do not take into account temporal factors when composing the

operator-proposition graph, nor do we take into account which actions have already been attempted.

If the heuristic graph does not dictate what we should do, we attempt to choose an action or split in order to avoid a successor state from which we have previously backtracked (or that are known to be failure states). Here we use the kind of splits discussed earlier (see Section 5.2.1): splitting on the causation or preservation preconditions of transitions that lead us to the bad state. We also attempt to enable actions that can help us avoid bad states.

There is one additional step taken before we consult the operator-proposition graph. If there are any events that lead from the current state to a failure state, we attempt to split the state so that the events will be isolated, in the hopes of making them unreachable. Because events cannot be preempted, it is not worth considering action assignments or other splits in this special case.

To sum up, here is a table of heuristic choices:

- Choosing next state (†):
  1. Initial state.
  2. Predecessor to failure state.
  3. Predecessor to state from which we have previously backtracked.
  4. Successor of most recently closed state.
  5. First in openlist.
- Choosing whether to preempt a temporal (‡):
  1. If successor state is one from which we have previously backtracked, choose to preempt.
  2. Prefer *not* to preempt.
- Choosing action or split proposition (††):
  1. Check fail events, split (or fail) if any are applicable.
  2. Construct operator-proposition graph, suggesting actions and splits. Prefer actions if they are available.
  3. Attempt to refine state to avoid successors that are failures or from which we have backtracked.
  4. Attempt any remaining actions.

We have done no formal experimentation to determine which of these heuristics are helpful, nor how helpful they might be. Certainly the operator-proposition graph is very helpful. Without it, the DAP planner simply flailed about, only achieving goals when it blundered into them. We have tested it on the original CIRCA planner and found that it improved the performance of that planner as well. The original CIRCA planner simply tried all possible actions and used lookahead to choose between them. If the lookahead horizon was set too low, the original planner wouldn't find goal-achieving plans; if set too high, the planner would take a very long time to run.

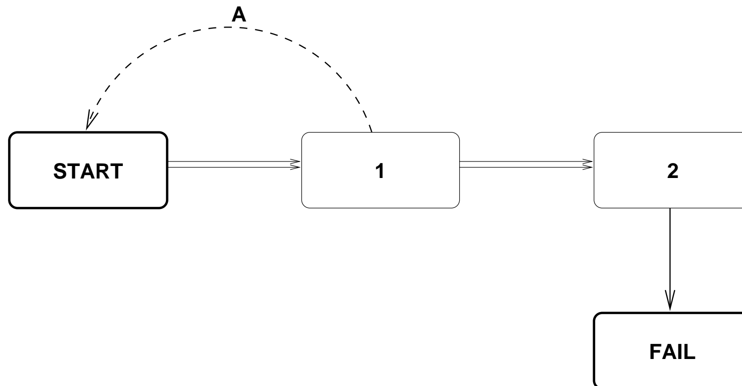


Figure 14. A case where poor choice of next state causes failure to find a plan.

5.2.6. Search Incompleteness

The current DAP search algorithm is incomplete. Incompleteness arises from limitations in the order in which search actions are performed. One possible source of incompleteness is the order in which states are chosen from the openlist; because we do not consider all possible states, some plans could be missed. A second source of incompleteness is the fact that we consider all preemption decisions before deciding on state refinements. Neither of these sources of incompleteness appears, at present, to be significant enough to be worth the cost of repairing it. More significantly, there are substantial sources of incompleteness in the temporal reasoning that is done by CIRCA. For more details on this source of incompleteness, see Appendix B.

One source of incompleteness is the way DAP pops states off the openlist. We do not systematically explore different orders in choosing a next possible state. It is theoretically possible that a bad choice of next state could make a search problem unsolvable. Figure 14 illustrates a domain in which state choice order could make the search fail. In this example, if we were to pop first the start state and then state 2, we would fail to find a solution. We would be unable to solve state 2 because it has an event to failure, and we would not be able to backtrack from state 2, because we would not have state 1 on the stack. On the other hand, were we to remove state 1 from the stack first, we could choose action `A`, preempt the transition to state 2 and find a successful plan. This problem does not arise in practice in the domains we have worked with; our state choice heuristic avoids it. Since exploring states in different orders would significantly increase the search space and since this problem has not come up in practice, we have accepted the incompleteness.

A second source of incompleteness is the order in which we make preemption decisions. It is possible that search to a state could fail in the phase of making preemption decisions. At this point we might fail to find an action because the preemption decisions are made before deciding on state refinements. So it is possible that a state that looked unsolvable to our implementation would be solvable if it could be refined. We could repair this incompleteness by permitting state refinements earlier in the search process. However, we have not done this yet because we are not certain that this incompleteness actually arises

in the examples of interest to us.

6. Evaluation

Dynamic abstraction provides the greatest benefits in domains where uncertainty is present and can be reasoned about or managed in abstract ways, rather than requiring fully-detailed reasoning at all points in the state-space. In the CIRCA planning paradigm, there are several possible sources of uncertainty that can be advantageous for DAP:

Events and Temporals — These nonvolitional transitions are outside of the system's control, and therefore lead to uncertainty in the planned trajectory through the state-space, as they can move the system off the direct planned path. The original planner must consider each state that results from an event or temporal and the resulting sequences, possibly not overlapping with the direct plan, that are required to move towards the goal.

Initial conditions — The system can be told it may begin in one of several possible initial conditions. This uncertainty requires the original planner to consider explicit paths leading from each initial state, while DAP may be able to ignore the differences between initial states and find a single plan.

Nondeterministic actions — Because actions can have nondeterministic outcomes, they can cause branching in the original planner that DAP may be able to avoid.

There are also some factors that can partially negate the advantages of DAP. In particular:

Goal propositions — Our heuristic DAP planner begins by splitting its maximally-abstract state space on all of the goal propositions. So if there a large number of goal propositions, or if features mentioned in goal propositions have large feature spaces, DAP may split out the space into a large number of distinguished states that are, in fact, unnecessary. In such domains, the original planner can actually outperform DAP because it enumerates only reachable states, rather than all of the possible values for all features currently included in the state description (as does DAP).

Dynamic Abstraction Planning is not simply applicable to CIRCA planning. The DAP technique could bring automated abstraction to other planners with different state representation, transition semantics, and temporal models. Therefore, we have evaluated the benefits of DAP independent of many of the CIRCA-specific details of the planning model. In particular, to evaluate DAP independent of the complex CIRCA temporal model, we avoid using temporal transitions. In turn, this means that the issue of preemption does not arise in the evaluation problems discussed here.

Our evaluation consisted of running both the DAP and Classic CIRCA state-space planners on numerous domains that were automatically generated to meet several sets of defining characteristics. Each of these sets of domains (or "domain classes") highlights

particular ways in which DAP differs from, and usually improves upon, Classic CIRCA and other state-space planners.

Note: for the sake of simplicity, we have restricted ourselves to boolean features in the test problems. In a slight abuse of terminology, but for the sake of conciseness, we will refer to the proposition `(P T)` as "the proposition P" and a transition with `(P T)` in its postconditions as "establishing P." A transition with `(P F)` in its postconditions will be said to "delete P."

6.1. Eval-1 Domain: Benign Events

The first evaluation domain, Eval-1, shows DAP's ability to ignore irrelevant nonvolitional transitions. By ignoring these irrelevant transitions, and the features they affect, DAP avoids an exponential state-space explosion that plagued the original state-space planner.

One type of irrelevant transition we will focus on is a *benign event*: an event establishing a proposition that does not appear in any preconditions or postconditions of actions on the "causal chain" to the goal (we define this term below). The Eval-1 domains show how DAP can build small, abstract plans that accurately characterize much larger state spaces connected by benign events. On the other hand, the original planner must enumerate the entire exponential state spaces.

Throughout our experiments, we consider domains in which there is a *causal chain* from the initial state(s) to the goal. To automatically generate one of these domains, we build a set of simple actions that must be chained together to achieve the final goal. We only declare the final proposition in the chain as a `*goal*` proposition to CIRCA, so that DAP only splits on that single goal proposition initially. Using a notation derived from that used by Barrett and Weld [1], we can describe these actions by the template:

(make-instance 'action :name Achieve-Goal-i
:preconds ((G_i F) (G_{i-1} T))
:postconds ((G_i T)))

In other words, the set of actions implicitly creates a sequence of goal features, $G_1, G_2, \ldots G_n$. The action achieving G_k has, as its precondition, that the prior goal in the sequence (G_{k-1}) has already been achieved. For our purposes, this has the desired effect of requiring a final plan with several actions, but only one input goal for CIRCA (the last goal, G_n).

To introduce benign events, we create an additional set of "external" or non-goal features ($\{P_1, P_2, \ldots P_m\}$) which are irrelevant to the causal chain. The values of these external features may change over time due to events. We define a number of events that establish these propositions. Note that these events do not interact with the causal chain, for good or ill:

(make-instance 'event :name Add-i
:preconds ((P_i F))
:postconds ((P_i T)))

```
(make-instance 'action :name "Achieve-G1"
          :preconds '((G1 F))
          :postconds '((G1 T)))
(make-instance 'action :name "Achieve-G2"
          :preconds '((G2 F) (G1 T))
          :postconds '((G2 T)))
(make-instance 'action :name "Achieve-G3"
          :preconds '((G3 F) (G2 T))
          :postconds '((G3 T)))
(make-instance 'event :name "Add-P1"
          :preconds '((P1 F))
          :postconds '((P1 T)))
(make-instance 'event :name "Add-P2"
          :preconds '((P2 F))
          :postconds '((P2 T)))
(make-instance 'event :name "Add-P3"
          :preconds '((P3 F))
          :postconds '((P3 T)))
(setf *goals* '((G3 T)))
(setf *initial-states*
      (list (make-instance 'state
                  :features '((P1 F) (P2 F) (P3 F) (G1 F) (G2 F) (G3 F)))))
```

Figure 15. The 3-goal, 3-event Eval-1 domain.

Figure 16 and Figure 17 illustrate the results for the Rand1 domain listed in Figure 15, in which $n = 3$ (the causal chain is of length three) and $m = 3$ (there are three external features and, hence, three establisher events).

Note that DAP has even managed to omit seemingly important information about the initial state (namely, `(G2 F)`). The planner was given this information initially but chooses to ignore it. This leads to apparent nondeterminism in the outcome of a planned action, the one chosen for the initial state. In fact, that action is deterministic (see Figure 15), but ignoring that fact does not keep DAP from achieving the goal.

As might be expected, the performance of the planners on this highly-structured domain can be predicted analytically. As the number of benign events increases, the DAP planner's final plan size does not change at all. All the events are sufficiently modeled by the simple self-loops shown in Figure 16. On the other hand, the original CIRCA planner's state space grows exponentially. Each benign (and independent) event in the domain forces the original planner to replicate the entire path from initial state to goal (which is $n + 1$ states long). For each of the 2^m combinations of values of the external features, there is one

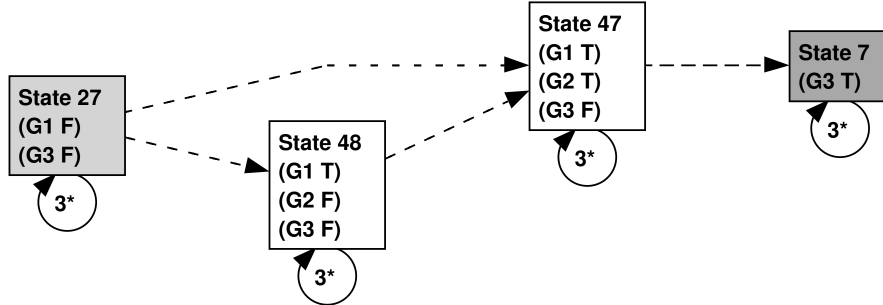


Figure 16. DAP plan for the Eval-1 domain with 3 goals and 3 benign events: no state-space explosion. The 3^* notation indicates that the self-loops are triply-replicated, corresponding to the events affecting the three external events not included in each abstract state. In all diagrams, initial states are lightly shaded, goal states are darker.

such path. So the size of the original planner's state space is $(n+1) * 2^m$. We have confirmed this relationship experimentally.

