MIT-CSAIL-TR-2018-005 September 1, 2006

Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch and Bound

Robert Effinger

Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch and Bound

by

Robert T. Effinger IV

B.S. in Mechanical Engineering
Texas A&M University, 2003

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

Masters of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2006

Author…………………………………………………………………………………
Department of Aeronautics and Astronautics
August 25, 2006

Certified by……………………………………………………………………………
Brian C. Williams
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by…………………………………………………………………………...
Jaime Peraire
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch and Bound

by

Robert T. Effinger IV

Submitted to the
Department of Aeronautics and Astronautics on

August 25, 2006

In partial fulfillment of the requirements for the degree of
Masters of Science in Aeronautics and Astronautics

Abstract

Autonomous robots are being considered for increasingly capable roles in our society, such as urban search and rescue, automation for assisted living, and lunar habitat construction. To fulfill these roles, teams of autonomous robots will need to cooperate together to accomplish complex mission objectives in uncertain and dynamic environments. In these environments, autonomous robots face a host of new challenges, such as responding robustly to timing uncertainties and perturbations, task and coordination failures, and equipment malfunctions.

In order to address these challenges, this thesis advocates a novel planning approach, called temporally-flexible contingent planning. A temporally-flexible contingent plan is a compact encoding of methods for achieving the mission objectives which incorporates robustness through flexible task durations, redundant methods, constraints on when methods are applicable, and preferences between methods. This approach enables robots to adapt to unexpected changes on-the-fly by selecting alternative methods at runtime in order to satisfy as best possible the mission objectives. The drawback to this approach, however, is the computational overhead involved in selecting alternative methods at runtime in response to changes. If a robot takes too long to select a new plan, it could fail to achieve its near-term mission objectives and potentially incur damage.

To alleviate this problem, and extend the range of applicability of temporally-flexible contingent planning to more demanding real-time systems, this thesis proposes a temporally-flexible contingent plan executive that selects new methods quickly and optimally in response to changes in a robot's health and environment. We enable fast and optimal method selection through two complimentary approaches. First, we frame optimal method selection as a constraint satisfaction problem (CSP) variant, called an Optimal Conditional CSP (OCCSP). Second, we extend fast CSP search algorithms, such as Dynamic Backtracking and Branch-and-Bound Search, to solve OCCSPs. Experiments on an autonomous rover test-bed and on randomly generated plans show that these contributions significantly improve the speed at which robots perform optimal method selection in response to changes in their health status and environment.

Thesis Supervisor: Brian C. Williams

Title: Associate Professor of Aeronautics and Astronautics

Acknowledgements

I would like to thank my advisor, Prof. Brian Williams, for his key insights and technical guidance which have made this thesis possible. From my first course in Artificial Intelligence to the final edits of this thesis, his dedication to my professional growth has transformed my inexperienced enthusiasm into publishable work.

I would like to thank my entire family for their unconditional love and support, and for encouraging me to always pursue my dreams.

I would like to thank Jon Kennel, I-hsiang Shu, and Seung Chung for their invaluable technical mentorship. I would like to thank Paul Robertson, Hui Li, and Paul Elliot for carefully reviewing drafts of this thesis. I would like to thank the entire Model-based and Embedded Robotic Systems (MERS) group for being such a fun and exciting group of people to work with.

This research was funded by the NASA Human and Robotic Technology program under grant [NNA04CK91A], and the DARPA Self Regenerative Systems program under grant [FA8750-04-2-0243].

Contents

CHAPTER 1 - INTRODUCTION 11

1.1 A Motivating Example 13
1.2 Temporally-Flexible Contingent Planning 15
1.3 Problem Statement 19
1.4 Thesis Contributions 19
1.5 Thesis Layout 21

CHAPTER 2 – BACKGROUND AND RELATED WORK 22

2.1 Continuous Planning, HTN Planning, and Temporally-Flexible Planning 22
2.2 Constraint Satisfaction 23
2.2.1 Constraint Satisfaction Problem (CSP) 23
2.2.2 Generate and Test Framework 24
2.2.3 Chronological Backtracking (BT) 26
2.2.4 Dynamic Backtracking (DB) 27
2.2.5 Optimal Conditional CSP (OCCSP) 32

CHAPTER 3 – TEMPORALLY-FLEXIBLE CONTINGENT PLANNING 35

3.1 Reactive Model-based Programming Language (RMPL) 36
3.2 Temporal Plan Network (TPN) 38
3.3 Optimal Method Selection 40
3.4 Plan Reformulation and Dispatching 47

CHAPTER 4 – FRAMING OPTIMAL METHOD SELECTION AS AN OCCSP 48

4.1 Encoding a TPN and its Primitive Activities as an OCCSP 49
4.1.1 The LRV Deployment Scenario Encoded as an OCCSP 51
4.2 Constructing a TPN from RMPL Expressions 55
4.2.1 TPN Primitive Activity 56
4.2.2 TPN Composition Operator 56
4.2.3 Sequence 58
4.2.4 Parallel 59
4.2.5 Choose 60
4.2.6 If Then Else 60
4.2.7 When Then 61
4.2.8 Do Maintaining 61
4.3 Framing Optimal Method Selection as an OCCSP 63
4.3.1 Phase 1: Candidate Execution Generation 63
4.3.2 Phase 2: Temporal Consistency Check 63
4.3.3 Phase 3: Symbolic Consistency Checking 64
4.3.3 Phase 4: Completeness and Optimality Check 68

CHAPTER 5 – EXTENDING DYNAMIC BACKTRACKING TO SOLVE OCCSPS 70

5.1 Extension #1: A Systematic Method for Searching over Conditional Variables 72
5.2 Extension #2: OCCSP Backjumping Resolution Step 74
5.3 Extension #3: Checking for Deactivated Variables 75
5.4 Extension #4: A Branch-and-Bound Framework 76
5.5 Taking CondDB-B+B for a Test-Drive 77
5.6 Sketching Out Proofs of Completeness and Termination 79
5.6.1 Completeness 80
5.6.2 Termination 81

CHAPTER 6 – IMPROVING TPN SEARCH EFFICIENCY 82

6.1 – USING CONFLICTS TO GUIDE OPTIMAL METHOD SELECTION 82
6.1.1 Detecting Temporal Conflicts in TPNs 83
6.1.2 Using Conflicts to Guide Optimal Method Selection 83
6.2 – A TECHNIQUE FOR COMPUTING TIGHT LOWER BOUNDS 87
6.3 – A RELAXED UNION OPERATOR THAT ENABLES EARLY DETECTION OF TEMPORAL CONFLICTS 90
6.3.1. Defining the Relaxed Union Operator for TPNs 91
6.4 – EXTRACTING FOCUSED TEMPORAL CONFLICTS FROM A TPN 92
6.4.1 Identifying Irrelevant Variable-Value Assignments 93
6.4.2 Identifying Large Sets of Irrelevant Variable-Value Assignments 95

CHAPTER 7 – EMPIRICAL RESULTS, DISCUSSION, IMPLEMENTATION, FUTURE WORK, AND CONCLUSIONS 98

7.1 OVERVIEW 98
7.2 EMPIRICAL RESULTS AND DISCUSSION 98
7.2.1 Random TPN Generator 98
7.2.2 The Algorithms 99
7.2.3 Empirical Results and Discussion 101
7.3 IMPLEMENTATION DETAILS 105
7.4 FUTURE WORK 105
7.4.1 Improving the Temporally-Flexible Contingent Planning Architecture 105
7.4.2 Incorporating other OCCSP Search Techniques 108
7.5 CONCLUSIONS 109

BIBLIOGRAPHY 110

Table of Figures

Figure 1.1: An Illustration of LRV Deployment (*courtesy of NASA) 13
Figure 1.2: The Apollo LRV Deployment Sequence (*courtesy of NASA) 14
Figure 1.3: A Temporally-Flexible Contingent Plan encoded in RMPL. 15
Figure 1.4: Execution of Temporally-Flexible Contingent Plans. 17
Figure 1.5: A Temporally-Flexible Contingent Plan encoded as a TPN. 17
Figure 2.1: A Map Coloring Example 29
Figure 2.2: Recording the first Elimination Explanation 29
Figure 2.3: England’s domain is exhausted. 30
Figure 2.4: Denmark’s color assignment is retracted. 30
Figure 2.5: Bulgaria’s color assignment is retracted. 31
Figure 2.6: Dynamic Variable Reordering (Bulgaria and Czechoslovakia). 31
Figure 2.7: An example OCCSP, a car buyer’s configuration task. 34
Figure 3.1: Kirk: A Temporally-Flexible Contingent Plan Executive. 35
Figure 3.2: RMPL Constructs Supported by the Kirk Planner. 36
Figure 3.3: LRV Deployment Example encoded in RMPL 37
Figure 3.4: Mapping from RMPL to TPN. 39
Figure 3.5: TPN encoding of the LRV deployment example. 39
Figure 3.6: Optimal Method Selection 40
Figure 3.7: Four Candidate Executions for the LRV Deployment Scenario. 42
Figure 3.8: Conflicting state assertions 45
Figure 3.9: Two possible orderings to resolve conflicting state assertions. 45
Figure 3.10: Ask Consistency Constraints 46
Figure 4.1: TPN Primitive Activity. 49
Figure 4.2: Labeling and encoding each primitive activity in the TPN. 52
Figure 4.3: TPN_1 and TPN_2 56
Figure 4.4: Composing TPN_1, TPN_2, and t_1 into a new TPN, called TPN_{new}. 57
Figure 4.5: Sequence RMPL Expression. 59
Figure 4.6: Parallel RMPL Expression. 59
Figure 4.7: Choose RMPL Expression. 60
Figure 4.8: If Then Else RMPL Expression 61
Figure 4.9: When Then RMPL Expression 61
Figure 4.10: Do Maintaining RMPL Combinator 62
Figure 4.11: Conflicting state assertion constraints that threaten to co-occur. 64
Figure 4.12: Resolving Conflicting State Assertions Framed as Solving an OCCSP 65
Figure 4.13: Examples of state requests, Ask(a = 1) and Ask(b = 2). 66
Figure 4.14: Satisfying state requests framed as OCCSP variables and values 67
Figure 5.1: Dependency graph for the car configuration task in Section 2.3.5. 73
Figure 5.2: Solving the car buyer OCCSP with CondDB-B+B 78
Figure 6.1: A Temporal Conflict, { i_A=v_A, i_1=v_1, i_2=v_1, i_3=v_1}. 83
Figure 6.2: Conflict-directed Optimal Method Selection 84
Figure 6.3: Conflict-directed Candidate Execution Generation 86
Figure 6.4: A Partial Execution, {i_1=v_2} 87

Figure 6.5: A Pre-processing Step for Computing Tight Lower Bounds. 89
Figure 6.6: Branch and Bound Search with and without tight lower bounds. 90
Figure 6.7: A Relaxed Constraint: The Feasible Bound on a Choice Expression. 91
Figure 6.8: The Union and Relaxed Union Operators 92
Figure 6.9: An Alternative Way to View of a Temporal Conflict: A Conflict Timeline. 93
Figure 6.10: A Tricky Conflict. 96
Figure 7.1: Two Dimensions of Complexity in a TPN... 99
Figure 7.2: Average Case Performance for Experiment #1 102
Figure 7.3: Individual Test Cases for # of Choices in Parallel (b) 102
Figure 7.4: Average Test Results for # of Nested Choices (n) 104
Figure 7.5: Individual Test Results for # of Nested Choices (n) 104
Figure 7.6: Temporally-Flexible Contingent Planning Architecture. 106
Figure 7.7: A Contingency Plan for the R1.remove(insulation-blanket) Activity.......... 108
Figure 7.8: Contingency Handling Operator: RMPL to TPN. 108

Chapter 1 - Introduction

As Artificial Intelligence (AI) methods steadily mature, and processor speed continues to improve, autonomous robots are being considered for increasingly capable roles in our society. Three such roles are autonomous search and rescue [24], automation for assisted living [32], and autonomous lunar habitat construction [3]. To fulfill these ambitious roles, teams of autonomous robots will need to cooperate together to accomplish complex mission objectives. Traditionally, however, cooperation between robots has been restricted to tightly scheduled and choreographed routines involving only a small number of robots. Additionally, robots have traditionally operated only in structured environments where changes are strictly regulated. This will no longer be the case in uncertain and dynamic environments such as collapsed buildings or on the surface of the moon. To operate reliably in these environments, autonomous robots face a host of new challenges, such as responding robustly to timing uncertainties and perturbations, task and coordination failures, and equipment malfunctions. These challenges pose difficulties for traditional planning techniques which assume static conditions and predictable outcomes, leading to a need for new approaches to planning.