Figure 18 illustrates how the DAP planner's state-space is constant over benign events and scales linearly as the number of goals increase. In contrast, the original CIRCA planner is exponential in the number of benign events, and linear in the number of goals (with a slope determined by the exponential benign event factor).

Figure 19 shows that the runtime for the old planner grows exponentially as the number of benign events (and hence states) grows. As expected, DAP remains linear in runtime. It is useful to note that DAP's runtime is not constant as the number of benign events grows (unlike the DAP state space). By plotting DAP's runtime alone we can zoom the scale in and see, in Figure 20, that the DAP runtime is linear in both the number of goals and benign events. This occurs because, although additional benign events do not lead to new states, they must each be considered and understood to be self-loops in every state of the plan.[8]

6.2. Eval-2 Domain: Uncertainty in Initial Conditions

This domain class shows the advantages of DAP when there is uncertainty in the initial conditions of the planning problem. The domain examples are generated using the same goal structure as in Eval-1, where a chain of intermediate goal-achieving actions leads to a single final goal predicate. Unlike Eval-1, there are no external events at all; instead, the

[8]Minor runtime variations away from linearity are simply due to the unpredictable Unix timesharing environment, and appear here because we are plotting individual runs, not averaged performance over many runs.

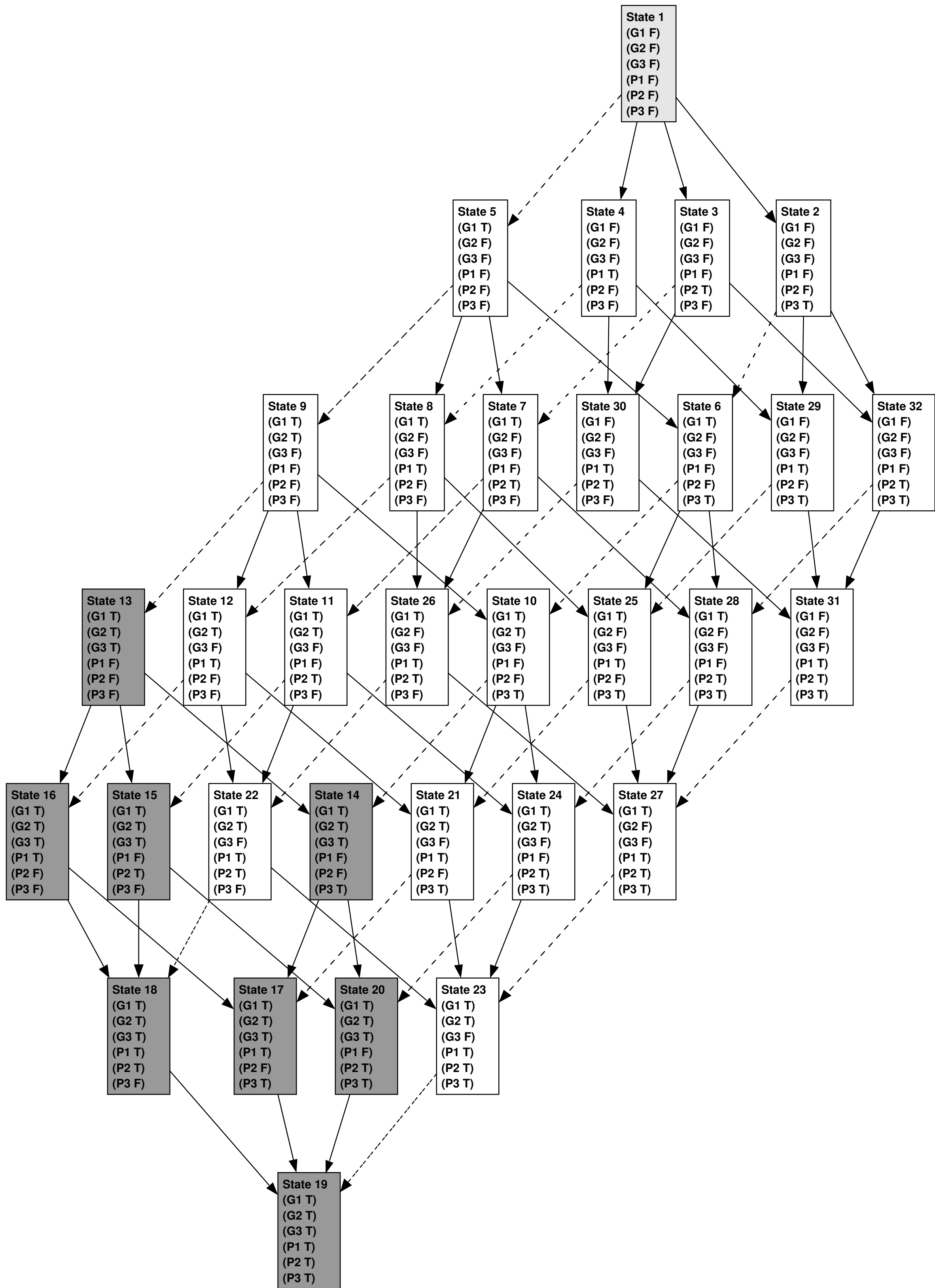


Figure 17. Classic CIRCA plan for the Eval-1 domain: all combinations of events.

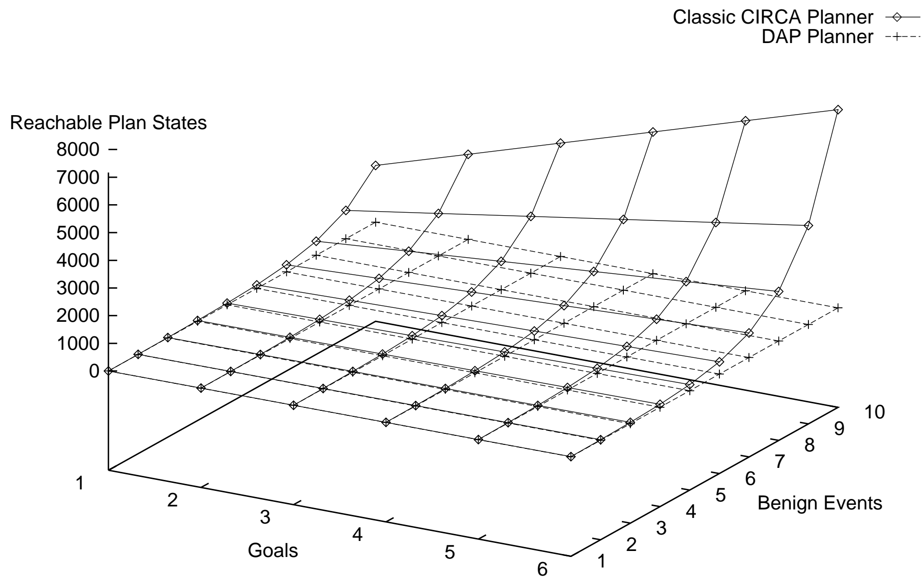


Figure 18. DAP avoids state-space explosion on Eval-1 domains.

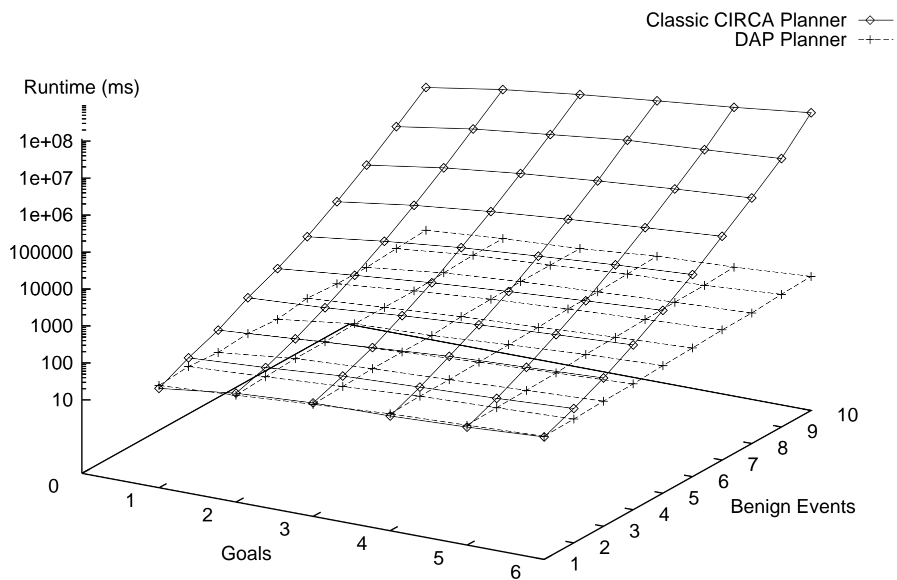


Figure 19. Classic CIRCA's runtime is exponential in the number of benign events. Note the logarithmic runtime scale.

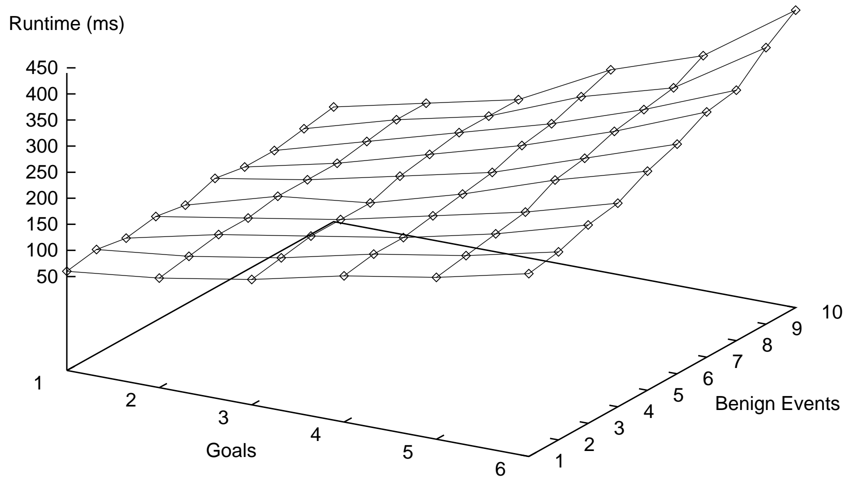


Figure 20. DAP's runtime on Eval-1 domains is linear in the number of goals and benign events.

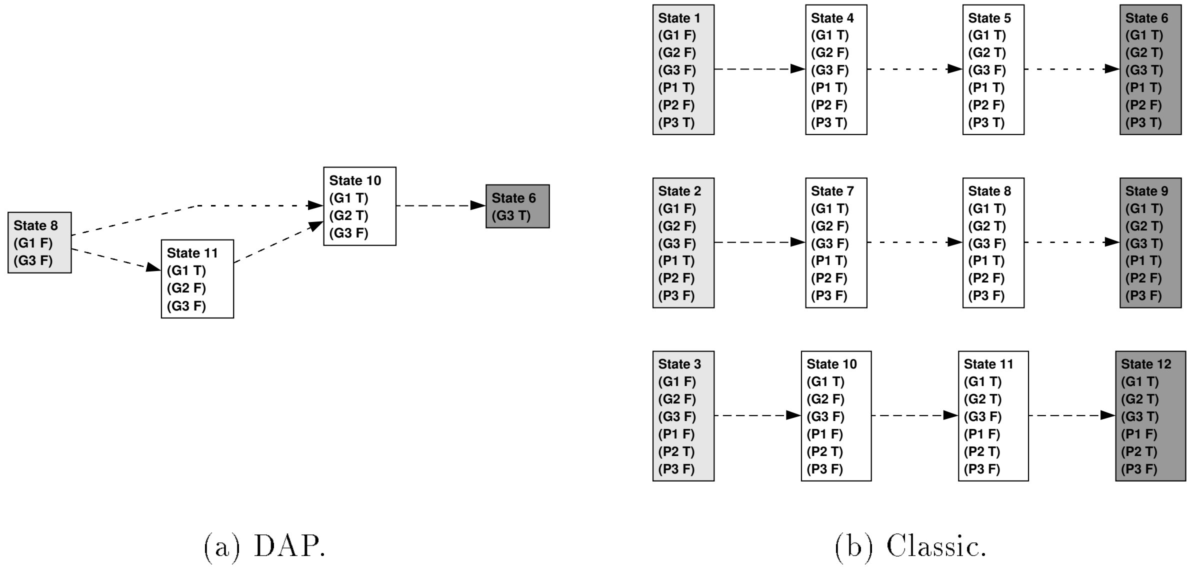
only uncertainty arises in the initial conditions. Multiple initial states are created by adding external features and randomly choosing value assignments to them. The randomness serves only to build different initial conditions. It is not important that they be smoothly distributed throughout the space, since the names of the features involved, and their values, have no effect on goal achievement.

On Eval-2 domains, DAP completely ignores the differences between the declared initial conditions, building only a single abstract start state. This yields substantial savings in the number of states enumerated and in the planner's runtime. For example, Figure 21a shows the DAP plan for a Eval-2 domain declared with three goal predicates and three initial states. The graph shows only one initial state because DAP never splits the state space on any of the predicates that differentiate the declared initial states. In contrast, Figure 21b shows the Classic planner's output, in which each of the initial states leads to a different path through the state-space.

As this example implies, for Eval-2 domains DAP's plan size is constant with respect to the number of initial conditions, while the Classic plans grow linearly (see Figure 22).

6.3. Eval-3 Domain: Required Events

The Eval-3 domain class investigates planner performance when the planner must consider events as critical elements of the path to the goal. To force the planner to rely on events in



(a) DAP. (b) Classic.

Figure 21. Plans for Eval-2 domain with 3 goals and 3 initial states.

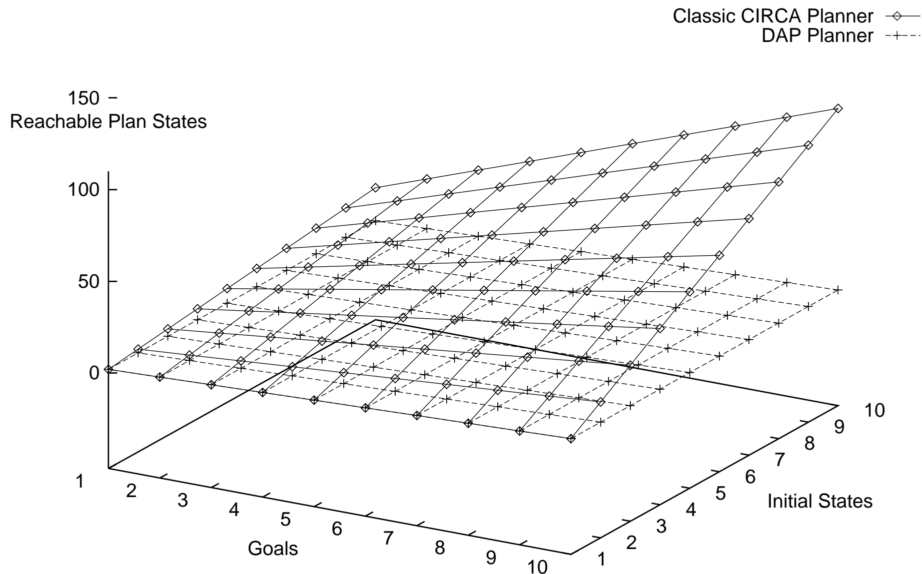


Figure 22. Plan size for Eval-2 domains with uncertainty in initial conditions.

the planned path, we made each goal-achieving action include a single external predicate as a precondition. These external propositions could only be established by events; there are no suitable actions.

The state of these external features determines the planner's choice of action. We replicated the actions so that, for each goal in the causal chain, there was one establishing action enabled by each external proposition:

$\forall i, \forall j$ (make-instance 'action :name Achieve-Goal-i
:preconds ((G_i F) (G_{i-1} T) (P_j T))
:postconds ((G_i T)))

These actions are as before (cf. p. 30), but with the additional precondition (P_j T).

Complicating matters further, we specified new events that can make the external predicates false, or "delete" them, in addition to the original "adding" events from the Eval-1 domain.

(make-instance 'event :name Delete-i
:preconds ((P_i T))
:postconds ((P_i F)))

We expect DAP to outperform Classic in this domain because it can avoid thinking about all the combinations of external predicate values. Still, the domain class is more complex than Eval-1 because at least one external predicate (and event) *must* be relevant and considered to solve the problem. In fact, at any point in the state space, the planner needs to have exactly one external predicate true to be able to achieve the entire sequence of goals. Ideally, DAP would perform a single split on one external predicate and then rely on that predicate and the correspondingly-enabled actions for the rest of the plan. This would make DAP's plans only slightly larger than for the Eval-1 domains.

Unfortunately, the current heuristic that chooses which splits to perform is not quite smart enough to take advantage of this structuring, and it occasionally chooses to split on a different external predicate. Figure 23 shows that, for a three-goal Eval-3 domain, DAP has chosen to first split on the external predicate `P3`. Since actions exist to take states with `(P3 T)` all the way to the goal, that should have been enough detail, but the heuristic does not realize this and splits on `P2` later on.

Even with this sub-optimal heuristic, DAP significantly outperforms the Classic planner. Unlike the Classic planner, DAP does not enumerate the effects of all of the irrelevant events. Figure 24 shows that the Classic plans blow up exponentially, as with Eval-1, while DAP's grow at a much smaller, but apparently still exponential, rate. The runtime for both planners is exponential, as shown in Figure 25.

Note that the current heuristic code, which arbitrarily orders the propositions chosen for splits, often does as poorly as possible for DAP. So the exponential runtime here is very much a worst case. We are now working to improve the heuristic behavior, as discussed in Section 7.

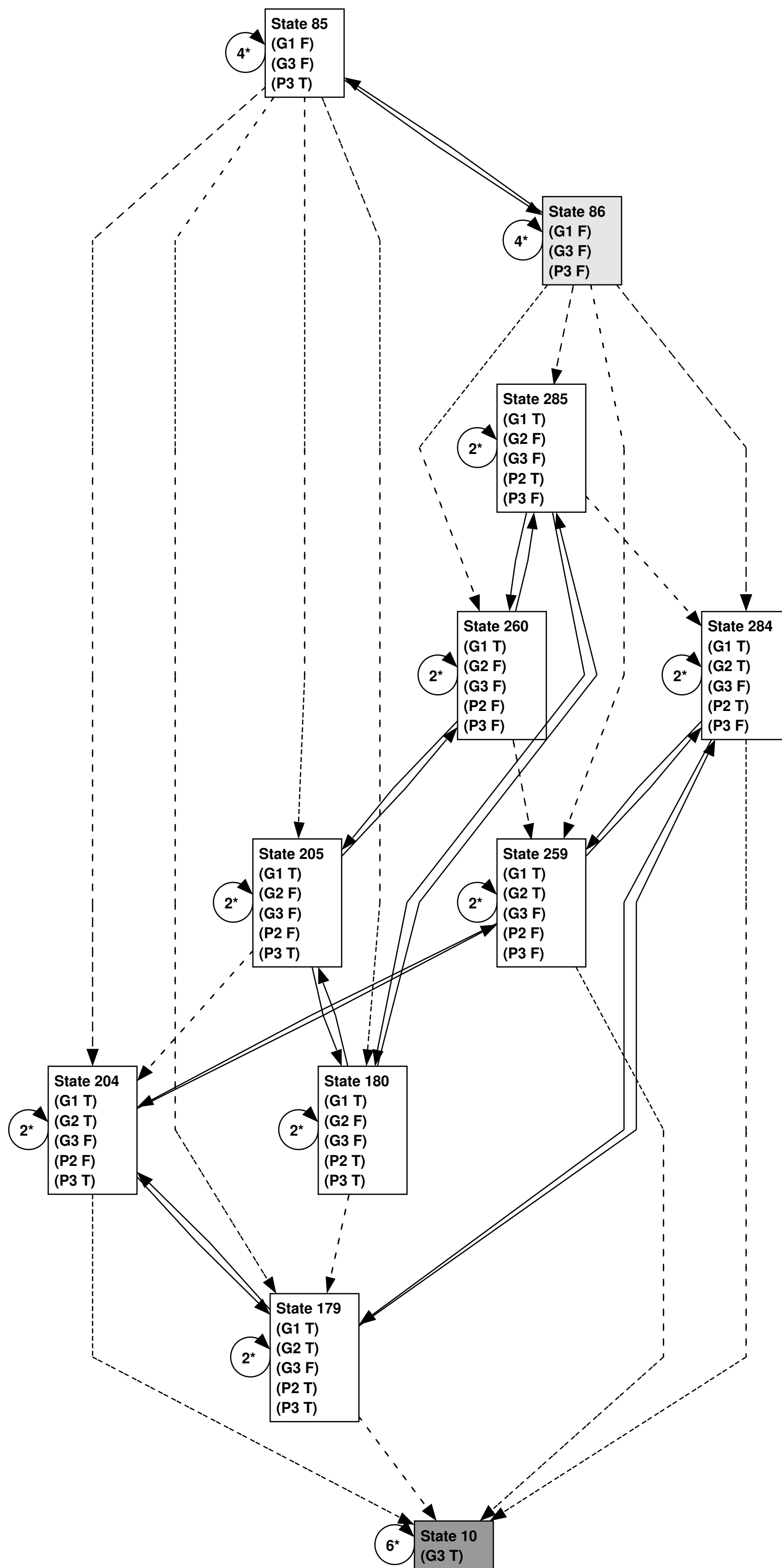


Figure 23. DAP solves a 3-goal Eval-3 domain with sub-optimal split choices.

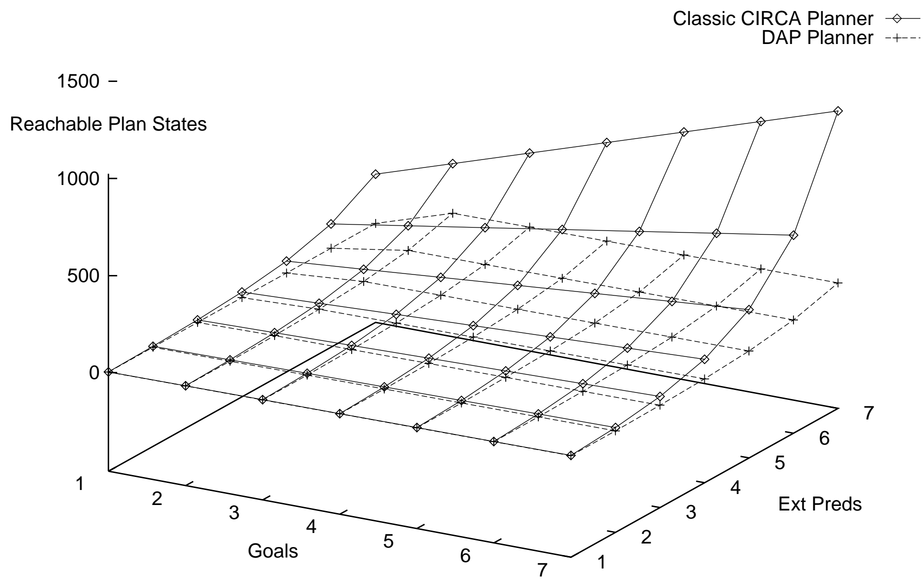


Figure 24. Plan size for Eval-3 domains with required events.