In order to address these challenges, this thesis advocates a novel planning approach, called temporally-flexible contingent planning [52]. A temporally-flexible contingent plan is a compact encoding of strategies for achieving the mission objectives which incorporates robustness through flexible task durations [7], redundant methods [19], constraints on when methods are applicable [45], and preferences between methods [45]. This approach enables robots to adapt to unexpected changes on-the-fly by selecting alternative methods at runtime in order to satisfy as best possible the remaining mission objectives. The drawback to this approach, however, is the computational overhead involved in selecting alternative methods at runtime in response to changes. If a robot takes too long to select a new set of methods, it could fail to achieve its near-term mission objectives and potentially incur damage. For example, consider a space probe performing a time-critical aero-braking maneuver. If a thruster fails during aero-braking, the probe will need to select alternative methods so that the probe performs an equivalent maneuver while avoiding the failed thruster. Imagine, however, that the probe takes too long to select the appropriate methods and overheats while plowing too deeply into the atmosphere. This scenario highlights an important point. When an autonomous robot takes too long to select alternative methods in response to changes at runtime, it can fail to accomplish its near-term mission objectives and potentially incur damage from the environment. This is particularly important to avoid in critical real-time systems such as rescue robots and space satellites.

The central contribution of this thesis is to extend the range of applicability of temporally-flexible contingent planning to more demanding real-time systems by minimizing the occurrence such failures. To achieve this, we propose a novel temporally-flexible contingent plan executive that selects alternative methods quickly and optimally in response to changes in a robot's health status and environment. We enable fast and optimal method selection through two complimentary approaches. First, we frame optimal method selection as a constraint satisfaction problem (CSP) variant, called an Optimal Conditional CSP (OCCSP) [17,21]. Second, we extend fast CSP search algorithms, such as Dynamic Backtracking [13] and Branch-and-Bound Search [37], to solve OCCSPs. These algorithms build upon the ideas of conflict-directed search and optimal heuristic search, which reason on the structure of a problem to guide the search towards an optimal and consistent solution. Experiments on an autonomous rover test-bed and randomly generated plans show that these contributions significantly improve the speed at which robots can perform optimal method selection in response to changes in their health status and environment.

The remainder of this chapter is organized as follows. First, we motivate the need for temporally-flexible contingent planning through an example lunar exploration scenario. Then, we present a high-level overview of temporally-flexible contingent planning, followed by a statement of the problem being addressed by this thesis. Finally, we describe our proposed solution approach, and give an outline for the rest of this thesis.

1.1 A Motivating Example

To motivate the need for temporally-flexible contingent planning, consider NASA's Vision for Space Exploration [29]. NASA intends to construct a manned base on the Moon by the year 2020. To decrease mission costs and reduce Astronaut risk, NASA envisions several robotic precursor missions to the Moon in order to identify future lunar base locations and also to construct a lunar habitat before the Astronauts arrive. In this ambitious example, we assume that NASA has sent two humanoid robots to the South Pole-Aitken Basin, and their mission is to scout out potential lunar base locations. The robots have traveled to the Moon in an Apollo-style spacecraft, and to increase their scouting mobility, the robots' first task upon arrival is to deploy an Apollo-style Lunar Roving Vehicle (LRV). Therefore, in this example, the robotic assistants must execute the exact same LRV deployment sequence as the original Apollo Astronauts. Figure 1.1 shows an illustration of two Apollo astronauts deploying the LRV, and Figure 1.2 shows the original Apollo LRV deployment sequence. First, one astronaut removes the insulation blanket and operating tapes. Next, the two astronauts simultaneously lower the LRV and deploy the front and aft wheels by pulling on the deployment cables. Finally, the astronauts unfold the seats and footrests.

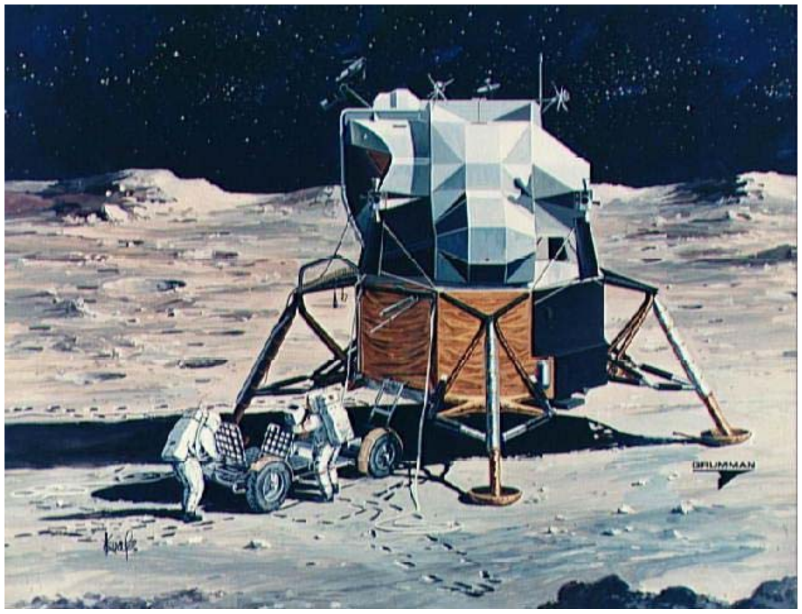


Figure 1.1: An Illustration of LRV Deployment (*courtesy of NASA)

This example highlights the need for many of the capabilities of temporally-flexible contingent planning. The two robots must coordinate together to accomplish a rather complex goal; successfully deploying the LRV. They must operate reliably in the uncertain lunar environment by adapting to plan perturbations and failures such as stumbling over a rock, missing a rung on the ladder, and equipment malfunctions. In addition, the robots must perform complex tasks in unison, such as grasping and pulling on the deployment cables and unfolding the seats and footrests. The next section provides a high-level overview of temporally-flexible contingent planning.

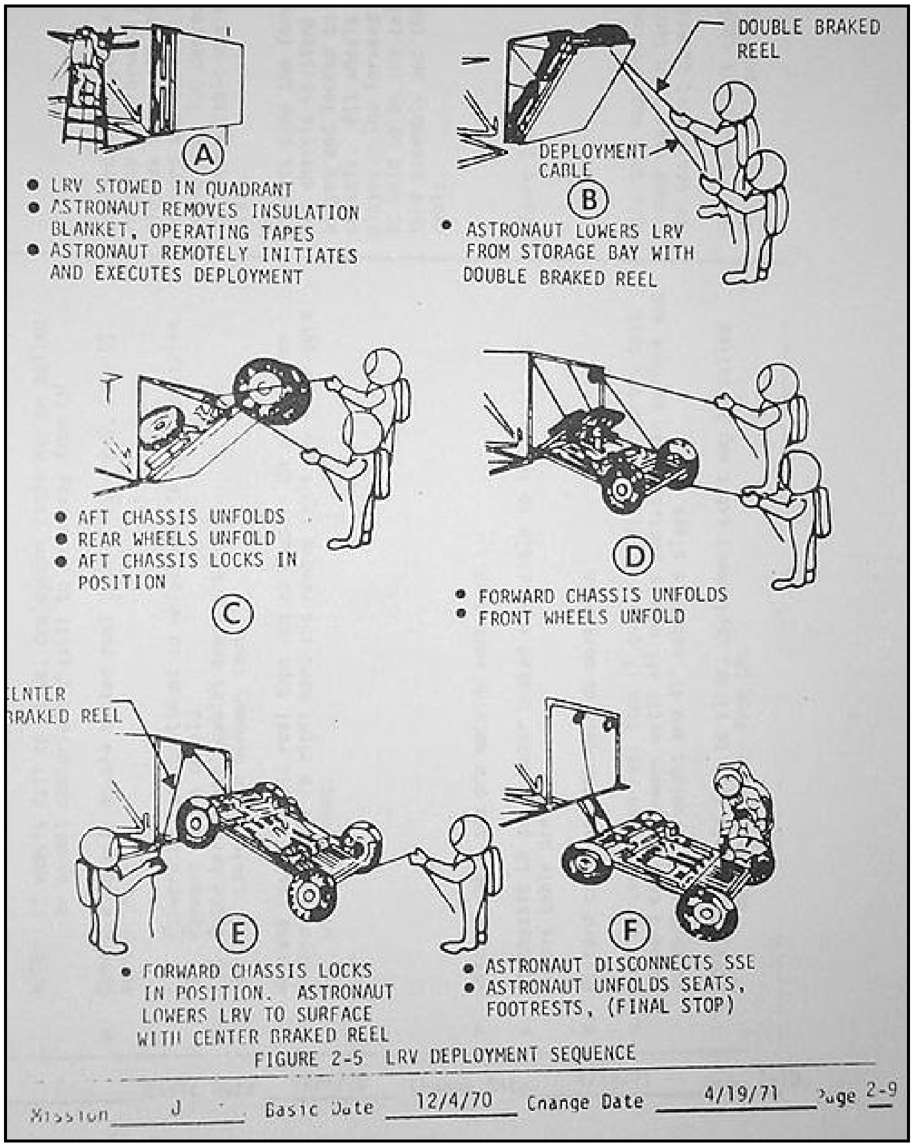

DOUBLE BRAKED REEL
DEPLOYMENT CABLE
A
• LRV STOWED IN QUADRANT
• ASTRONAUT REMOVES INSULATION BLANKET, OPERATING TAPES
• ASTRONAUT REMOTELY INITIATES AND EXECUTES DEPLOYMENT
B
• ASTRONAUT LOWERS LRV FROM STORAGE BAY WITH DOUBLE BRAKED REEL
• AFT CHASSIS UNFOLDS
• REAR WHEELS UNFOLD
• AFT CHASSIS LOCKS IN POSITION
C
D
• FORWARD CHASSIS UNFOLDS
• FRONT WHEELS UNFOLD
CENTER BRAKED REEL
E
• FORWARD CHASSIS LOCKS IN POSITION. ASTRONAUT LOWERS LRV TO SURFACE WITH CENTER BRAKED REEL
F
• ASTRONAUT DISCONNECTS SSE
• ASTRONAUT UNFOLDS SEATS, FOOTRESTS, (FINAL STOP)
FIGURE 2-5 LRV DEPLOYMENT SEQUENCE
Mission J Basic Date 12/4/70 Change Date 4/19/71 Page 2-9

Figure 1.2: The Apollo LRV Deployment Sequence (*courtesy of NASA)

1.2 Temporally-Flexible Contingent Planning

Temporally-flexible contingent planning is a novel approach towards writing and executing control programs for autonomous robots, which builds upon an idea called model-based programming [15,27]. Model-based programming enables a mission programmer to write control programs at a common-sense level using intuitive language constructs to compactly encode a robot's capabilities, mission objectives, and environment. Temporally-flexible contingent plans extend model-based programs by incorporating temporal-flexibility. Temporal-flexibility is enabled through flexible time windows on the duration of activities, allowing autonomous robots to adapt more reliably to timing uncertainties and perturbations at runtime.

In this thesis, we encode temporally-flexible contingent plans using the Reactive Model-based Programming Language (RMPL) [15,49]. RMPL builds upon previous work in robotic execution languages, such as RAPS [9], ESL [11], and TDL [40]. Next, to introduce temporally-flexible contingent planning and the RMPL language, we walk through a simple example, depicted in Figure 1.3. Suppose that an autonomous robot is tired of doing its reinforcement learning exercises and wants to take a study break for 2 to 3 hours. The robot knows of three redundant strategies for taking a study break: sailing, hiking, and watching a movie. In Figure 1.3, the robot's choice between redundant strategies is encoded using an intuitive RMPL language construct, called a choose operator. In addition, each strategy may have associated constraints and

```
study-break() [2,3] = { ;;high-level mission objective
  parallel(
    choose(  ;;redundant methods to choose from
      do( sailing=1() [2,4] ) maintaining ( weather = sunny ),
      hiking=2() [4,5],
      do( watch-movie=3() [1.5,3] ) maintaining (weather = raining)
    )
    sequence( ;;the predicted state of the environment
      ( weather = raining ) [0,4]
    )
  )
}
```

Figure 1.3: A Temporally-Flexible Contingent Plan encoded in RMPL.

preferences. For example, suppose the robot prefers sailing over hiking, and hiking over watching a movie. This is encoded via the, =1, =2, and =3, preference values listed next to each respective strategy. In addition, suppose the robot's choice depends upon the state of the environment. These constraints can be encoded via do-maintaining language constructs which constrain a strategy to be executed only while a certain maintenance condition holds; in this case the weather. Also, the allowable temporal durations for each strategy, and for the overall mission, can be specified via numbers contained in brackets. For example, the overall mission objective is to take a study break for 2 to 3 hours, denoted [2,3], and the watch-movie strategy is known to last from 1.5 to 3 hours, denoted [1.5,3]. Once a strategy has been encoded in RMPL along with its associated timing constraints, maintenance constraints, and preferences, it is called a method. Finally, the state of the environment is encoded into the plan via state assertions. Suppose the robot checks the weather forecast, and the weather is predicted to rain for up to 4 hours. This is encoded with the state assertion Tell(weather = raining)[0,4]. Temporally-flexible contingent plans can be encoded into RMPL by a mission programmer, or alternatively, as suggested in this example, by another autonomous entity, such as a generative planner [16,8]. Next, we describe how an autonomous robot executes such a plan.

Temporally-flexible contingent plans are executed by a temporally-flexible contingent plan executive as a two step, iterative process, which is depicted in Figure 1.4. The first step, called optimal method selection, is to choose from among the redundant methods, the most preferred set of methods which satisfies the timing and maintenance constraints. In general, the number of possible combinations of methods to consider is exponential in the number choices in the plan. To improve the speed of finding an optimal set of methods, an RMPL control program is first converted into an equivalent plan graph representation, called a Temporal Plan Network (TPN), so that network and graph theory algorithms can be used to quickly test its timing and maintenance constraints for consistency. A detailed explanation of TPNs, and how they are checked for consistency, is provided in Chapter 2. The Temporal Plan Network encoding of the study-break example is presented in Figure 1.5. In this example, there is only a single choice with 3 alternative methods, so optimal method selection simplifies to just considering each method in order of preference. First, consider the "sailing" method.

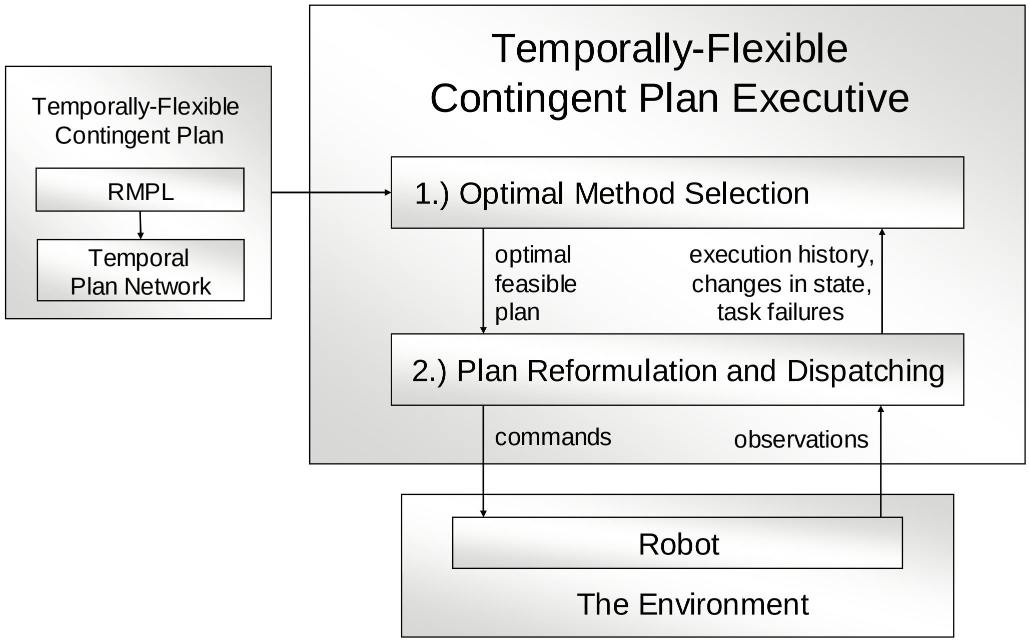

Temporally-Flexible Contingent Plan
RMPL
Temporal Plan Network
Temporally-Flexible Contingent Plan Executive
1.) Optimal Method Selection
optimal feasible plan
execution history, changes in state, task failures
2.) Plan Reformulation and Dispatching
commands
observations
Robot
The Environment

Figure 1.4: Execution of Temporally-Flexible Contingent Plans.

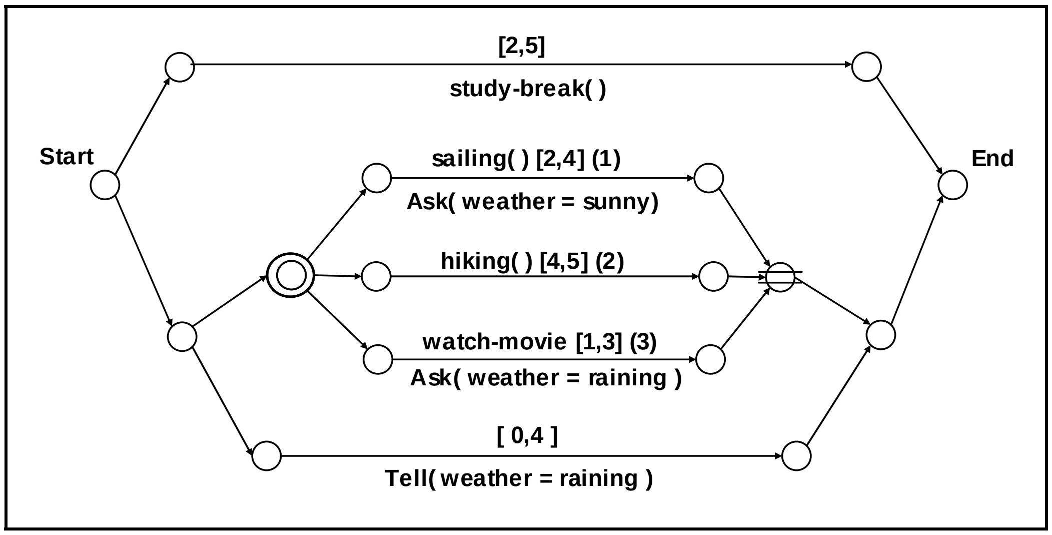


Figure 1.5: A Temporally-Flexible Contingent Plan encoded as a TPN.

Although it is the most preferred method for achieving the mission objective, it is not feasible, because the weather is predicted to rain. Next, consider the “hiking” method. While there is no maintenance constraint regarding the weather, hiking takes at least 4 hours to complete, exceeding the allowable duration of the study-break. Finally, consider the “watching-a-movie” method. While it is the least preferred method, it satisfies the timing and maintenance constraints and is therefore selected for execution. The methods selected for execution are collectively referred to as the optimal feasible plan, and are passed on to step 2.

During step 2, called plan reformulation and dispatching, an optimal feasible plan is prepared for execution (called reformulation) and then executed by the dispatcher. To execute a plan, the dispatcher traverses the plan sending commands at appropriate times to the robot. Additionally, the dispatcher continuously monitors the robot’s progress and immediately reports any task failures or changes in the state of the robot and environment. A detailed explanation of plan reformulation and dispatching is provided in Chapter 2.

Each time the dispatcher observes a task failure or a change in state, the currently selected plan is re-checked for validity. This is accomplished by passing information regarding execution history, task failures, and state changes back to step 1, and invoking optimal method selection again. If the previously selected plan becomes invalidated or suboptimal, a new optimal feasible plan will be selected for execution. For example, suppose that our study-break robot, an hour into watching its movie, decides to second guess the ‘human’ weatherman, and looks outside to check the weather for itself. Further reinforcing its distrust of human judgment, the robot sees sunny skies without a cloud in sight, and deduces a new state estimate: (weather = sunny)[0,4]. In general, state estimates can come directly from the robot’s observations, such as this one, or from a deductive controller [51], which infers hidden state from the robot’s observations. The robot passes this new information into its contingent plan executive, re-invoking optimal method selection. In this example, changing the state of the weather from raining to sunny revalidates the robots most preferred strategy, sailing, while simultaneously invalidating its current strategy, watching a movie, due to the maintenance constraints specified on each activity: `do(sailing=1()[2,4]) maintaining (weather = sunny)`, and `do( watch-movie=3() [1.5,3] ) maintaining (weather = raining)`.

This example illustrates how a temporally-flexible contingent plan executive enables robots to adapt to changes by selecting alternative methods at runtime in order to satisfy as best possible the mission objectives.

1.3 Problem Statement

In dynamic and uncertain environments, autonomous robots face a host of new challenges, such as responding robustly to timing uncertainties and perturbations, task and coordination failures, and equipment malfunctions. To handle such failures, an autonomous robot needs to adapt to unexpected changes on-the-fly by selecting alternative methods at runtime in order to satisfy as best possible the mission objectives. In addition, alternative methods need to be selected quickly and optimally. Otherwise, an autonomous robot could fail to achieve its near-term mission objectives or potentially incur damage from the environment while waiting for new methods to be selected. To enable reliable operation of autonomous robots in dynamic and uncertain environments, a temporally-flexible contingent plan executive needs to be developed that can select alternative methods quickly and optimally in response to changes in a robot's health status and environment.

1.4 Thesis Contributions

The central contribution of this thesis is to extend the range of applicability of temporally-flexible contingent planning to more demanding real-time systems. To achieve this, we propose a novel temporally-flexible contingent plan executive that selects alternative methods quickly and optimally in response to changes in a robot's health status and environment. We enable fast and optimal method selection through two complimentary approaches. First, we frame optimal method selection as a constraint satisfaction problem (CSP) variant, called an Optimal Conditional CSP (OCCSP). Second, we extend fast CSP search algorithms, such as Dynamic Backtracking (DB) and Branch-and-Bound Search (B+B), to solve OCCSPs. These algorithms build upon the ideas of conflict-directed search and optimal heuristic search, which reason on the structure of a problem to guide the search towards an optimal and consistent solution. To be specific, this thesis makes five key contributions, each of which reduces the number of candidate executions considered during optimal method selection:

1.) Frames Optimal Method Selection as an Optimal Conditional CSP (OCCSP).

- By encoding the dependencies and preferences between choices in a Temporal Plan Network (TPN) as activity and soft constraints, respectively, in an OCCSP, we can draw upon the advanced methods of constraint satisfaction to perform optimal method selection, in the form of search over a TPN, more efficiently.

2.) Extends Dynamic Backtracking (DB) to solve OCCSPs.

- Through novel enhancements to the Dynamic Backtracking algorithm, we extend DB to solve OCCSPs by utilizing activity constraints and soft constraints to quickly prune infeasible and suboptimal regions of the search space.

3.) Enables Conflict-directed Search of TPNs

- We develop a conflict-directed candidate execution generator that uses temporally inconsistent and sub-optimal partial executions, called *conflicts*, to guide optimal method selection to an optimal, complete, and consistent execution.

4.) Develops a Tight Bound for Branch and Bound Search of TPNs.

- We develop a method which exploits the hierarchical structure of a TPN to tighten the lower bound computed during Branch and Bound search.

5.) Develops a Relaxed Union Operator for TPNs.

- We develop a relaxed union operator that enables early detection of temporal conflicts in TPNs.

6.) Extracts Focused Temporal Conflicts from TPNs.

- First, we develop a method to identify irrelevant variable-value assignments in a temporal conflict. Then, we develop a method to identify large sets of irrelevant variable-value assignments in a temporal conflict. This capability enables us to significantly focus conflict-directed search of TPNs.

Experiments performed on an autonomous rover test bed and on randomly generated plans show that these contributions significantly improve the speed at which robots can perform optimal method selection in response to changes in their health status and environment.

1.5 Thesis Layout

This thesis is organized as follows. Chapter 2 provides required background information and summarizes several areas of related research. Chapter 3 introduces temporally-flexible contingent planning. Chapter 4 frames optimal method selection as an OCCSP. Chapter 5 extends the DB algorithm to solve OCCSPs. Chapter 6 introduces four techniques for improving TPN search efficiency. Chapter 7 provides an empirical evaluation, discussion of the results, implementation details, recommendations for future work, and conclusions.

Chapter 2 – Background and Related Work

The ideas developed in this thesis build upon previous work from several areas of research: hierarchical task network (HTN) planning, continuous planning, temporally-flexible planning, and constraint satisfaction. This chapter summarizes relevant topics from each of the aforementioned research areas.

2.1 Continuous Planning, HTN Planning, and Temporally-Flexible Planning

Temporally-flexible contingent planning combines previous work in continuous planning, hierarchical task network (HTN) planning, and temporally-flexible planning. In this section, we briefly discuss related work in each of these areas.

Continuous planning enables robust plan execution in uncertain and dynamic environments by considering plans as open-ended and continuously evolving in response to changes. Continuous planners, such as Aspen [34] and CPEF [28], enable fast adaptation of plans through correctness-preserving methods and fast plan repair techniques such as dependency-structure maintenance [28] and incremental reasoning [38,7].

Hierarchical task network (HTN) planners enable fast online planning by limiting the search for valid plans to a predefined library of strategies. To incorporate robustness into HTN planners, such as SHOP2 [30] and Aspen [34], strategies are encoded hierarchically within redundant methods for achieving a goal. In this way, an exact strategy to achieve each goal is not picked beforehand, but instead is hierarchically decomposed at run-time to ensure that the plan executes correctly given the state of the environment.

Temporally-flexible planners, such as HSTS [25] and Kirk [18], enable robust plan execution by incorporating temporal flexibility into plan specifications. These planners are able to adapt to timing uncertainties and perturbations at runtime by only imposing those temporal constraints required to guarantee a plan's success, leaving flexibility in the execution time of activities. This flexibility is then exploited, in order to adapt to uncertainty, by delaying the scheduling of each activity until it is executed. Temporally-flexible plans are enabled via simple temporal constraints, which bound the minimum and maximum duration of an activity, and were first introduced by Dechter, Meiri, and Pearl [7], called simple temporal problems (STPs). Fast and efficient execution of STPs was pioneered by Muscetolla et. al. via a novel reformulation algorithm [26]. In related work, STPs have been extended to specify preferences over when within the allowable time interval an activity is completed, as well as to support arbitrary disjunctions of simple temporal constraints, called disjunctive temporal problems with preferences (DTPP) [31]. In addition, recent work has mapped DTPPs to meta-weighted CSP problems [23]. Next we review relevant concepts in constraint satisfaction.