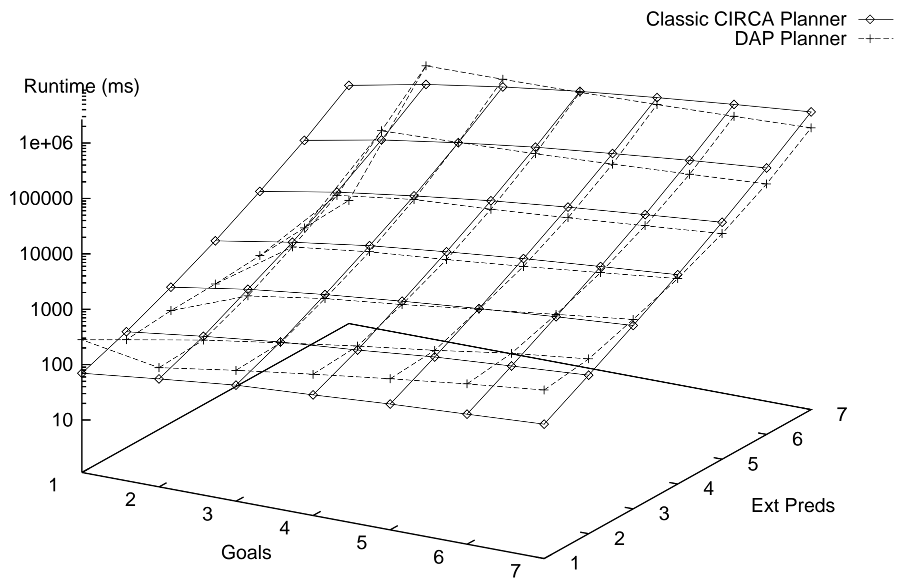


Figure 25. Runtime for Eval-3 domains with required events.

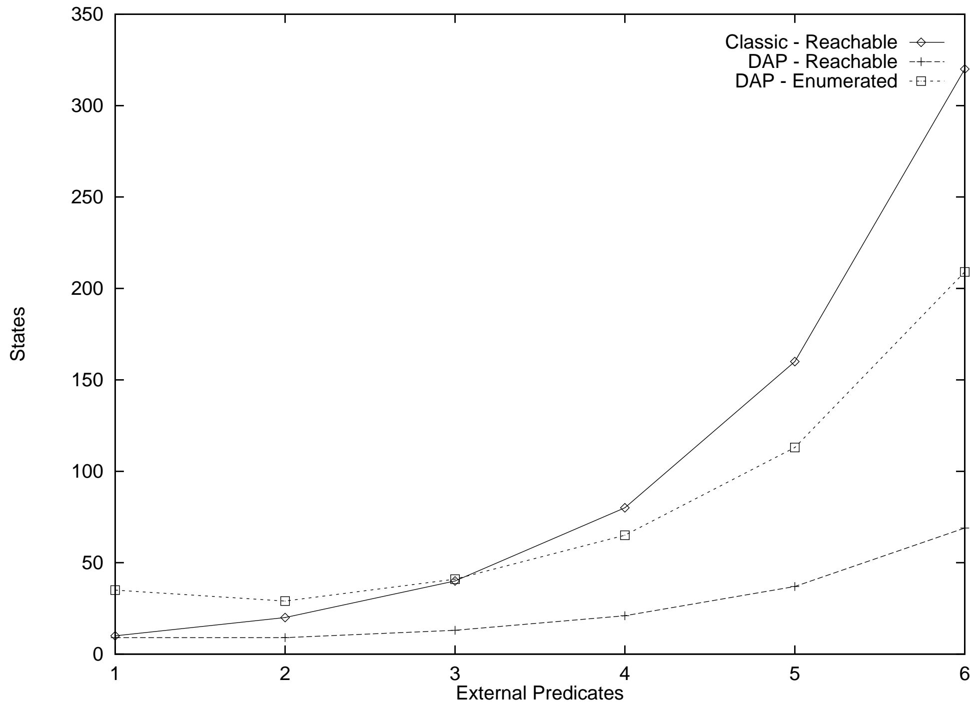


Figure 26. Detailed state space performance for a 4-goal Eval-3 domain.

Taking a slice of Figure 24 at four goals, Figure 26 shows more clearly how DAP's state space is growing exponentially, but more slowly than Classic's. We have also included a new metric of DAP performance, the total number of enumerated states, to show the exponential work DAP is doing. The enumerated states metric includes those states that are split and discarded during the DAP search process. For Classic, the number of enumerated states is the same as the number of reachable states for all domains without preemption or backtracking (which includes all of the domains discussed here).

There is an interesting crossover point in Figure 26 showing that, for less than 3 external predicates, DAP can actually enumerate more states than Classic. This case occurs when DAP's splitting process, which builds new more-detailed states and discards old overly-abstract ones, actually builds more states than are reachable in the fully detailed domain (which is enumerated by the Classic planner). Section 6.6 investigates this type of anomaly in more detail. Note, however, that the number of reachable states in the final DAP plan is always smaller than the Classic plan, for these domains.

6.4. Eval-4 Domain: Complex Event Interleaving

By modifying the Eval-3 domain class slightly, we produced a new class designed to highlight DAP's ability to abstract the state space in a non-homogeneous fashion. That is, DAP can include a feature in some parts of the space, and ignore it in others. Eval-4 domains consist of a set of goal-achieving actions that each require a different external predicate:

```
∀i (make-instance 'action :name Achieve-Goal-i
                          :preconds ( (G_i F) (G_{i-1} T) (P_i T) )
                          :postconds ( (G_i T) ))
```

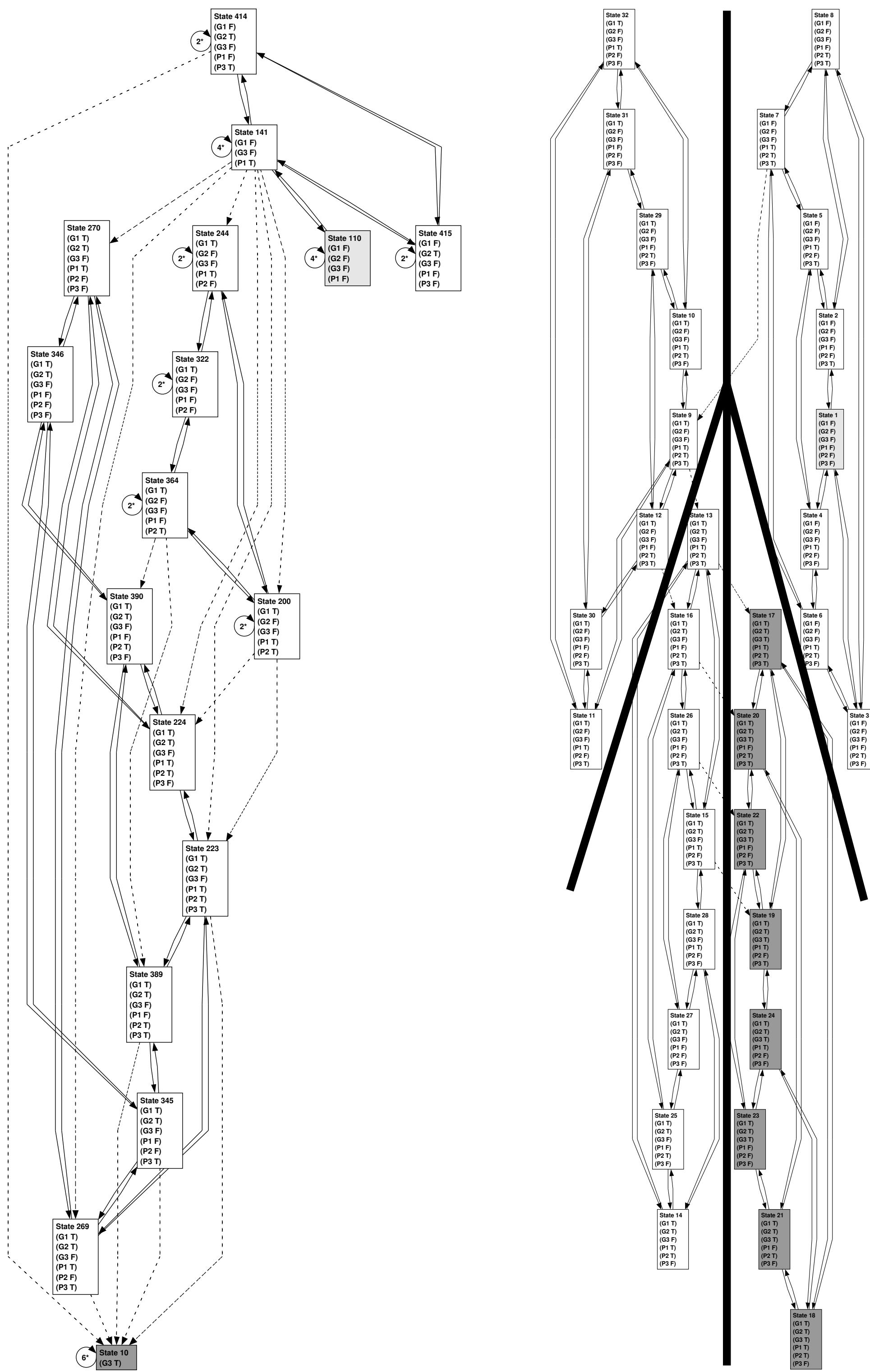
Thus an n-goal Eval-4 domain also has n events in the success path, and at some point even the DAP planner will have to consider all n external predicates. As illustrated by the example in Figure 27a, however, DAP can limit the propagation of those external predicates so that it still does not consider the exponential set of their combinations. The Classic CIRCA planner must enumerate all combinations, and builds plans like that shown in Figure 27b. The overall performance results are shown in Figure 28.

An interesting feature of the plans built by Classic points out another characteristic of the current search heuristic: the heuristic does not consider events in its projection of possible paths. As a result, the Classic planner will not choose any action until all of the external predicates have been made true by events. This is reflected in Figure 27b by the vertical banding structure highlighted with bold separating lines. Each of the four tightly-connected bands of states corresponds to the full set of external predicate combination for each of the possible states of the three goals. For example, the upper right band corresponds to all the values of `P1`, `P2`, and `P3` with none of the goals true. Only once State 7 is reached, and all the external predicates are true, does the heuristic recognize a path of actions to the goal. We will see this behavior reflected even more clearly in the next domain, and discuss more implications below in Section 6.5.2.

6.5. Eval-5 Domain: Eval-4 with No Deleter Events

The Eval-5 class of domains was formed by simply removing the "deleter" events from Eval-4. These events, which complicate matters by deleting the external predicates that are required for action preconditions, forced Classic to consider all combinations of the external predicates. By eliminating the deleters for those predicates, Classic can just reason about the states that result as they are made true by events, and they will never thence become false. This, combined with the aforementioned heuristic preference, leads Classic to build plans with the distinctive structure illustrated in Figure 29b. Here, in the four-goal Eval-5 domain, the Classic planner only chooses to act once all of the external predicates have been made true by events. DAP, on the other hand, has built a more complex plan interleaving selected actions with expected events, as shown in Figure 29a.