2.2 Constraint Satisfaction

In this section we summarize relevant topics in constraint satisfaction. Specifically, we review the Constraint Satisfaction Problem (CSP), the Chronological Backtracking (BT) and Dynamic Backtracking (DB) search algorithms for solving CSPs, and the Optimal Conditional CSP (OCCSP). In Chapter 4, optimal method selection is framed as an OCCSP, and in Chapter 5, Dynamic Backtracking is extended to solve OCCSPs.

To improve the uniformity and readability of this thesis, every algorithm that appears in this thesis is presented using a common generate and test framework and pseudocode, which is described in detail in Section 2.2.2.

2.2.1 Constraint Satisfaction Problem (CSP)

Constraint satisfaction problems (CSP) are a simple yet powerful formalism used extensively to solve combinatorial problems. Much effort has gone into developing fast search algorithms to solve CSPs. A few such algorithms are chronological backtracking, dynamic backtracking, forward checking, and arc-consistency [33, 13, 20, 14]. CSPs have proven useful in a variety of application areas such as diagnosis, planning and scheduling, product configuration, and design [51, 7, 8, 43]. The definition of a CSP is provided below in Definition 2.1. An assignment is a solution to a CSP if it assigns a value to every variable in the CSP and is also consistent with the problem's constraints, C_C. In practice, a CSP is solved by repeatedly extending a partial assignment of variables to values and by checking that partial assignment against the problem's constraints. Definitions of *partial assignment* and *CSP solution* are provided next in Definitions 2.2 and 2.3, respectively.

Definition 2.1 **- CSP**

A CSP is a tuple $\langle I, V, C_C \rangle$, where $I = \{i_1, i_2, \ldots, n\}$ is a set of variables each with a nonempty domain $v = \{v_1, v_2, \ldots, v_m\}$ of possible values. $C_C = \{C_1, C_2, ..., C_k\}$ is a set of constraints such that each constraint C_i involves some subset of the variables and specifies allowable combinations of values for that subset. An assignment of values to variables $\{i_i = v_i, i_j = v_j, \ldots\}$ is a full assignment if it assigns a value to every variable in I, and is a partial assignment if it assigns a value to only a subset of the variables. For a partial assignment P, we will denote by $\hat{P}$ the set of variables assigned values by P.

Definition 2.2 **– Solution to a CSP**

A solution, I^*, to a CSP is a full assignment to I that satisfies each constraint $c \in C_C$.

2.2.2 Generate and Test Framework

Each search algorithm described in this thesis uses a similar *generate and test* procedure to construct solutions. Therefore, to enhance uniformity, we use the same high-level pseudocode to describe each algorithm. The pseudocode used in this thesis was originally used by Ginsberg in [13] to describe Chronological Backtracking, Backjumping, Conflict-directed Backjumping, and Dynamic Backtracking, all within a uniform setting. In Ginsberg's generate and test framework, partial assignments are *generated* by assigning values to unassigned variables, and then *tested* for consistency against the problem's constraints. If a partial assignment is inconsistent with the problem's constraints, then a subset of that partial assignment which gives rise to the inconsistency, called a *conflict*, is returned as the cause of the problem. The definition of conflict is summarized in Definition 2.3.

Definition 2.3 **- Conflict**
A partial assignment, P , is a *conflict* if and only if there is no CSP solution containing P . Conflicts are often and interchangeably called *nogoods.*

Next, we provide two additional definitions which allow us to present Chronological Backtracking and Dynamic Backtracking within a uniform setting.

Definition 2.4 **– Eliminating Explanation**
Given a partial assignment P to a CSP, an *eliminating explanation* for a variable i is a pair $\left(i \neq v, \hat{P}\right)$.

The intended meaning of Definition 2.4, an *eliminating explanation*, is that i cannot take the value v because of the values already assigned to P . An eliminating explanation can be viewed as a conflict written in a directed form. For example, if the partial assignment $\{i_1 = v_1, i_2 = v_2, i_3 = v_3\}$ is a conflict, it can be written in the directed form, $\{i_1 = v_1, i_2 = v_2\} \rightarrow i_3 \neq v_3$, and corresponds to the eliminating explanation $\left(i_3 \neq v_3, \{i_1, i_2\}\right)$, where $\hat{P} = \{i_1, i_2\}$.

Definition 2.5 **– Elimination Mechanism,** $\varepsilon(P,i)$
An *elimination mechanism* $\varepsilon(P,i)$ for a CSP is a function that accepts as arguments a partial assignment, P , and a variable $i \notin \hat{P}$. The function returns a (possibly empty) set $E_i = \varepsilon(P,i)$ of eliminating explanations for i . An elimination mechanism tries to extend a partial solution, P , by assigning each possible value v for a variable i , and returns a

reason for each value assignment that isn't consistent with *P*. For a set E_i of eliminating explanations, $\hat{E}_i$ denotes all values that have been identified as eliminated for *i* while ignoring the variables $\hat{P}$ that explain the elimination. For example, $\hat{\varepsilon}(P,i)$ returns just the values eliminated by $\varepsilon(P,i)$, and $\hat{E}_i = \hat{\varepsilon}(P,i)$.

In the next two sections, we describe Chronological Backtracking and then Dynamic Backtracking.

2.2.3 Chronological Backtracking (BT)

Chronological backtracking (BT) ensures a complete and systematic search of a CSP, by always expanding a search node's children before expanding its siblings. The algorithm has a linear space complexity and an exponential time complexity in the number of variables. Although simple to implement, it often suffers from a condition called "thrashing". Thrashing occurs when the algorithm repeatedly explores large portions of the search space unnecessarily. This condition worsens as the problem space gets larger, and can cause BT to perform very poorly on some problems, called *extremely hard instances* (EHI's) [2]. In addition, BT's performance can vary greatly depending upon the order in which variables are assigned. The pseudocode for BT is presented next in Algorithm 2.1.

Algorithm 2.1 **- Chronological Backtracking Pseudocode (BT) [13]**

```
1. Take as input a CSP, $\langle I, V, C_C \rangle$. Set $P = \emptyset$. P is a partial solution to the CSP. Set
   $\hat{E}_i = \emptyset$ for each $i \in I$. $\hat{E}_i$ is the set of values that have been eliminated for variable i.

2. If $\hat{P} = I$, so that P assigns a value to every element in I, it is a solution to the original
   problem. Return it. Otherwise, select a variable $i \in I - \hat{P}$ Set $\hat{E}_i = \hat{\varepsilon}(P, i)$, the
   values that have been eliminated as possible choices for i.

3. Set $S = V_i - \hat{E}_i$, the set of remaining possible values for i. If S is nonempty, choose
   an element $v \in S$. Add $(i, v)$ to P, thereby setting i's value to v, and return to Step 2.

4. If S is empty, clear $\hat{E}_i$ and let $(j, v_j)$ be the last entry in P; if there is no such entry,
   return failure. Remove $(j, v_j)$ from P, add $v_j$ to $\hat{E}_j$, set i = j and return to Step 3.
```

2.2.4 Dynamic Backtracking (DB)

Dynamic backtracking (DB) [13] ensures a complete, systematic, and memory-bounded search of the state space, while leveraging conflicts to only generate candidate assignments that resolve all stored conflicts [41,10]. When DB encounters a dead-end, it utilizes a backjumping resolution step, Proposition 2.1, to backjump directly to the source of the inconsistency, thus reducing the "thrashing" behavior common to BT. In addition, DB dynamically reorders the partial solution when backjumping in order to preserve as much intermediate conflict information as possible. Search failure is indicated when the backjumping resolution step returns an empty conflict, indicating that all domain values for a variable are inconsistent with the problem's constraints. DB requires O(i^2v) space where *i* is the number of variables, and *v* is the largest domain size. The notation used in Proposition 2.1 is from [44]. The pseudocode for DB is given in Algorithm 2.2. For clarity, the differences between BT and DB are highlighted in grey.

Proposition 2.1 **- Backjumping Resolution Step**

Let i be a variable with domain, $v = \{v_1, v_2, \ldots, v_m\}$, and let $P_1, P_2, \ldots, P_m$ be partial assignments that do not include i. If,

$$P_1 \cup \{(i, v_1)\}, P_2 \cup \{(i, v_2)\}, \ldots, P_m \cup \{(i, v_m)\}$$

are all conflicts, then,

$$P_1 \cup P_2 \cup \ldots \cup P_m$$

is also a conflict.

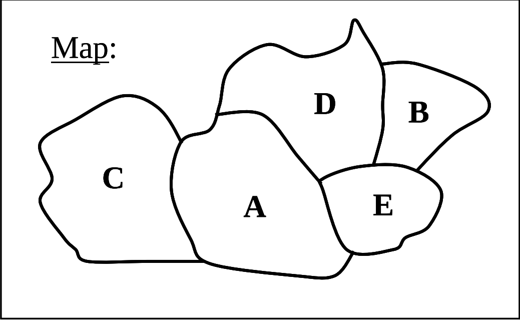
Algorithm 2.2 **- Dynamic Backtracking Pseudocode (DB) [13]**

```
1. Take as input a CSP, $\langle I, V, C_C \rangle$. Set $P = E_i = \emptyset$ for each $i \in I$.
2. If $\hat{P} = I$, return P. Otherwise, select a variable $i \in I - \hat{P}$. Set $E_i = E_i \cup \varepsilon(P, i)$.
3. Set $S = V_i - \hat{E}_i$. If S is nonempty, choose an element $v \in S$. Add $(i, v)$ to P and
   return to step 2.

4. If S is empty, we must have $\hat{E}_i = V_i$; let E be the set of all variables, $\hat{P}$,
   appearing in the explanations of each elimination explanation, $(i \neq v, \hat{P})$ for each
   $v \in \hat{E}_i$.

5. If $E = \emptyset$, return failure. Otherwise, let $(j, v_j)$ be the last entry in P such that
   $j \in E$. Remove $(j, v_j)$ from P and, for each variable k assigned a value after j,
   remove from $E_k$ any eliminating explanation that involves j. Set
       $E_j = E_j \cup \varepsilon(P, j) \cup \{(v_j, E \cap \hat{P})\}$
   so that $v_j$ is eliminated as a value for j because of the values taken by variables
   in $E \cap \hat{P}$. The inclusion of the term $\varepsilon(P, j)$ incorporates new information from
   variables that have been assigned values since the original assignment of $v_j$ to j.
   Now set i = j and return to step 3.
```

* The differences between Chronological Backtracking (BT) and Dynamic Backtracking (DB) are highlighted in grey.

To intuitively explain the DB algorithm, we use a simple map coloring example, which is paraphrased from [13]. We indicate the paraphrased text with indentations in both margins:

> Consider a map with five countries: Albania, Bulgaria, Czechoslovakia, Denmark and England. Each country must be assigned a color, subject to the constraint that bordering countries can't be assigned the same color.



Available Colors:
Red
Yellow
Blue

Figure 2.1: A Map Coloring Example

> Figure 2.1 illustrates shared borders, and each country is abbreviated with its starting letter. To start, we color Albania *red* and then Bulgaria *yellow*. Next, we consider Czechoslovakia. Czechoslovakia can't be colored *red,* since it borders Albania (which is already colored *red*), hence we eliminate *red* from Czechoslovakia's domain and record Albania as the reason. Instead, we color Czechoslovakia *blue*. DB keeps track of all relevant search information in an elimination table, as shown in Figure 2.2.

Region	Color	Red	Yellow	Blue
A	red			
B	yellow			
C	blue	A		
D				
E				

Figure 2.2: Recording the first Elimination Explanation

Now, we look at coloring Denmark. Denmark can't be colored *red,* because of Albania, or *yellow*, because of Bulgaria, hence we color it *blue*. But now we've reached a dead end. England cannot be colored any color because it borders Albania, Bulgaria, and Denmark. The updated elimination table is shown in Figure 2.3. To resolve this problem, DB merges each of England's elimination explanations into a new conflict, $\{A, B, D\}$, using the backjumping resolution step described in Proposition 2.1. Then, the most recently instantiated variable in the new conflict (Denmark) is unassigned, and any eliminating explanation involving that variable is removed. The updated elimination table is shown in Figure 2.4.

Region	Color	Red	Yellow	Blue
A	red			
B	yellow			
C	blue	A		
D	blue	A	B	
E		A	B	D

Figure 2.3: England's domain is exhausted.

Region	Color	Red	Yellow	Blue
A	red			
B	yellow			
C	blue	A		
D		A	B	
E		A	B	

Figure 2.4: Denmark's color assignment is retracted.

DB then turns the conflict, $\{A, B, D\}$, into an elimination explanation for Denmark, $(D \neq blue, \{A, B\})$, as described in Definition 2.5. This new elimination explanation is used to eliminate the value *blue* from Denmark's domain, and any elimination explanation involving Denmark is removed from the elimination database. The updated elimination table is shown in Figure 2.5.