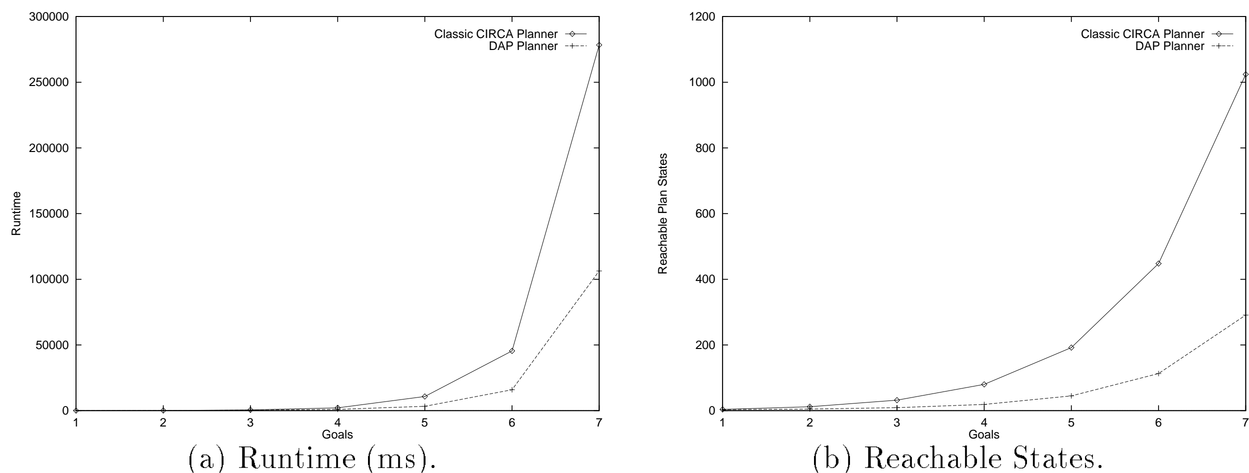
Because DAP can end up splitting on several predicates in these domains, it can enumerate many of the combinations of those predicate values (albeit in a "local" fashion, since splits are not global across the state space, as the abstract representation is heterogeneous). Since all of the external predicates must be specified at some point to find the goal-achieving path, DAP is forced to split on each one in at least some areas of the search space. In this challenging environment, the DAP overhead (that results from starting with abstract states and progressively refining them) overwhelms the advantages it has over the Classic planner. Figure 30 shows the state-space and runtime performance metrics indicating that Classic is significantly better in these Eval-5 domains.



(a) DAP.

(b) Classic.

Figure 27. Plans for Eval-4 domain with 3 goals.



(a) Runtime (ms). (b) Reachable States.

Figure 28. Eval-4 domain shows DAP using non-homogeneous abstraction to advantage.

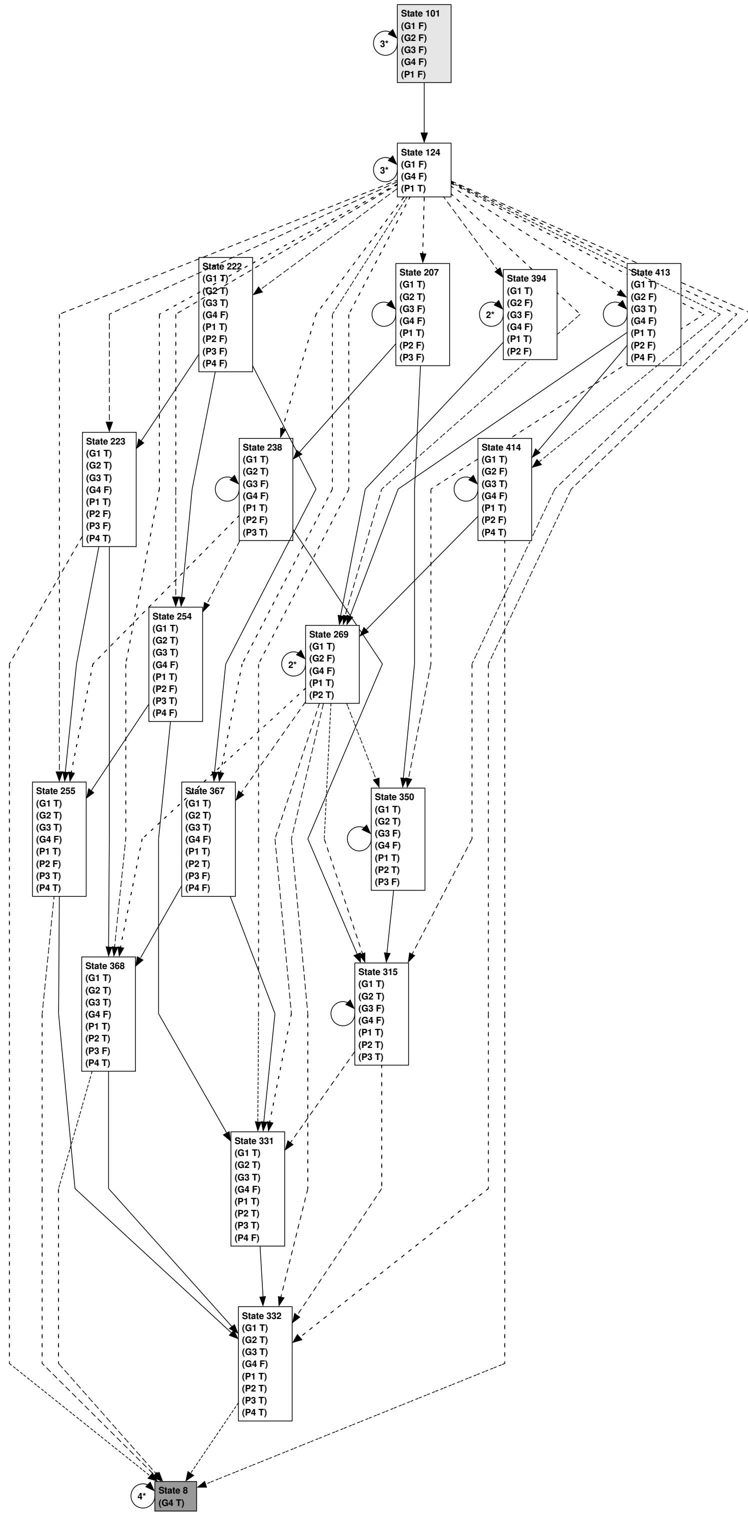
6.5.1. Implications of Heuristic Behavior on the DAP Planner

The DAP planner actually plans *less* efficiently for this set of scenarios. This inefficiency arises because DAP considers state refinement on a feature-by-feature basis. In turn, this causes it to add states to its NFA that are not, in fact, reachable. This can be seen in Figure 29a. Consider State 207 in this illustration. The feature set for this state is `((G1 T)(G2 T)(G3 F)(G4 F)(P1 T)(P2 F)(P3 F))`. The DAP planning algorithm considers this state to be (possibly) reachable from the initial state by means of an event that establishes `P1` and an action that establishes `G1`. However, if one considers the underlying domain description, one can see that this state is *not*, in fact, reachable. First, in this state `G2` is true. But `G2` is false in the initial state and there is no transition on a path from the initial state (101) to State 207 that establishes `G2`. So if we actually consider execution paths, we see that State 207 is unreachable.

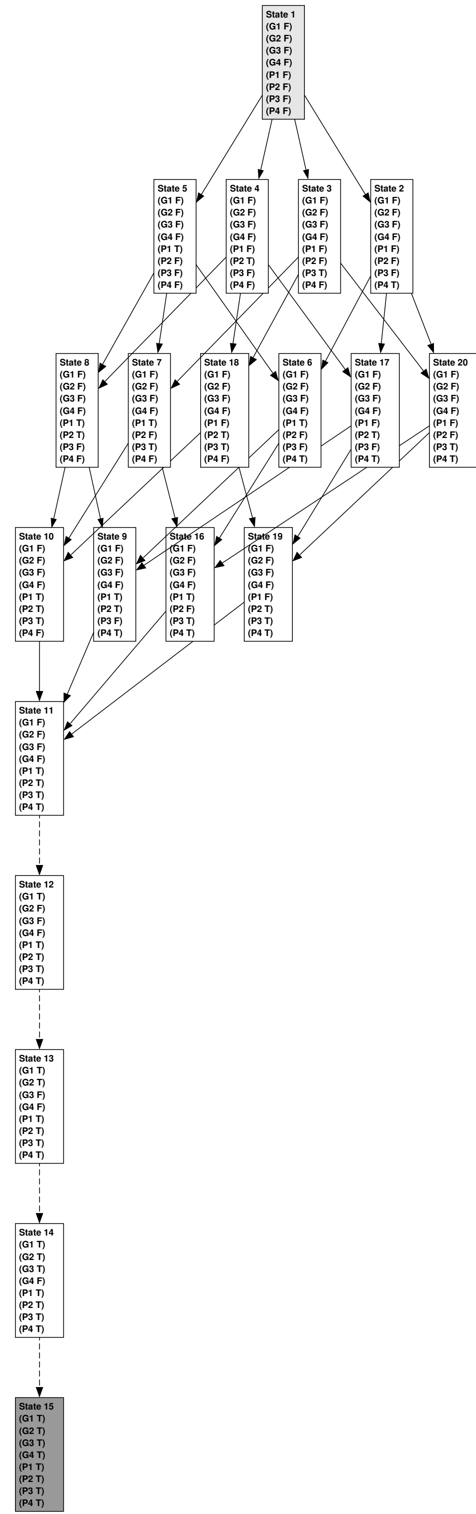
However, the DAP planner never simulates the execution of plans from the initial state, instead operating in an abstract space alone. In the abstracted representation of this plan, State 207 appears reachable. Further, the DAP planner has no reason to look more closely: it has successfully determined what to do in the event that CIRCA is in a state consistent with State 207. In that event, it should simply wait for an event to carry it to somewhere from which it can reach the goal state, a state in which either `P3` is true or `P2` is true (State 258 or 350).

DAP CIRCA would behave differently if it were unable to identify a course of action to take in this state. In that case, it would further refine the plan in order to determine that the state was unreachable.

In most of the cases we have examined, the DAP algorithm behaves acceptably. As we have pointed out, all that happens is that the planner considers contingencies that do not arise. In our experience, there are not enough of these contingencies to make the planner

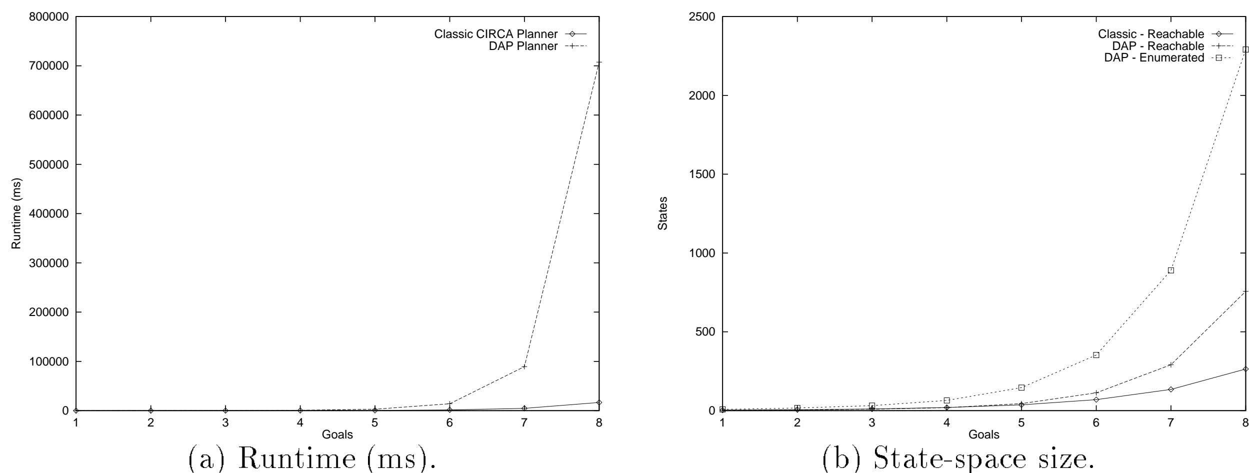


(a) DAP.



(b) Classic.

Figure 29. Plans for Eval-5 domain with 4 goals and no external predicate deleter events.



(a) Runtime (ms).
(b) State-space size.

Figure 30. Eval-5 domains with no deleter events shows Classic CIRCA outperforming DAP planner.

behave badly in general. However, in this domain the planner creates a large set of unreachable states, leading to bad performance.

6.5.2. Implications of Heuristic Behavior on the Classic Planner

The distinction between the plans of the two CIRCA planners on Eval-4 and Eval-5 domains is actually an artifact of the different ways they interact with the heuristic graph algorithm. In the case of the Classic CIRCA planning algorithm, the heuristic graph "tells" CIRCA that it is pointless to plan any actions until a sequence of events have carried the automaton to a favorable state from which to begin acting. On the other hand, the DAP planner, because it abstracts and refines on a feature-by-feature basis, can introduce unnecessary states into its NFA.

As we saw in the Eval-4 and Eval-5 domains, the current search heuristic does not take into account the possibility of using temporals or events to achieve goals. So, in the initial state of a Eval-4 domain, when it tries to find a path from the current state to the goal state, it fails. In the Classic planner, where splits are not an option, the heuristic is saying "it's hopeless here; you might as well just do nothing." Fortunately, the appropriate events are then applied and the planner, in a depth-first fashion, finds the later states in which actions are applicable and can lead directly to the goal.

Note that this says something interesting about prepositioning problems — it introduces an asymmetry into how Classic CIRCA treats such problems. On the one hand, Classic CIRCA is quite smart about prepositioning assets in order to avoid bad outcomes (i.e., making failure unreachable). However, with the present heuristic, Classic CIRCA is not smart about prepositioning itself to *achieve* goals by fortuitous events or temporal transitions. Since the heuristic graph doesn't include nonvolitional transitions, it is unable

```
(make-instance 'action :name "Achieve-P1"
          :preconds '((P1 F))
          :postconds '((P1 T)))
(make-instance 'event :name "Achieve-G1"
          :preconds '((G1 F) (P1 T))
          :postconds '((G1 T)))
(setf *goals* '((G1 T)))
(setf *initial-states*
      (list (make-instance 'state
                  :features '((P1 F) (G1 F)))))
```

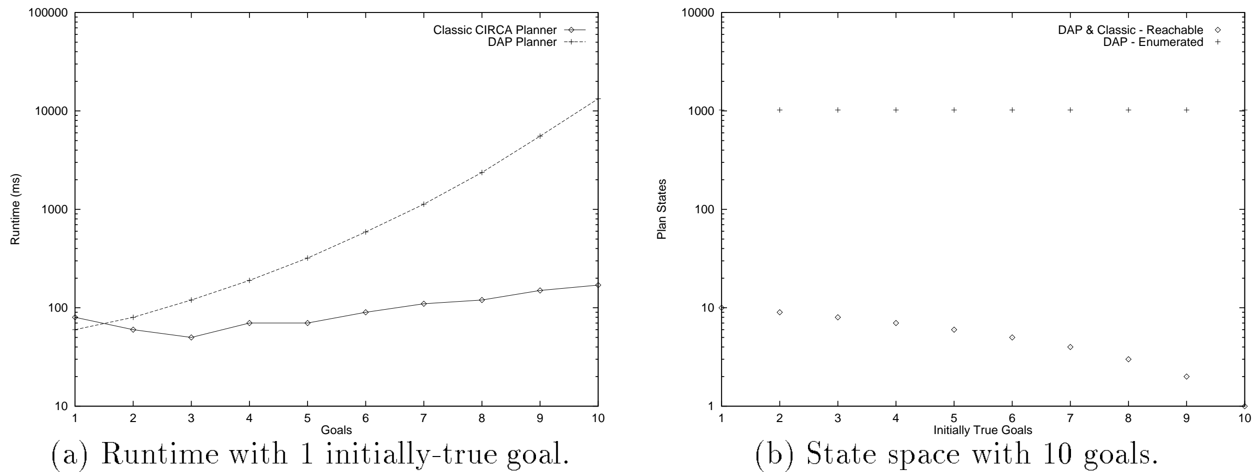
Figure 31. A simple domain illustrating the difficulties of prepositioning for fortuitous events.

to see that there is a point to doing something so that a later event will take it to the goal state.

For example, consider the domain shown in Figure 31. Here, the planner must decide to take the `Achieve-P1` action *before* the `Achieve-G1` event in order to reach the goal `(G1 T)`. On first encountering the initial state, Classic CIRCA will not notice that it is worthwhile to do the `Achieve-P1` action. In fact, it builds a safe but sub-optimal plan that simply loops in the initial state, thinking that the goal is not reachable. Then, because both the DAP and Classic planners run within an outer loop that attempts to find a safe plan that achieves all of the goals, the Classic planner will backtrack to search for another, more goal-achieving plan. Once the planner backtracks, it chooses action `Achieve-P1` as the only remaining alternative for the initial state, and then successfully realizes that the goal is now reachable via the `Achieve-G1` event. Essentially, the heuristic was not useful in finding this goal-achieving plan, and the planner "stumbles" onto the goal using brute-force backtracking search.

6.6. Eval-6 Domain: Expensive Splitting on Goals

As noted earlier, the DAP algorithm begins by splitting the state space on all the declared `*goals*`. In situations where this splitting is unnecessary because some goal predicate values are never reachable, the Classic planner can outperform DAP because it will only consider reachable states. To demonstrate this behavior, the Eval-6 domain class is formed using only the goal-achieving actions from Eval-1 and then declaring all the goal predicates in `*goals*`. For an n-goal Eval-6 domain, DAP splits on the goals to form at least 2^n states. In contrast, the Classic planner's state space is not directly affected by the `*goals*` declaration, and it creates only reachable states. If, for example, all the goals just happen to be true in the initial state, DAP will still generate 2^n states while Classic will generate



(a) Runtime with 1 initially-true goal. (b) State space with 10 goals.

Figure 32. Eval-6 domains with many `*goals*` shows Classic outperforming DAP.

exactly one state. Eval-6 is the most extreme case where DAP's initial splitting on goal propositions is harmful: all the goals are declared in `*goals*` *and* they are all true in the initial state, so that they could actually be ignored in domains in which goals cannot be undone.

This distinction may help clarify the difference between the total number of states enumerated by the planner and the number of reachable states in the final plan. When n goals are declared, DAP will enumerate at least 2^n states, but only a few of them may turn out to be reachable. In simple domains without failures and backtracking, the Classic planner enumerates only reachable states.

7. Future Directions

We have identified a number of interesting directions for future work on the CIRCA architecture. On the theoretical level, we are interested in improving the expressiveness of CIRCA problem descriptions, in order to expand the set of problems CIRCA can tackle. We have also considered allowing more flexible state refinement in the DAP planning algorithm; this may be useful in distributed planning problems and in problems where there are features with very large state spaces. In the experimental evaluation work, we have identified cases that mislead the current heuristic function, causing the DAP planner to perform badly. We have identified solutions to this problem and plan to improve the heuristic accordingly. At the same time we are exploring applications of the CIRCA architecture for mission-critical planning: autonomous spacecraft for NASA deep space missions [12]; Unmanned Air Vehicles (UAVs) for both ground attack and reconnaissance applications; and mobile robots for military Small Unit Operations.

7.1. Extending CIRCA's Expressiveness

At the same time that we have been working to increase the efficiency of CIRCA's planning, we are working to relax limits on its expressiveness. In doing this we have been driven by consideration of the scenario outlined by Gat in his paper "News From the Trenches: An Overview of Unmanned Spacecraft for AI" [4].

In this paper, Gat presented a planning scenario from the Cassini mission that he argued no current AI planning system could tackle. The problem concerns the Saturn orbital insertion of the Cassini spacecraft. In order to successfully navigate, the Cassini spacecraft must have an inertial reference unit (IRU) powered up and functioning. The spacecraft has a primary and a secondary IRU. The problem is to foresee the possibility of a primary IRU failure and warm up both IRUs early enough that they will be available for navigation at the time of orbital insertion.

CIRCA is quite capable of planning to warm up both IRUs, provided that it is informed that doing orbital insertion without guidance is a failure and that the primary IRU can fail. CIRCA can do this because, unlike most other planners, [9] CIRCA considers and plans against, external events. CIRCA can warm up the IRUs early enough, because of its temporal reasoning.

However, this scenario has led us to consider two shortcomings of the current CIRCA approach. First, CIRCA considers exogenous processes only as threats, rather than as opportunities. CIRCA's planner only chooses either to preempt exogenous processes or allow them to happen. Accordingly, the current CIRCA world model provides only lower bounds on the delay of temporal transitions. This makes it impossible for CIRCA to rely on external processes (like the warming of an IRU), because doing so requires CIRCA to reason about the *upper bound* on the duration of the warming process.

A second shortcoming has to do with the lack of a systemwide clock. Currently, CIRCA can reason only about duration relative to the time it enters a particular state. In order to properly meet deadlines, as in this example, where the IRUs must be warmed prior to orbital insertion, the RTS must be able to act at an appropriate time relative to a planned future event.

We have developed preliminary solutions to the above two problems. The existing temporal model already takes into account some upper bounds — those on the duration of actions. We plan to expand the model to include *reliable temporals*, with upper bounds on their time of completion, together with state-encoding of the progress of those processes.

We are also addressing the problem of CIRCA not having a systemwide clock. We do not want to abandon the unclocked executive, because inclusion of global time into the state space can cause it to explode (see comparison to Kabanza's execution model earlier). What we would like to do is to provide chosen clock signals for particular times to the RTS. It is certainly possible to provide such signals — for most applications like autonomous

[9] With the exception of Blythe's [2] and Kabanza's [8].

spacecraft, there will be a system or mission clock. What we need to be able to do is to identify important times and set up signals to the RTS accordingly. The RTS will then detect these signals like any other state feature. Our preliminary investigations suggest that we can detect the need for such features through search failures in the AIS.

Overcoming these expressive limitations is an important area of ongoing theoretical investigation at HTC. We hope to begin experimenting with solutions to these problems sometime this year.

7.2. More Flexible State Refinement

In Section 3, we described experiments involving manually planning in a hypothetical distributed planning domain. Although these "thought experiments" were done assuming CIRCA's base level planning algorithm (rather than DAP), we observed that DAP could be improved if it were allowed to split states based not only into all the possible values of a domain of a feature, but to split into "F has value V" and "F has any other value."

Consider the following example, based on the multi-arm Puma domain of Section 3: As Agent-A is planning (abstractly), it realizes at some point that it needs to split to determine whether the feature `Location-of-A` has the value `at-box`. This value is important because, if it holds, Agent-A realizes that an important precondition for the synchronized packing action is satisfied. If the agent is anywhere else, its location has no effect on its choice of action. So, essentially treating an n-ary feature as a binary feature is sufficient for Agent-A to complete its planning. If the DAP planning algorithm could treat all locations other than `at-box` as equivalent, it could substantially reduce its overall state space.

It appears to be impossible for Agent-A to know that a given binary state refinement is sufficient the first time it comes up, since other paths may be discovered later that require more refinement of the "anything else" abstract value. Still, there should be no harm in trying to avoid extra state enumeration by starting with a binary split and adding detail only when necessary. New heuristics would certainly be needed to guide the additional splitting functionality.

7.3. Improving the Heuristic Function

The current heuristic function, based on the operator-proposition graph, is a clear improvement over the original depth-bounded lookahead heuristic of CIRCA. However, as our experiments have shown, there are cases that mislead the heuristic, leading the planner to waste a great deal of computational effort (cf. Section 6.3). In future work we intend to improve the heuristic function to avoid some of these pitfalls.

The heuristic in its current form does not actually *rank* action assignment and state refinement operations on degree of desirability. Instead it simply partitions the set of possibly applicable actions and splits into those that might take the agent towards its goal and those that will not. In most of the engineered domains, this was sufficient.

When we started to work on the artificial domains for evaluation, however, we encountered cases where this crude pruning of actions and splits was not sufficient. In particular, we encountered domains where:

1. There were multiple possibly-applicable actions and
2. Each such action required *multiple* state refinements in order to be applied.

In this situation what went wrong was simple: the heuristic would choose to refine the state on one feature that would enable an action, a_1. Then the refined state would be reconsidered and a second feature would be chosen for further refinement. Unfortunately the heuristic would not bias the choice of feature towards a second feature that would *also* make a_1 applicable. Instead, the heuristic would make an arbitrary choice and often would choose a feature that was not one of a_1's preconditions, but rather a precondition for some different action a_2. Indeed, because of arbitrary features of the code (the order in which features were defined in the domain descriptions), the heuristic would almost always make such a bad choice.