Region	Color	Red	Yellow	Blue
A	red			
B				
C	blue	A		
D		A	B	A,B
E		A	B	

Figure 2.5: Bulgaria's color assignment is retracted.

Now DB has encountered another dead end, Denmark's domain has been exhausted. Therefore, DB employs the backjumping resolution step again, and merges each of Denmark's eliminating explanations together to construct a new conflict $\{A, B\}$. DB then eliminates *red* from Bulgaria's domain, $(B \neq red, \{A\})$, since Bulgaria was the most recently instantiated variable in the conflict $\{A, B\}$, via Definition 2.5. Bulgaria is subsequently assigned a new color, *yellow*, and the updated elimination table is shown in Figure 2.6. Note that DB is able to skip over changing Czechoslovakia's variable assignment even though it was instantiated after Bulgaria. Instead, DB simply switches the instantiation orders of Bulgaria and Czechoslovakia. This is referred to as *dynamic variable reordering*, and is the key innovation of DB.

Region	Color	Red	Yellow	Blue
A	red			
C	blue	A		
B			A	
D		A		
E		A		

Figure 2.6: Dynamic Variable Reordering (Bulgaria and Czechoslovakia).

The problem is now trivially solved by coloring Bulgaria *red*, Denmark *yellow*, and England *blue*. As Ginsberg [13] points out, "the thing to note is that when we changed the color for Bulgaria, we retained both the blue color for Czechoslovakia and the information indicating that none of Czechoslovakia, Denmark and England could be red. In more complex examples, this information may be very hard-won and retaining it may save us a great deal of subsequent search effort."

2.2.5 Optimal Conditional CSP (OCCSP)

Here we present a hybrid CSP formalism that combines the conditional constraint satisfaction problem (CCSP) and the optimal constraint satisfaction problem (OCSP). This hybrid, called an OCCSP, is presented below in Definition 2.6.

Definition 2.6 **- Optimal Conditional CSP (OCCSP)**

An OCCSP is formally defined as a 6-tuple $\langle I, V, C_C, I_I, C_A, f(P) \rangle$. Where,

- $I = \{i_1, i_2, \ldots, i_n\}$, is a set of all variables which may appear in the problem.
- $v = \{v_1, v_2, \ldots, v_n\}$, is afinite domain corresponding to each variable in *I*.
- C_C, represents the set of all constraints to be satisfied.
- $I_I \subseteq I$, is a set of non-empty initially active variables.
- C_A, is a set of activity constraints (Definition 2.7) describing when variables become active.
- $f(P)$, is a multi-attribute cost function, $f(P) = \sum_{(i,v)\in P} f_i(v)$, that sums together the costs of individual variable value assignments, $f_i(v \in V_i) \in \Re$ for every $i \in I$ that is assigned a value, $v \in v_i$, in the partial assignment P.

Definition 2.7 **– Activity Constraint**

An *activity constraint* is an expression of the form $AC \rightarrow active(i_k)$, where AC represents an assignment of values to variables, $\{i_1 = v_1, \ldots, i_j = v_j\}$, and is the condition under which variable i_k becomes *active*. [22]

When a variable becomes *active,* that variable must subsequently be assigned a value which is consistent with all other variable-value assignments. If all of an activity constraint's conditions, AC, are met but the activated variable, i_k, *is not* assigned a value, then that activity constraint is said to be *unsatisfied.* If all of an activity constraint's conditions, AC, are met and the activated variable, i_k, *is* assigned a value, then that activity constraint is said to be *satisfied*.

Definition 2.8 **– Solution to an OCCSP.**

The optimal solution to an OCCSP, I^*, is defined as:

$$I^* = \arg\min(I) \; s.t. \; \forall \; c_c \in C_C \; is \; satisfied \; and \; \forall \; c_a \in C_A \; is \; satisfied \,.$$

The OCCSP formalism presented here has also been referred to as an activity-based Dynamic Preference CSP (aDPCSP) [21,17]. However, in this thesis we choose the naming convention Optimal Conditional CSP to reflect the conditionally active nature the variables [35], and to reflect the optimization being performed. In [17], the A* search algorithm and an order-of-magnitude preference logic were extended in order to solve OCCSPs more efficiently. In Chapter 4, we extend Dynamic Backtracking to solve OCCSPs with the same motivation, to improve the speed at which OCCSPs can be solved.

A Simple Example of an OCCSP:

To give a simple example of an OCCSP, we introduce a simple car configuration task commonly employed in the CCSP literature [22]. In this example, the car buyer's objective is to minimize the cost of the vehicle subject to the car dealer's configuration requirements, shown in Figure 2.7. The car buyer must choose from *Base Package* (B) one of three values {*luxury, standard, convertible*}. Choosing *luxury* activates the options *Air-Conditioning* (A) and *Sunroof* (S), while choosing *standard* activates no additional options, and choosing *Convertible* (C) activates the options *Hardtop* (H) and *Ragtop* (R). Each base package and option has an associated cost. Each option has a corresponding activity constraint:

$$C_A = \{B = 1 \rightarrow active(A), \quad B = 1 \rightarrow active(S), \quad B = 3 \rightarrow active(R), \quad B = 3 \rightarrow active(H)\}.$$

In addition, the car buyer does not want a sunroof, so there are two compatibility constraints: $C_C = \{S \neq 1 \wedge S \neq 2\}$. The preferences, which are costs associated with variable-value assignments, in this example are:

$$C_S = \{\{B = 1\} \rightarrow 9, \{B = 2\} \rightarrow 10, \{B = 3\} \rightarrow 9, \{A = 2\} \rightarrow 2, \{S = 1\} \rightarrow 3, \{S = 2\} \rightarrow 2, \{H = 2\} \rightarrow 2,$$
$$\{R = 1\} \rightarrow 3, \{R = 2\} \rightarrow 2\}.$$

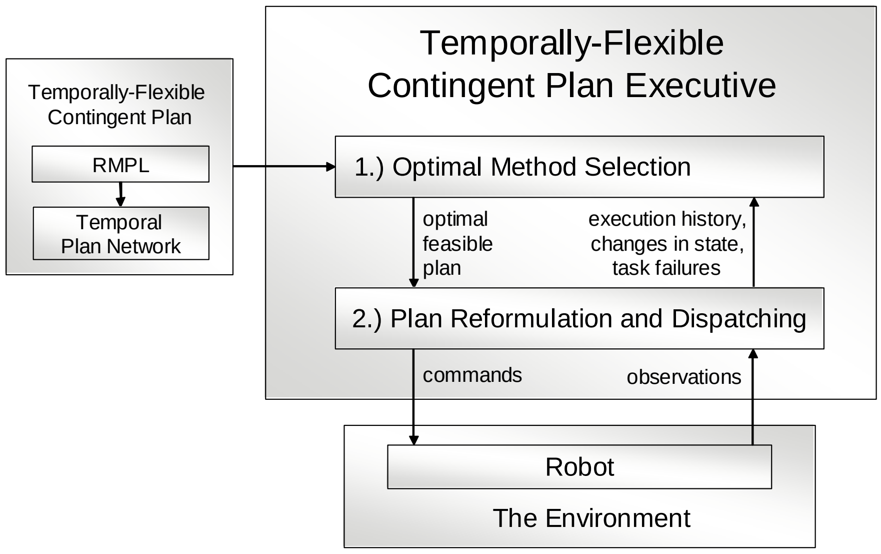
Variable	Values	Activates:	Cost $
(B) base package	1.) luxury	airConditioning, sunroof	$9 K
	2.) standard	-	$10 K
	3.) convertible	ragtop, hardtop	$9 K
(A) Air-Conditioning	1.) no	-	$0 K
	2.) yes	-	$2 K
(S) sunroof	1.) tint	-	$3 K
	2.) no tint	-	$2 K
(H) hardtop	1.) no	-	$0 K
	2.) yes	-	$2 K
(R) ragtop	1.) automatic	-	$3 K
	2.) manual	-	$2 K

Figure 2.7: An example OCCSP, a car buyer's configuration task.

Next, in Chapter 3, we describe Temporally-Flexible Contingent Planning.

Chapter 3 – Temporally-Flexible Contingent Planning

The Model-based Embedded and Robotic Systems Group at MIT has designed a temporally-flexible contingent plan executive, called Kirk [18]. A diagram of Kirk's planning architecture, initially shown in Figure 1.4, is shown again for convenience in Figure 3.1. Kirk takes as input a temporally-flexible contingent plan, encoded in *the Reactive Model-based Programming Language* (RMPL) [15]. To enable fast online execution, Kirk translates an RMPL plan into a temporally-flexible plan graph called a Temporal Plan Network (TPN). Kirk then searches the TPN to determine an optimal set of methods for accomplishing the mission objectives, called optimal method selection. In addition, Kirk can adjust the plan as necessary, in response to changes in state and task failures during execution, to ensure that a robot always optimally achieves its mission objectives. In this chapter, we describe temporally-flexible contingent planning in more detail. The central focus of this thesis, developed in Chapters 4 through 6, is to improve the speed at which Kirk performs Step 1, optimal method selection.



Temporally-Flexible Contingent Plan
RMPL
Temporal Plan Network
Temporally-Flexible Contingent Plan Executive
1.) Optimal Method Selection
optimal feasible plan
execution history, changes in state, task failures
2.) Plan Reformulation and Dispatching
commands
observations
Robot
The Environment

Figure 3.1: Kirk: A Temporally-Flexible Contingent Plan Executive.

The remainder of this chapter is organized as follows. First, we describe the language in which Kirk understands its input plan, called the *Reactive Model-Based Programming Language* (RMPL). Then, we describe Kirk's temporally-flexible plan specification, called a *Temporal Plan Network* (TPN), which enables fast online execution of RMPL programs. Finally, we describe Kirk's underlying planning algorithms, which perform optimal method selection, and plan reformulation and dispatching.

3.1 Reactive Model-based Programming Language (RMPL)

To encode temporally-flexible contingent plans compactly, RMPL supports many intuitive language constructs, such as: parameterized commands with cost, state assignments, temporally-flexible duration constraints, sequential and parallel composition, non-deterministic choice, conditional and pre-emptive execution, and maintenance constraints. Each construct is shown, respectively, in Figure 3.2, and is described briefly in the following paragraph. A more thorough description of these constructs is available in [50].

```
A := a( p1, p2, … )=c |
     s |
     A [lb,ub] |
     sequence ( A, A′, … ) |
     parallel ( A, A′, … ) |
     choose ( A, A′, … ) |
     if s then A else A′ |
     when s then A |
     do A maintaining s |
     do A watching s

a := primitive activity or command
p := activity parameter
c := cost associated with activity
s := assignment to state variable
     of the form (xi = vi)
```

Figure 3.2: RMPL Constructs Supported by the Kirk Planner.

The RMPL language supports two primitive constructs, commands and state assignments. A command, **a(p_1, p_2, …)=c**, is a function call that the robot can directly interpret and execute. Each ***p*** is a function parameter passed to the robot with the function call, and ***c*** is the cost to be incurred by executing that activity. A state assignment, **s**, is an assignment to a state variable, $x_i = v_i$, where x_i is a state variable and v_i is an element of x_i's domain. Kirk uses state assignments for two purposes, state assertions and state queries. A state assertion declares that a state variable will hold a particular value from its domain for a specified duration of time. A state query is used in conjunction with maintenance constraints to ensure that a certain state constraint is satisfied. The upper case letter **A** denotes well-formed RMPL expressions. **A[lb,ub]** is the basic construct for expressing flexible timing requirements, and implies that **A** *must not* finish executing before **lb** time units, and *must* finish executing before **ub** time units.

```
LRV-deployment-sequence() [5,20] = {
 sequence(
   R1.remove(insulation-blanket)=1 [1,3],
   R1.remove(operating-tapes)=1 [1,3]
   parallel(
     R2.pull(reel)=3 [1,5],
     reel = in-tension [1,5],
     do(
         sequence(
           R1.deploy(aft-wheels)=2 [0.5,2],
           R1.deploy(front-wheels)=2 [0.5,2])
     )maintaining ( braked-reel = in-tension )
   )
   parallel(
      choose(
        parallel(
          R1.unfold(seats)=2 [1,5],
          R1-status = unfolding-seats [1,5] )
        parallel(
          R1.unfold(footrests)=3 [1,5],
          R1-status = unfolding-footrests [1,5])
      )
      if( R1-status = unfolding-seats )
        R2.unfold(footrests)=2 [1,5],
      else(
        R2.unfold(seats)=3 [1,5]
      )
   )
 )
} [5,20]
```

Figure 3.3: LRV Deployment Example encoded in RMPL

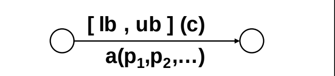
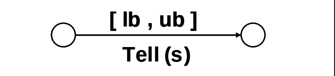
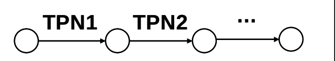
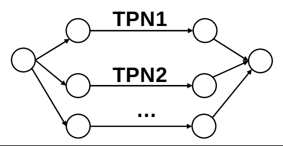
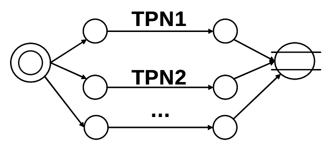
The **parallel** and **sequence** constructs are for concurrent and sequential tasks, and **choose** is used to express multiple strategies and contingencies. **If (c) then (A) else (A')** provides the construct for conditional execution, and **when(c) then (A)** provides the construct for reactive execution. **Do (A) maintaining (c)** and **do (A) watching (c)** act as maintenance conditions. Maintenance conditions monitor a state assignment, c, such as {power = on}, for the duration of an enclosed sub-plan, A, and halts execution of the sub-plan if c is violated.