We intend to improve the choice of state refinements by using cost information inside the operator-precedence graph. When choosing a feature for state refinement, we will traverse the graph top-down, choosing a min-cost branch at each node. This should significantly improve the behavior of the DAP planner on examples like those in Section 6.3. We have already designed this improved heuristic function and are in the process of implementing it.

8. Conclusions

The DAP technique provides major advantages, including:

- The selection of which features to "abstract away" is performed automatically during planning.
- The abstractions are *local*, in the sense that different parts of the state space may be abstracted to different degrees.
- The abstractions preserve guarantees of system safety.
- The planning system need not plan to the level of fully-elaborated states to construct a feasible, executable plan.

In all but the most contrived domains, DAP dramatically outperforms the Classic CIRCA planner, generating much smaller state spaces and final plans using much less computation time. DAP represents a significant new contribution to the planning field, bringing practical automated abstraction to bear on the complexity problems that have long prevented successful application of this technology. We anticipate that further development and refinement of the DAP concept will lead to major improvements in our ability to apply planning technology to practical, large-scale domains.

References

[1] A. Barrett and D. Weld, "Partial Order Planning: Evaluating Possible Efficiency Gains," *Artificial Intelligence*, vol. 67, no. 1, pp. 71–112, 1994.

[2] J. Blythe, "A Representation for Efficient Planning in Dynamic Domains with External Events," in the AAAI workshop on "Theories of Action, Planning and Control: Bridging the gap", July 1996.

[3] E. H. Durfee, *Coordination of Distributed Problem Solvers*, Kluwer Academic, 1988.

[4] E. Gat, "News From the Trenches: An Overview of Unmanned Spacecraft for AI," in *AAAI Technical Report SSS-96-04: Planning with Incomplete Information for Robot Problems*, I. Nourbakhsh, editor. American Association for Artificial Intelligence, March 1996. Available at `http://www-aig.jpl.nasa.gov/home/gat/gp.html`.

[5] M. L. Ginsberg, "Dynamic Backtracking," *Journal of Artificial Intelligence Research*, vol. 1, pp. 25–46, 1993.

[6] R. P. Goldman, D. J. Musliner, M. S. Boddy, and K. D. Krebsbach, "The CIRCA Model of Planning and Execution," in *Working Notes of the AAAI Workshop on Robots, Softbots, Immobots: Theories of Action, Planning and Control*, 1997.

[7] R. P. Goldman, D. J. Musliner, K. D. Krebsbach, and M. S. Boddy, "Dynamic Abstraction Planning," in *Proc. Nat'l Conf. on Artificial Intelligence*, pp. 680–686, 1997.

[8] F. Kabanza, M. Barbeau, and R. St-Denis, "Planning Control Rules for Reactive Agents," Technical Report 197, Computer Science Dept., University of Sherbrooke, 1997.

[9] D. McDermott, "A Heuristic Estimator for Means-Ends Analysis in Planning," in *Proc. Third Int'l Conf. on Artificial Intelligence Planning Systems*, pp. 142–149, 1996.

[10] D. J. Musliner, M. S. Boddy, R. P. Goldman, and K. D. Krebsbach, "The Link Between Distributed Planning and Abstraction," in *Working Notes of the AAAI Fall Symposium on Model-Directed Autonomous Systems*, 1997.

[11] D. J. Musliner, E. H. Durfee, and K. G. Shin, "World Modeling for the Dynamic Construction of Real-Time Control Plans," *Artificial Intelligence*, vol. 74, no. 1, pp. 83–127, March 1995.

[12] D. J. Musliner and R. P. Goldman, "CIRCA and the Cassini Saturn Orbit Insertion: Solving a Prepositioning Problem," in *Working Notes of the NASA Workshop on Planning and Scheduling for Space*, October 1997.

[13] E. Pednault, "ADL: Exploring the middle ground between STRIPS and the situation calculus," in *First International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, Inc., 1989.

[14] E. D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, vol. 5, no. 2, pp. 115–135, 1974.

Appendix A.

The CIRCA Temporal Model

This section presents the temporal model used by the CIRCA state-space planner to reason about transition timing information and form guarantees that plans will avoid certain types of failures. Timing information for a CIRCA plan is derived from bounds on the delay associated with arcs out of a node in the NFA, which are taken directly from the delay bounds for the corresponding transitions. The *latency* of a transition arc with respect to a node in the plan is the time before that transition will occur, if no other transition occurs first, once some state in the set corresponding to that node has been reached. Latency bounds are path-dependent, which breaks the Markov assumption for nodes. We restore this property by calculating and employing path-independent bounds on latency in providing timing guarantees (most significantly, in determining preemption of transitions by actions).

A.1. Notation

- States: $s \in \mathcal{S}$.

  The set of states associated with a node in the plan graph we denote by $S \in 2^{\mathcal{S}}$. There being a $1:1$ relationship between sets S and nodes, we will use the set S to refer to the node.

- Transitions: $t \in \mathcal{T}$

  - $\mathsf{pre}(t)$ – preconditions of transition t
  - $S \models \Diamond t \equiv \exists s \in S, s \models \mathsf{pre}(t)$ – t is possibly enabled at node S
  - $d_{max}(t)$ – maximum delay for t
  - $d_{min}(t)$ – minimum delay for t

  The fact that latency bounds are calculated based on paths means that we must distinguish between arcs and the associated transitions.

- Arcs: $a \in \mathcal{A}$.

  - $\mathsf{transition}(a) \in \mathcal{T}$ – the transition label on a
  - $\mathsf{origin}(a) \in 2^{\mathcal{S}}$ – the node from which a leads
  - $\mathsf{result}(a) \in 2^{\mathcal{S}}$ – the resulting node.

  We will employ two syntactic substitutions:

  - $d_{min}(a)$, for $d_{min}(\mathsf{transition}(a))$ (also $d_{max}(a)$, *mutatis mutandis*)
  - $a \in (S, S')$, for $\mathsf{origin}(a) = S \wedge \mathsf{result}(a) = S'$

- Action assignments:

  - $\mathsf{action}(S)$ – the action assigned to state S

A.2. Transition Timing

We assume that the "clock" for transition delay starts as soon as an enabling node (one in which pre(t) is possibly satisfied) is entered, and stops only when either the transition occurs, or a node is entered in which pre(t) is not possibly satisfied. In particular, the clock keeps running across other transitions between enabling nodes. This assumption applies to all transitions, volitional and nonvolitional.

A.2.1. Events and Temporals

For nonvolitional transitions (temporals and events), the bounds $[d_{min}, d_{max}]$ are specified as part of the domain description. In the current implementations, those bounds are:

- temporals: $[d, \infty]$
- events: $[0, \infty]$

A.2.2. Actions

Actions are more complicated. The bounds on delay until an action happens are determined by the current TAP schedule, and by the delay associated with the action itself. One corollary of this statement is that the transition bounds for a given action are to a considerable extent determinable by the AIS.

Let's take a more detailed look at action timing. Under the control of a TAP schedule, the RTS takes a "snapshot" of the current state. It then evaluates that snapshot according to some test or sequence of tests, and decides whether or not to perform a given action. We assume complete and correct knowledge of the current state, so determination of what action to perform will be correct. The question is, how long will it take? There is a physical minimum time before the action could have an effect, consisting of the minimum time required for a test and the time required for the action itself. This is a "minimum upper bound" on the action (a lower bound on any specifiable d_{max}).

The precise execution of the TAP schedule is something we don't need to deal with at this point. For example, we neither need nor want to think about whether a single snapshot is tested for several actions, or whether each action takes its own snapshot. We assume that taking the snapshot, as opposed to testing, takes no time to accomplish. If this assumption is relaxed, there's another scheduling optimization involved about when snapshots get taken and which tests are done on which snapshot. The characteristic that must be preserved is that the test for a given action is performed on a succession of snapshots, taken with no more than a specified maximum separation. The maximum delay before the action takes effect is then the sum of that maximum separation, the test delay, and the time required for the action itself.

d_{max} for actions is not an intrinsic feature, it's a parameter set by the planner in the planning process. Faced with a temporal transition to preempt, the planner can

- Choose an old action with a (previously specified) sufficiently small d_{max} for that node.
- Choose an old action with an insufficiently small d_{max} and specify a new, tighter bound.

- Choose a new action for the node and specify a sufficiently small d_{max}.
- Split the state, etc...

Note that any or all of these plan modifications may require a new TAP schedule to be generated. This suggests that the planner and scheduler should operate in fairly close synchronization. The current TAP schedule limits the allowable values for d_{max} for a given action (which specification in turn constrains the space of feasible schedules), while the suitability or otherwise of an action to preempt a given transition (what was previously called "applicability") is determined by that same specification.

Ghosting and Inappropriate Actions A further complication with actions is that the test and action are not atomic. It is entirely possible for some nonvolitional transition to occur between the time that the current state is evaluated and the time the action takes effect. It is therefore possible for an action to be attempted in a state in which it is not technically "enabled."

The classical planning community calls these plans "ill-formed." For CIRCA, we adopt a similar convention, by defining the outcome of any such *inappropriate action* to be a failure state. Some inappropriate actions can be avoided by ensuring that the relevant (temporal) transitions are preempted. Events leading to unsuitable states cannot be preempted. This situation can be planned around, e.g. by splitting the node (separating the action and the event), splitting the event's destination (making the action be enabled in the result), choosing a different action, or declaring the current state a failure state as well.

A.3. Definitions

Preemption of one transition by another at a node is defined in terms of the latency bounds L_{min} and L_{max}:

$$\mathsf{preempts}(t, t', S) \equiv L_{max}(t, S) < L_{min}(t', S)$$

In words: t preempts t' in S iff t is guaranteed to occur before t' once S is reached, no matter how you got there.

The *maximum dwell* of a node S is relevant because we can guarantee that no transition out of that node will take place with a longer delay.

$$D_{max}(S) = \min_{a \in (S,X)} L_{max}(\mathsf{transition}(a), S)$$

The lower bound on latency for a transition t at a node S is the lower bound on delay for t, unless there are "enabling predecessors" (defined below), in which case the lower bound on latency is the minimum value derivable from those predecessors.

$$L_{min}(t, S) = \begin{cases} d_{min}(t) & \text{if } \mathsf{enabling\text{-}preds}(t, S) = \emptyset \\ \min_{S' \in \mathsf{enabling\text{-}preds}(t,S)} L^*_{min}(t, S, S') & \text{otherwise} \end{cases}$$

The lower bound on latency for S derivable from an enabling predecessor S' is recursively defined as $L_{min}(t, S')$, minus the maximum possible transition time from S' to S that does *not* follow transition of type t: $T^*_{max}(S', S, t)$.

$$L^*_{min}(t, S, S') = L_{min}(t, S') - \min(D_{max}(S'), \max_{a \in (S', S) \wedge \mathsf{transition}(a) \neq t} L_{max}(\mathsf{transition}(a), S'))$$

An *enabling predecessor* (enabling-preds) for t at S is any node S' at which t is enabled, from which S is reachable by an arc with some label *other than* t (otherwise the clock resets).

$$\mathsf{enabling\text{-}preds}(t, S) = \{S' | S \neq S', S' \models \Diamond t \wedge \exists a \in (S', S), \mathsf{transition}(a) \neq t\}$$

The upper bound on latency for t at S is the maximum delay $d_{max}(t)$, unless there are enabling predecessors, *et cetera.*

$$L_{max}(t, S) = \begin{cases} \max_{S' \in \mathsf{enabling\text{-}preds}(t,S)} L^*_{max}(t, S, S') & \text{if} \forall a, \mathsf{result}(a) = S \Rightarrow \mathsf{origin}(a) \in \mathsf{enabling\text{-}preds}(S \\ d_{max}(t) & \text{otherwise} \end{cases}$$

The upper bound on latency for S derivable from an enabling predecessor S' is recursively defined as $L_{max}(S')$ minus the minimum possible transition time from S to S'.

$$L^*_{max}(t, S, S') = L_{max}(t, S') - \min_{a \in (S', S) \wedge \mathsf{transition}(a) \neq t} L_{min}(\mathsf{transition}(a), S')$$

One of the interesting results of this timing model is that one can achieve "better than real-time" performance. Given a node with a troublesome temporal, say one where L_{min} is less than any achievable d_{max} for the desired action(s), preemption can be guaranteed by ensuring that the node is only reachable from nodes at which the action is enabled, and only via temporal transitions with a sufficiently large L_{min}. The current planner does not exploit this opportunity, and we have no immediate plans to do so.