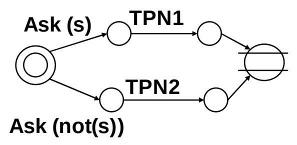
As an example, we provide an RMPL encoding of the LRV deployment scenario in Figure 3.3. Each command of the form *R1.command( )* is to be performed by robot #1, and each command of the form *R2.command( )* is to be performed by robot #2. The sequence and parallel constructs provide the basic building blocks to piece together the network of commands, the do maintaining construct ensures simultaneous lowering and deployment of the LRV, and the if else construct allows for conditional execution, for example, robot #2 will unfold the footrests if robot #1 is unfolding the seats, and vice versa. In addition, a cost is associated with each command, for example, it costs more for robot #1 to unfold the footrests than for robot #2 to unfold the footrests.

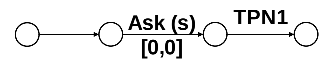
3.2 Temporal Plan Network (TPN)

To support fast online planning, Kirk converts an RMPL program into a temporally-flexible plan graph, called a *Temporal Plan Network* (TPN) [52]. A TPN represents all possible executions of an RMPL program over a finite window. Like RMPL, a TPN supports all of the language constructs listed in Figure 3.2. Figure 3.4 presents the mapping from RMPL to TPN. There are several aspects of the mapping from RMPL to TPN worth noting: state assertions are qualified with the statement Tell($x_i = v_i$), state queries are qualified with the statement Ask($x_i = v_i$), and all arcs with no time bounds in a TPN are assumed to have [0,0] time bounds.

To give an example of how an RMPL program is represented as a TPN, the LRV deployment example is converted into a TPN in Figure 3.5. The corresponding RMPL specification is in Figure 3.3.

$a(p_1,p_2,\ldots)=c$ [lb , ub]	
s [lb , ub]	
sequence (TPN1 , TPN2 , ...)	
parallel (TPN1 , TPN2 , ...)	
choose (TPN1 , TPN2 , ...)	

if (s) then (TPN1)
else (TPN2)
	

when (s) then (TPN1)	

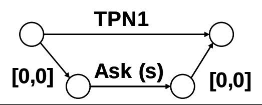
do (TPN1) maintaining (s)
do (TPN1) watching (s)
	

Figure 3.4: Mapping from RMPL to TPN.

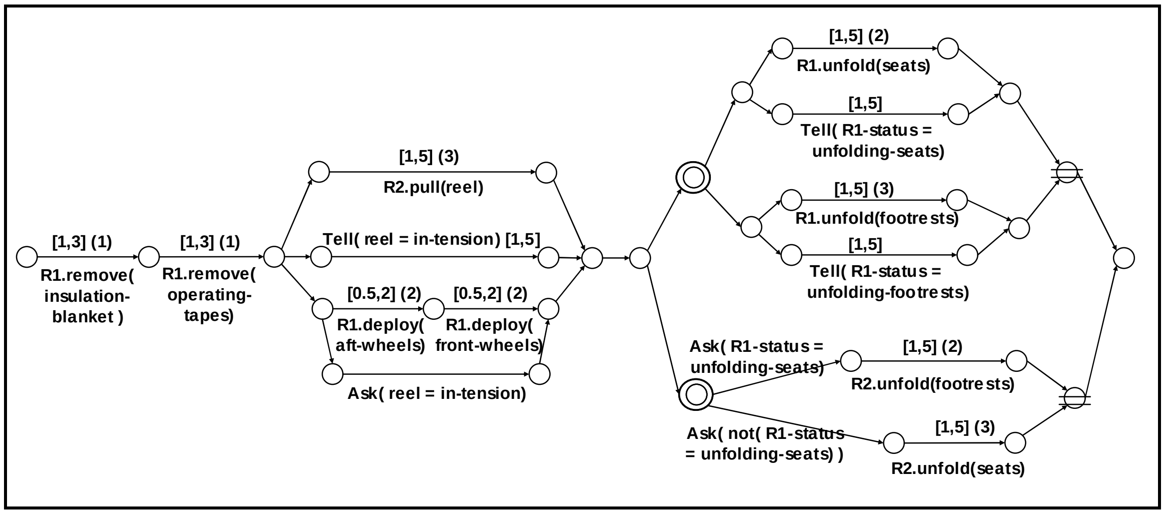


Figure 3.5: TPN encoding of the LRV deployment example.

3.3 Optimal Method Selection

To improve robustness, RMPL allows a mission programmer to specify redundant methods for accomplishing mission objectives [19], preferences between methods [45], and maintenance constraints on when methods are applicable [19]. Because of these features, an RMPL program is a non-deterministic program which represents many possible alternatives, called *candidate executions*, for achieving the mission objectives. Not all candidate executions are guaranteed to optimally achieve the mission objectives, however. Some candidate executions will violate timing constraints or maintenance constraints, while still others will satisfy the constraints, but do so sub-optimally. In Kirk, the process of finding an *optimal, consistent, and complete execution* of an RMPL program is called *optimal method selection* and occurs in four phases, as depicted in Figure 3.6.

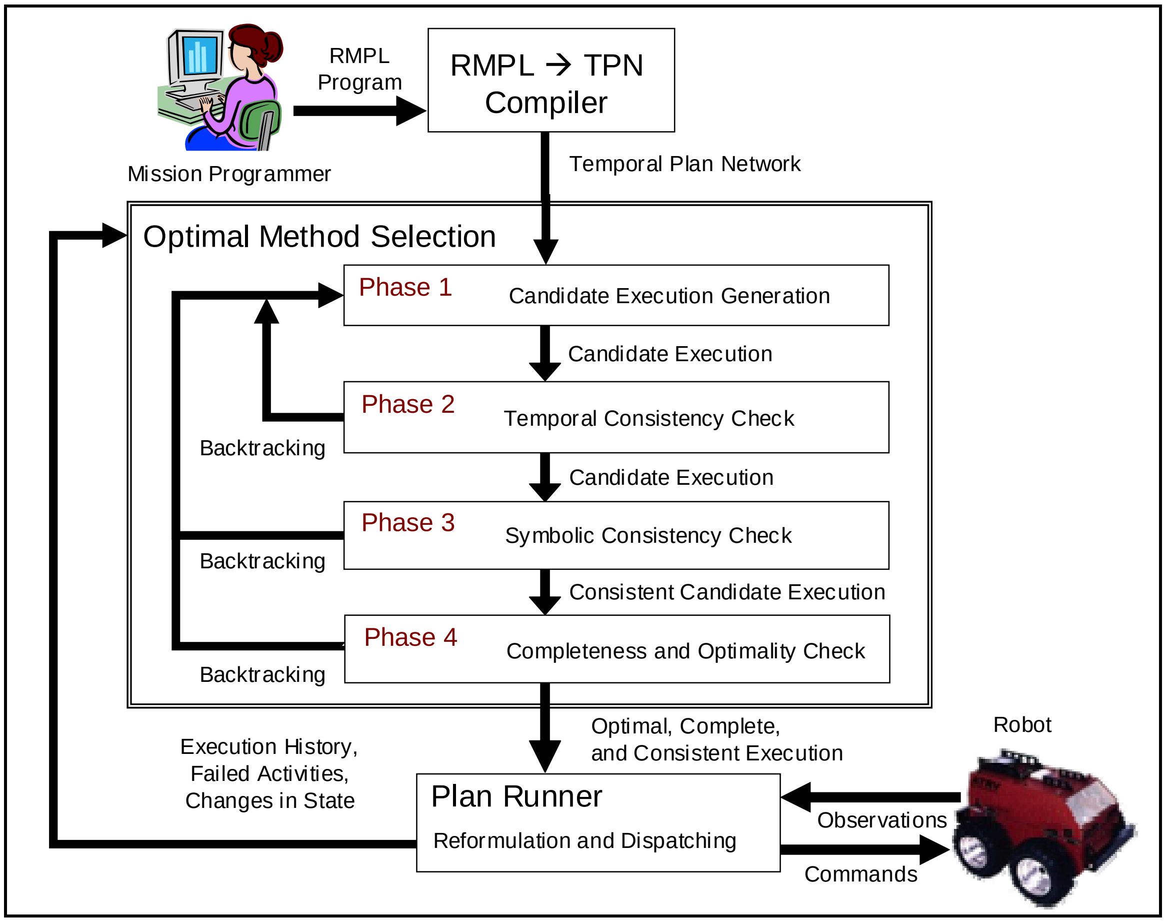

RMPL Program
RMPL → TPN Compiler
Mission Programmer
Temporal Plan Network
Optimal Method Selection
Phase 1
Candidate Execution Generation
Candidate Execution
Phase 2
Temporal Consistency Check
Backtracking
Candidate Execution
Phase 3
Symbolic Consistency Check
Backtracking
Consistent Candidate Execution
Phase 4
Completeness and Optimality Check
Backtracking
Execution History, Failed Activities, Changes in State
Optimal, Complete, and Consistent Execution
Robot
Plan Runner
Reformulation and Dispatching
Observations
Commands

Figure 3.6: Optimal Method Selection.

In order to perform optimal method selection efficiently via graph-based algorithms, Kirk first compiles an RMPL program into a TPN, as described in Section 3.2. Then, Kirk employs a generate-and-test cycle in order find an optimal, complete, and consistent execution. Kirk generates candidate executions in Phase 1 by making choices between redundant methods in the TPN. Then, Kirk tests the candidate executions for consistency, completeness, and optimality in Phases 2 through 4. A candidate execution is consistent if all temporal and maintenance constraints along its selected threads of execution are satisfiable. A candidate execution is complete if it makes a choice for every choice point reached by a selected thread of execution, starting from the beginning of the TPN. An optimal execution is an execution with the lowest total cost. If a candidate execution passes through Phases 2 through 4, then it is an optimal, complete, and consistent execution, and is sent to the plan runner for execution. Otherwise, it is eliminated, and Kirk backtracks to Phase 1 to generate another candidate execution. Next, we describe each of the four phases of optimal method selection in more detail.

Phase 1: Candidate Execution Generation

The process of generating candidate executions corresponds to choosing between redundant methods in an RMPL program. For example, consider the LRV example presented in Figures 3.3 and 3.5. There are 2 choices in the plan with 2 alternatives each, so there are a total of 2^(2) = 4 candidate executions of this RMPL program, each of which is shown in Figure 3.7.

To perform optimal method selection quickly, Kirk's candidate generation algorithm must be capable of ruling out large sets of candidate executions quickly in search of an optimal, complete, and consistent execution. In addition, to *ensure* that Kirk finds an optimal, complete, and consistent execution, if one exists, the algorithm used to generate candidate executions must satisfy two important properties; soundness and completeness.

Soundness ensures that an algorithm does not generate false positives; meaning it never falsely reports that an execution is optimal, complete, or consistent when it really is not. Completeness ensures that an algorithm will eventually find an optimal, complete, and consistent execution, if one exists, by forcing the algorithm to consider all combinations of methods that are not provably suboptimal or inconsistent.

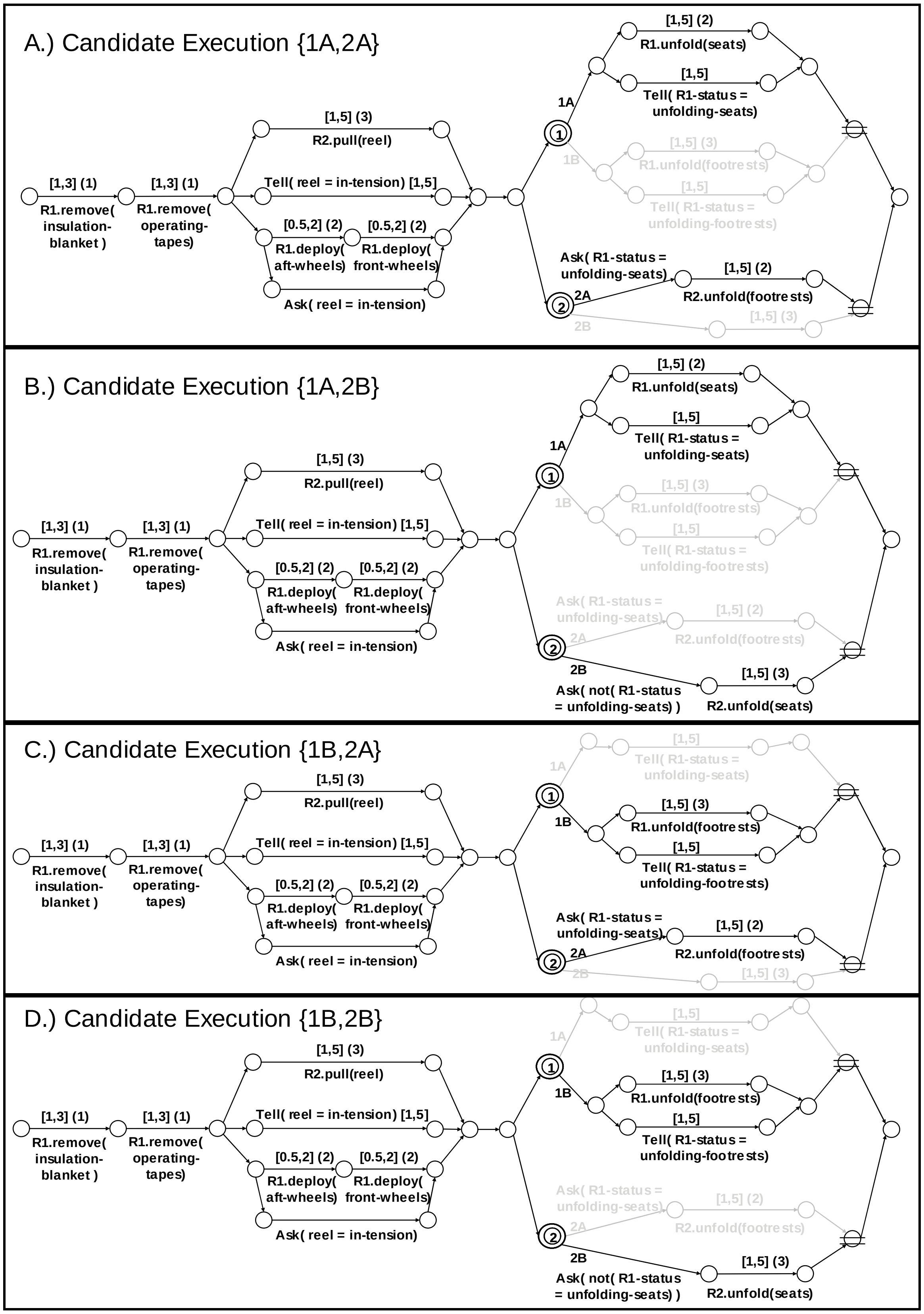

A.) Candidate Execution {1A,2A}
B.) Candidate Execution {1A,2B}
C.) Candidate Execution {1B,2A}
D.) Candidate Execution {1B,2B}
[1,3] (1)
R1.remove(insulation-blanket)
R1.remove(operating-tapes)
[1,5] (3)
R2.pull(reel)
Tell(reel = in-tension) [1,5]
[0.5,2] (2)
R1.deploy(aft-wheels)
R1.deploy(front-wheels)
Ask(reel = in-tension)
1A
1B
2A
2B
[1,5] (2)
R1.unfold(seats)
[1,5]
Tell(R1-status = unfolding-seats)
[1,5] (3)
R1.unfold(footrests)
Tell(R1-status = unfolding-footrests)
Ask(R1-status = unfolding-seats)
R2.unfold(footrests)
Ask(not(R1-status = unfolding-seats))
R2.unfold(seats)

Figure 3.7: Four Candidate Executions for the LRV Deployment Scenario.

In previous work, two sound and complete algorithms have been used to generate candidate executions in Kirk: Chronological Backtracking [19,48] and A* search [45]. Chronological Backtracking, as discussed in Section 2.2.3, is a simple algorithm to implement that requires very little runtime memory. However, it often suffers from a condition called "thrashing", making it an inefficient algorithm for generating candidate executions. In addition, Chronological Backtracking is incapable of considering preferences, so while it is sound with respect to completeness and consistency, it is not sound with respect to optimality. A* addresses the drawbacks of Chronological Backtracking by ensuring soundness with respect to optimality, and by incorporating a heuristic which decreases the number of candidate executions considered. These improvements come at a price, however. A* search requires exponential run-time memory in the worst-case.

To address the memory problem of A* search, in Chapter 4 we introduce a novel algorithm, called Conditional Dynamic Backtracking Branch-and-Bound, which requires a much smaller amount of runtime memory (worst-case polynomial) while retaining the performance efficiency and soundness with respect to optimality of A*. Next, we describe Phase 2, temporal consistency checking.

Phase 2: Temporal Consistency Checking

A candidate execution is a TPN with choices selected, as shown in Figure 3.7. Determining temporal consistency of a candidate execution corresponds to proving that an allowable time exists in which to execute each time event in the candidate execution. Because a TPN is built upon a framework of simple temporal constraints, this computation can be performed in polynomial time [7]. A candidate execution with all information omitted except for the timing constraints is a simple temporal network (STN), for which fast shortest path algorithms have been developed to determine temporal consistency. An inconsistent STN causes the shortest path algorithm to loop continuously, creating a negative cycle, which can be detected quickly by looking for self-loops in the set of support [4].

Kirk performs an incremental form of temporal consistency checking with an algorithm called ITC [39]. ITC supports fast temporal consistency testing through an incremental set of support [4], and through a novel set of incremental update rules [39]. ITC's update rules can both update a consistent STN *and* repair an inconsistent STN. In addition, ITC can extract the underlying cause of temporal inconsistency in a candidate plan, which is called a conflict. ITC's conflict extraction mechanism is instrumental to the enhancements developed in this thesis, and is described in Section 6.1.1. Next, we describe Phase 3, symbolic consistency checking.

Phase 3: Symbolic Consistency Checking

Kirk supports two types of state assignments, state assertions and state requests. State assertions allow a mission programmer to declare that a state assignment will hold for a specified duration of time in a plan, while state requests allow a mission programmer to request that a state assignment hold for a specified duration in a plan. These features enable Kirk to:

1.) Avoid selecting combinations of methods that entail a conflicting state.
2.) Avoid selecting methods whose maintenance conditions are not satisfied.

In Phase 3, any candidate execution that violates either of these two conditions is called *symbolically inconsistent* and is eliminated. A candidate execution that satisfies these two conditions is called a *consistent candidate execution* and is passed on to Phase 4 to be tested for optimality and completeness. Next, we describe in detail the two conditions that are checked during symbolic consistency checking.

1.) Detecting State Assertion Conflicts:

Consider the two concurrent threads of execution in Figure 3.8. If these two threads execute simultaneously, they will assert (erroneously) that the power is both on and off at the same time. One of these two state assertions will necessarily be violated at execution time, potentially causing a plan failure. A state assertion conflict occurs when two different values are assigned to the same state variable at a single instant in time. To avoid selecting combinations of methods that entail a state assertion conflict, Kirk first identifies all pairs of conflicting state assertions which could potentially co-occur during

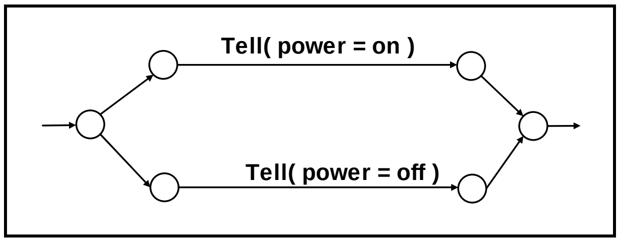


Figure 3.8: Conflicting state assertions.

execution, such as the ones in Figure 3.8. Then, Kirk resolves any threat of co-occurrence by forcing one to occur strictly before or after the other. For example, in Figure 3.9, Kirk considers two possibilities; inserting a timing constraint to ensure that *power = on* occurs before *power = off*, and inserting a timing constraint to ensure that *power = on* occurs after *power = off*. These two options are depicted in Figure 3.9.

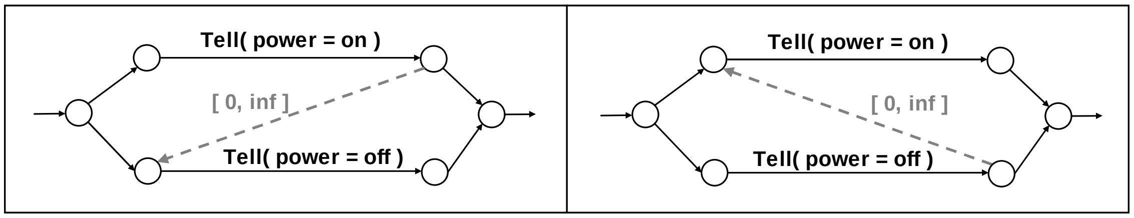


Figure 3.9: Two possible orderings to resolve conflicting state assertions.

2.) Detecting methods whose maintenance conditions are not satisfied:

Maintenance conditions correspond to state requests that ask a state assignment to hold for a specified duration of time in a plan. To ensure that state requests are satisfied, Kirk attempts to pair each state request with a state assertion that satisfies the request. For example, in Figure 3.10 the state request *Ask(reel = in-tension)* is paired with the state assertion *Tell(reel = in-tension)* by adding timing constraints to the plan which ensure that the state assertion holds for the entire duration of the state request. This is accomplished via two timing constraints, as indicated in Figure 3.10 with dashed grey arrows, which ensure that the state assertion starts before the state request starts and ends after the state request ends. In order to consider all candidate executions, Kirk must consider all consistent pairings of asks and tells for each candidate. Next we describe Phase 4, the completeness and optimality check.

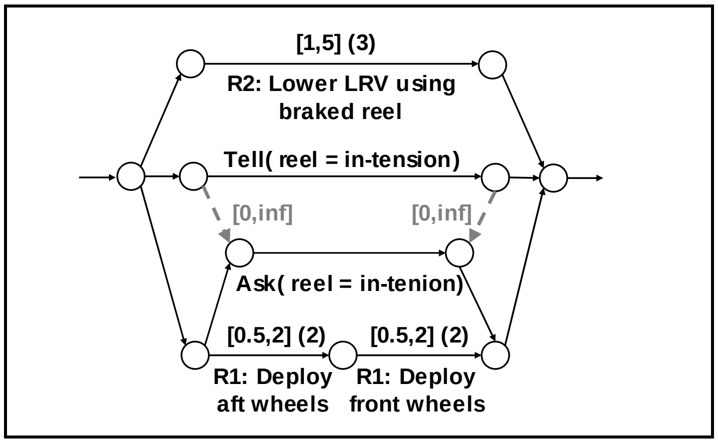


Figure 3.10: Ask Consistency Constraints

Phase 4: Completeness and Optimality Check

A candidate execution is complete if it makes a choice for every choice point reached by a selected thread of execution, starting from the beginning of the TPN.

To test a candidate execution for completeness, Kirk simply starts at the beginning of the TPN and follows the selected threads of execution to make sure that a choice is selected at every choice point. If not, then the candidate is passed back to Phase 1 for further method selection.

To test a candidate for optimality, Kirk keeps in memory the lowest cost complete and consistent execution found so far, called the *incumbent*. If a new complete and consistent execution has a lower cost than the incumbent, then it is stored as the new incumbent, otherwise it is pruned based on suboptimality. When no more candidate executions exist with a lower cost than the incumbent, the incumbent is returned as an *optimal, complete, and consistent execution* of the RMPL program, and is passed to the plan runner for reformulation and dispatching. If no incumbent is found, then the RMPL program has no consistent execution, and execution of the program is not allowed.

3.4 Plan Reformulation and Dispatching

Because Kirk supports temporal flexibility during planning and execution, there are many feasible schedules for executing a plan. To make explicit all feasible schedules, the allowable time bounds between activities needs to be calculated. This can accomplished using Floyd-Warshalls All-Pair Shortest-Path algorithm or Johnson's algorithm [5], and is called plan reformulation. Once reformulated, a plan is then sent to the dispatcher which executes the temporally-flexible plan dynamically while adapting to timing uncertainties and uncertain activity durations at runtime. We refer the interested reader to [26] and [42] for more information on plan reformulation and dispatching.

Next, we frame optimal method selection as an Optimal Conditional CSP (OCCSP).

Chapter 4 – Framing Optimal Method Selection as an OCCSP

Research on constraint satisfaction problems has lead to many breakthroughs in our ability to understand, analyze, and solve combinatorial problems. These advances have taken the form of fast and sophisticated search algorithms [33,41,13,37,20,14], as well as in-depth complexity analyses to help differentiate between fundamentally easy and hard to solve CSP instances [10,2,7]. To leverage these advances into more expressive domains, such as conditional planning with preferences and design configuration, many dynamic and flexible CSP variants have emerged [21]. One such variant, the Optimal Conditional CSP (OCCSP), employs activity constraints and soft constraints to model both conditional dependencies and preferences within a unified framework [17,21].

In this chapter, we frame optimal method selection, specifically TPN search, as an OCCSP. The motivation behind this approach is two-fold: Firstly, a mapping between TPN search and an OCCSP enables us to use fast algorithms from constraint satisfaction to perform optimal method selection. Secondly, this mapping enables us to leverage future developments in the field of constraint satisfaction with no further development effort.