There are several simplifications we can make. We start by assuming that $L_{max}(t, S) = d_{max}(t)$ in all cases. This assumption preserves the correctness of the latency bounds and preemption calculations, by virtue of the fact that $L_{max}(t, S) \leq d_{max}(t)$. The bound is weaker only in the somewhat peculiar "better-than-real-time" case described above.

This leads to additional simplifications. Here is the complete set of revised definitions. By assumption:

$$L_{max}(t, S) = d_{max}(t)$$

For the maximum dwell, we use the assumption above, plus the fact that there is exactly one action specified for an node in the plan graph ($d_{max}(\text{no-op}) = \infty$):

$$D_{max}(S) = d_{max}(\mathsf{action}(S))$$

L_{min} does not change. However, L^*_{min} does:

$$L^*_{min}(t, S, S') = L_{min}(t, S') - d_{max}(\mathsf{action}(S'))$$

It doesn't matter whether $\mathsf{result}(\mathsf{action}(S')) = S$ or not. Also note that the definition of enabling-preds has not changed.

A.4. Algorithm

Calculating L_{max} and D_{max} is reduced to lookup operations. The simplified definition of L^*_{min} above suggests a simple depth-first graph search, from nodes to their enabling predecessors. The algorithm has an additional termination condition: terminate with a bound of zero any time the summed "path cost" (d_{max} values for the appropriate actions) is greater than $d_{min}(t)$. This termination condition allows this algorithm to complete even in plans (graphs) with cycles: once the computed L_{min} along any path drops to (or below) 0, we're done.

Appendix B.

Limitations of CIRCA

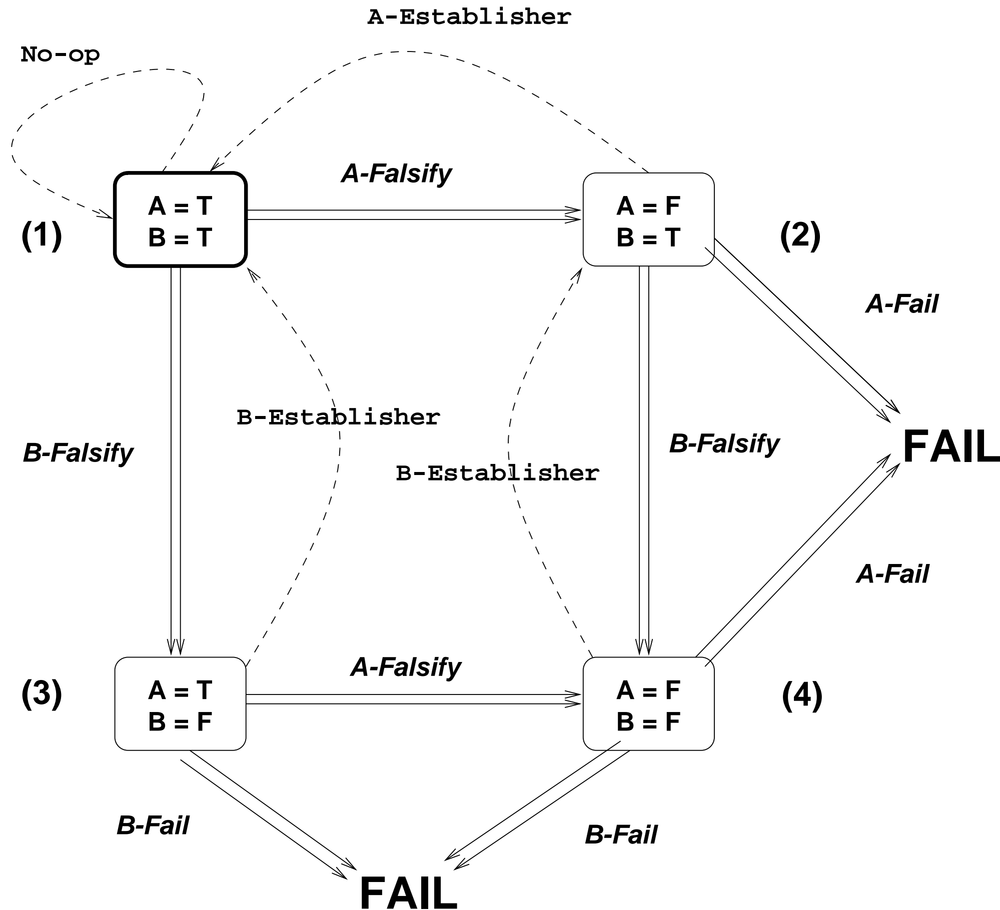


Figure B.1. A simple problem unsolvable by the CIRCA planner.

There are classes of real-time plans that CIRCA is unable to find because of the temporal reasoning it does. The problem arises because, instead of individually considering the possible ways that the CIRCA NFA could reach a particular state, CIRCA simply computes worst-case bounds on how much time it has to preempt a particular transition. In this section we give a simple example of such a problem.

Figure B.1 shows a simple, valid plan that CIRCA is unable to generate.[10] In this situation, there are two bad processes active, *A-Fail* and *B-Fail*. These temporal transitions to failure are active when the corresponding feature has the value false. There are temporal transitions that falsify the corresponding propositions, *A-Falsify* and *B-Falsify*. The CIRCA agent has at its disposal the two actions `A-Establisher` and `B-Establisher`, which make the corresponding propositions true.

The job of this simple controller is to react to the negation of A or B in a sufficiently timely way to preempt the corresponding transitions to failure. The CIRCA agent starts in the safe state, with the heavy outline, in the upper left corner of the diagram. With sufficiently fast actions, this is a valid CIRCA plan. Figure B.2 shows a trace of execution of this plan, illustrating that the plan is, in fact, safe.

[10]This example is an abstraction of a problem in the Puma robot arm domain.

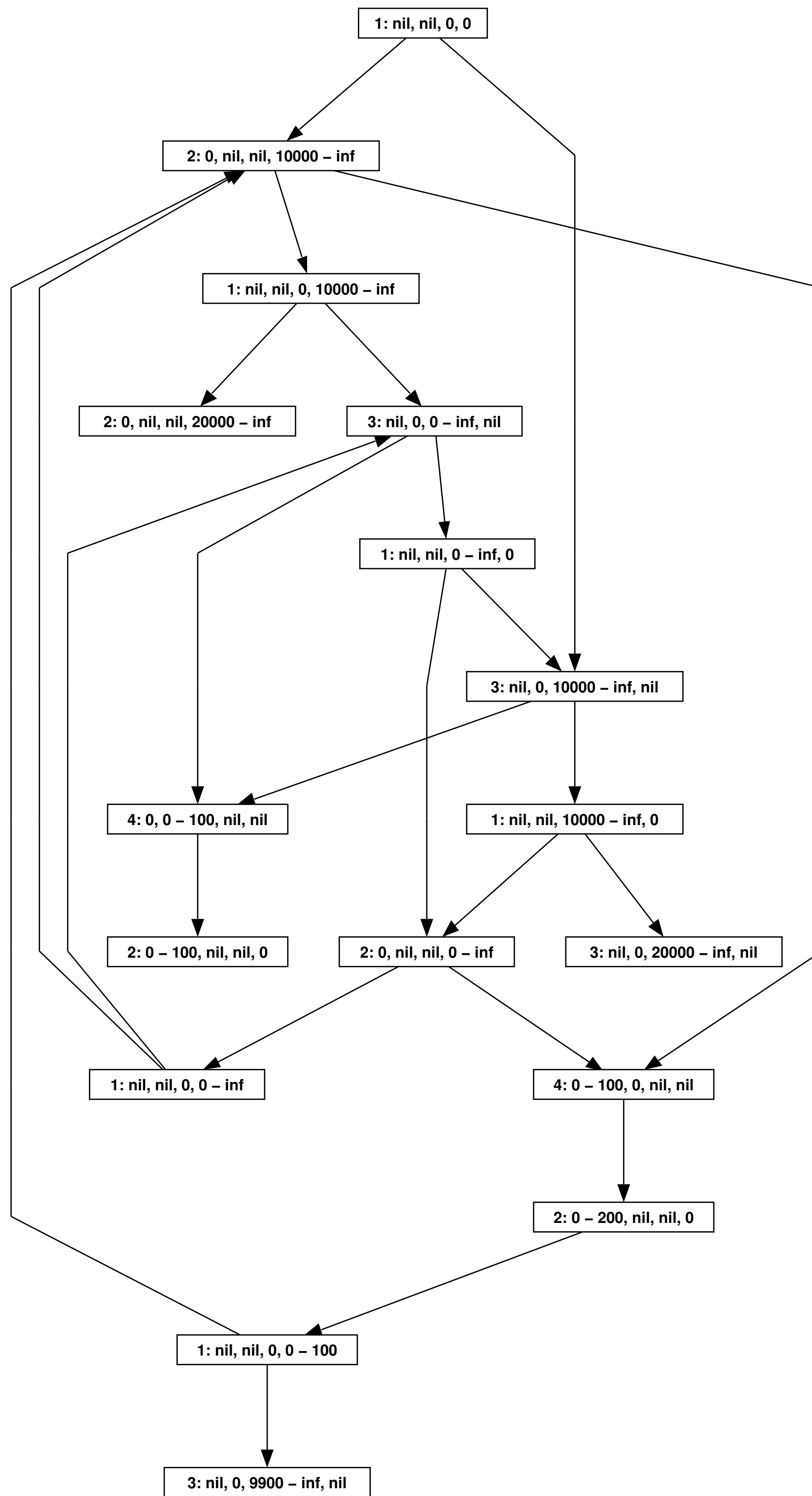


Figure B.2. A trace of the execution of the plan given in Figure B.1.

The nodes in the graph are labeled by state number, followed by upper and lower bounds on the amount of time subject to the various transitions (the entries are, in order, for *a-fail, b-fail, a-falsify* and *b-falsify*). For example, the root state of the trace indicates that we start execution in state 1. In state 1, *a-fail* and *b-fail* are not enabled (the corresponding entries are nil), and 0 time has been spent subject to *a-falsify* and *b-falsify*. One of the nodes immediately following this one is the result of following the *a-falsify* transition: CIRCA is now in state 2: *a-fail* is enabled; *b-fail* is not, nor is *a-falsify*. *b-falsify* is enabled and at least 10,000 time units have run on its "clock"; no upper bound on the time spent subject to this transition can be determined (so it will not be preemptible). The trace terminates when it reaches a state that is subsumed by one that has already been considered.

Why can't CIRCA find this simple, sound plan? The reason lies in the way the CIRCA planners (both original and DAP) compute temporal bounds in order to determine whether transitions will be preempted. When CIRCA considers whether a state is safe, it computes bounds on the latency of all temporals, based on all the possible ways of reaching that state. These bounds consider the best and worst cases for each transition *independently*. However, in order to determine that some plans, like the one in this example, are safe, one has to consider the interaction between temporals.

To see this, examine the trace in Figure B.2. In this trace, there are six nodes that correspond to being in state (1). In each such node, either *a-falsify* or *b-falsify* is preemptible. However, if we consider all possible ways of being in state 1, *neither* of the two temporals looks preemptible, because there is some way to reach state 1 that renders each unpreemptible.

Note that this is a problem of the CIRCA planner, not of the CIRCA execution model. The plan given in Figure B.1 *is* executable by the RTS. It is not necessary to consider how CIRCA reaches a state in order for it to execute the plan — only in order to determine that it can be executed correctly.