It is important to note that the concepts presented in this chapter were developed jointly with Jon Kennel, and build upon previous work by Andreas Wehowsky and Aisha Walcott. In [48], Wehowsky framed a TPN as an activity-based Dynamic CSP with activity constraints encoding the conditional dependencies between choices. In [45], Walcott investigates a technique to perform optimal search over a TPN which weighs preferences between redundant choices. In this chapter, we merge these two research ideas by framing optimal search over a TPN as finding a solution to an OCCSP.

A TPN represents a physical process, which consists of primitive activities that are composed together using RMPL expressions. In Section 4.1, we show how to encode a TPN and its primitive activities as an OCCSP. Then, in Section 4.2, we show how primitive activities and TPNs, which are encoded as OCCSPs, are composed together using RMPL expressions. In Section 4.3, we show how optimal method selection, in the form of TPN search, can be encoded as a process of dynamically adding in more variables and values to the OCCSP encoding of a TPN.

4.1 Encoding a TPN and its Primitive Activities as an OCCSP

A TPN is comprised of four types of primitive activities: commands, state assertions, state requests, and timing constraints. To begin this section, we develop a general definition of a primitive activity, depicted in Figure 4.1 and described in Definition 4.1, which encompasses each of the primitive activity types listed above. Then, in Definition 4.2, we present an OCCSP encoding of a TPN. Finally we show that a primitive activity, Definition 4.1, is a restricted instance of a TPN, Definition 4.2.

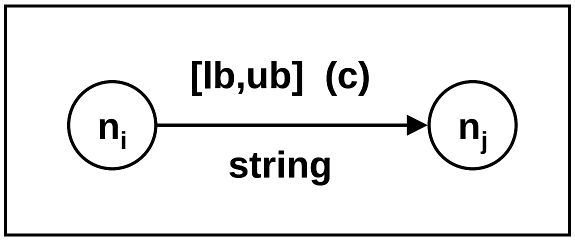


Figure 4.1: TPN Primitive Activity.

Definition 4.1 – TPN Primitive Activity

A *TPN primitive activity* is a 5-tuple $\langle string, n_i, n_j, [lb, ub], c \rangle$. Where,

- ***string*** represents a command, state assertion, state request, or timing constraint.
    - o Commands take the form *Robot.command(params).*
    - o State assertions take the form *TELL(var = val).*
    - o State queries take the form *ASK(var = val)* .
    - o Timing constraints have the empty string, $\varnothing$.
- ***n$_i$*** is a time event representing the start time of the activity.
- ***n$_j$*** is a time event representing the end time of the activity.
- **[*lb, ub*]** enforces a metric constraint, $lb \leq n_j - n_i \leq ub$, on the activity's duration

- $\boldsymbol{c} \in \Re$ is a real-valued cost associated with executing a command. Costs for state assertions, state requests, and timing constraints are all zero.

A TPN describes a physical process comprised of primitive activities, as follows:

Definition 4.2 – OCCSP Encoding of a TPN

An OCCSP encoding of a TPN is a 6-tuple $\langle I, V, i_A, C_A, f(P), (s,e) \rangle$. Where,

- $I = \{i_1, i_2, \ldots, i_n\}$, is a set of discrete choice variables.
- $v_i = \{A_{i1}, A_{i2}, \ldots, A_{im}\}$, is the domain for each $i \in I$, where a domain value, $A_{ik} = \{a_1, a_2, \ldots, a_j\}$, is a set of TPN primitive activities, as defined above.
- $i_A \in I$, a TPN always has one initially active variable, with one value, $v_A = \{A_A\}$.
- C_A, is a set of activity constraints describing when variables become active.
- $f(P)$, is a multi-attribute cost function that sums up the costs of all primitive activities selected with a candidate execution, *P*. This is accomplished by summing the costs of all primitive activities that belong to each variable-value assignment in the candidate execution *P*: $f(P) = \sum c_a \mid a \in A_m \wedge (i_k = A_{km}) \in P$.
- (s,e), represents the starting time event, *s*, and the end time event, *e*, of the TPN.

There are several aspects of Definition 4.2 worth clarifying:

1.) Discrete variables represent the disjunctive choices between redundant methods in a TPN and should not be confused with the TPN's time events. Every choice in an RMPL specification, and analogously its TPN, has a corresponding discrete choice variable in its OCCSP encoding, and every time event has a corresponding continuous variable (i.e. node) in the TPN.

2.) A TPN has exactly one initially active discrete variable, i_A, with exactly one value, v_A, which encodes all primitive activities between the TPN start and end events that are not nested within any choices, and, therefore, *must* be selected for execution.

3.) The activity constraints, C_A, encode the hierarchal dependencies between choices in a TPN, and are of the form $\{i_1 = v_1\} \rightarrow active(i_k)$, where $\{i_1 = v_1\}$ represents an assignment of one value to one variable, and is the *exact* condition under which variable i_k becomes *active*. Note that this is a restricted instance of the more general OCCSP definition of an activity constraint (Definition 2.7): $\{i_1 = v_1, \ldots, i_j = v_j\} \rightarrow active(i_k)$.

4.) This general definition allows a range of TPNs to be defined that don't correspond to an RMPL program, or even a contiguous process. In the next section, Section 4.2, we show how primitive activities and TPNs, which are encoded as OCCSPs, can be composed together using RMPL combinators to ensure that the resulting TPN corresponds to an actual RMPL program and represents a contiguous physical process. We call a TPN that is formed only from RMPL combinators a "well-formed" TPN.

Next, to give a concrete example of Definitions 4.1 and 4.2, we encode the TPN for the LRV deployment scenario as an OCCSP.

4.1.1 The LRV Deployment Scenario Encoded as an OCCSP

In this section we encode the LRV deployment scenario as an OCCSP. For convenience, the TPN is re-shown in Figure 4.2a. To begin, we label each TPN primitive and encode them using Definition 4.1, as depicted in Figure 4.2b and listed below. All primitive activities without time bounds specified are assumed to have [0,0] time bounds.

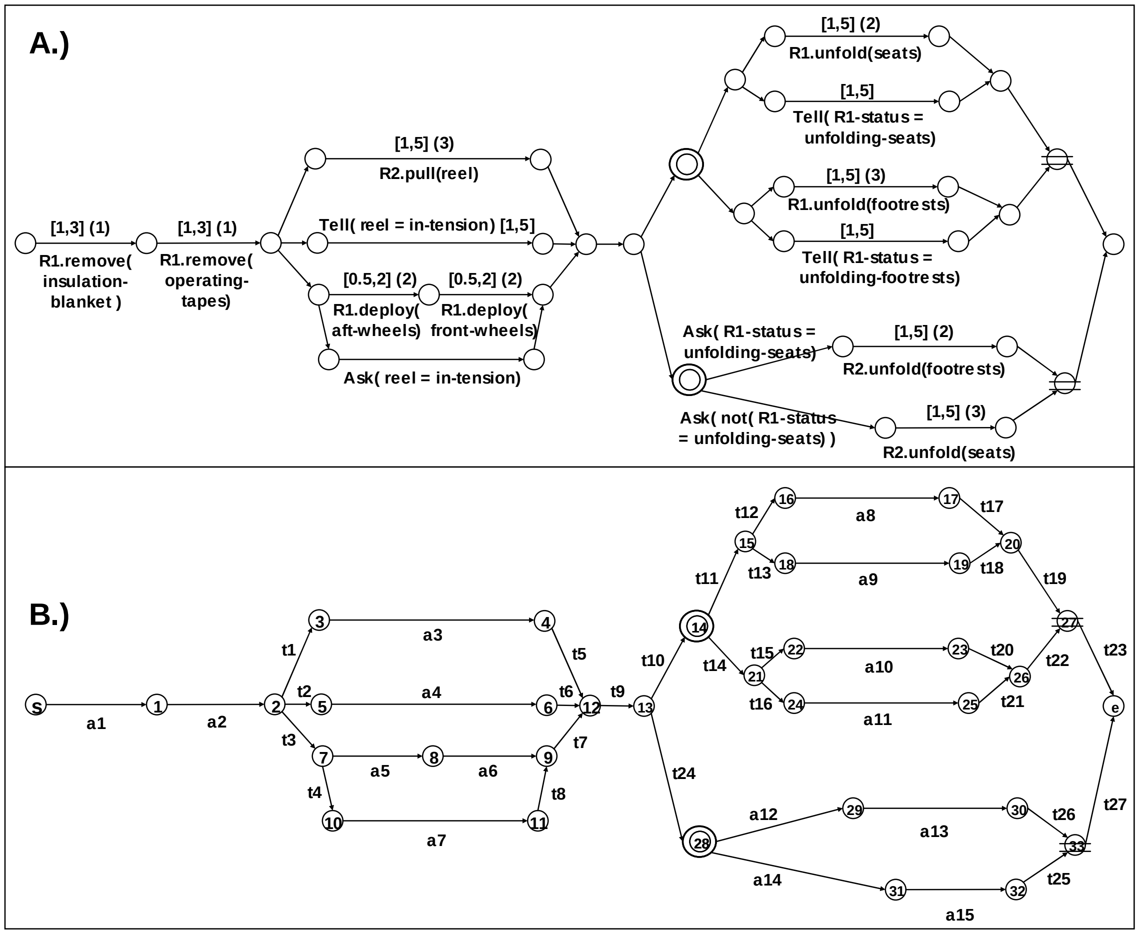


Figure 4.2: Labeling and encoding each primitive activity in the TPN.

TPN primitive activities:

$a1 = \langle \text{R1.remove}(\textit{insulation-blanket}), s, 1, [1,3], 1 \rangle$,

$a2 = \langle \text{R1.remove}(\textit{operating-tapes}), 1, 2, [1,3], 1 \rangle$,

$a3 = \langle \text{R2.pull}(\textit{reel}) 3, 4, [1,5], 3 \rangle$,

$a4 = \langle \text{Tell}(\ \textit{reel} \ = \ \textit{in-tension}) \ , 5, 6, [1,5], 0 \rangle$,

$a5 = \langle \text{R1.deploy}(\textit{aft-wheels}), 7, 8, [0.5,2], 2 \rangle$,

$a6 = \langle \text{R1.deploy}(\ \textit{front-wheels}), 8, 9, [0.5,2], 2 \rangle$,

$a7 = \langle \text{Ask}(\ \textit{reel} \ = \ \textit{in-tension}), 10, 11, [1,4], 0 \rangle$,

$a8 = \langle \text{R1. unfold}(\textit{seats}), 16, 17, [1,5], 3 \rangle$,

$a9 = \langle \text{Tell}(\ \textit{Rl-status} \ = \ \textit{unfolding-seats}), 18, 19, [1,5], 0 \rangle$,

$a10 = \langle \text{R1.unfold}(\ \textit{footrests}), 22, 23, [1,5], 3 \rangle$,

$a11 = \langle \text{Tell}(\ \textit{R}1\textit{-status} \ = \ \textit{unfolding-footrests}), 24, 25, [1,5], 0 \rangle$,

$a12 = \langle \text{Ask}(\ \textit{R}1\textit{-status} \ = \ \textit{unfolding-seats}), 28, 29, [0,0], 0 \rangle$,

$a13 = \langle \text{R2.unfold}(\ \textit{footrests}), 29, 30, [1,5], 2 \rangle$,

$a14 = \langle \text{Ask}(\ \text{not}(\textit{R}1\textit{-status} \ = \ \textit{unfolding-seats})), 27, 31, [0,0], 0 \rangle$,

$a15 = \langle \text{R2.unfold}(\textit{seats}), 31, 32, [1,5], 3 \rangle$

In addition, all of the timing constraints, t1 through t27, have [0,0] time bounds:
t1 through t27 $= \langle \varnothing, i, j, [0,0], 0 \rangle$.

Next, we identify the primitive activities that belong to the initially active variable-value assignment, $\{i_A = A_A\}$. Each primitive activity in the plan that does not reside within any nested choices belongs to the initially active variable-value assignment. This encompasses all primitive activities in the plan in Figure 3.2, except for ones along the conditional threads of execution between events 14 and 27, and 28 and 33. Therefore, the initially active variable assignment consists of these primitive activities:

$$\{\ i_A = A_A\} = \left\{ \begin{matrix} a1, a2, a3, a4, a5, a6, a7, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, \\ t23, t24, t27 \end{matrix} \right\}$$

There are two choices to make in the plan, so two discrete choice variables, i_1 and i_2, need to be defined. Variable i_1 has two possible values corresponding to the two possible threads of execution emanating from event 14 and converging at event 27. The domain for variable i_1 is defined as follows:

$$v_1 = \{A_{1a}, A_{1b}\} = \left\{ \begin{matrix} \{a8, a9, t11, t12, t13, t17, t18, t19\}, \\ \{a10, a11, t14, t15, t16, t20, t21, t22\} \end{matrix} \right\}$$

Similarly, variable i_2 has two possible values corresponding to the two possible threads of execution emanating from event 28 and converging at event 33. The domain for variable i_2 is defined as follows:

$$v_2 = \{A_{2a}, A_{2b}\} = \{\{\mathrm{a12, a13, t26}\}, \{\mathrm{a14, a15, t25}\}\}$$

Next, we need to define an activity constraint for each variable that isn't initially active. In this example, that corresponds to variables i_1 and i_2. A variable's activating condition is determined by the thread of execution upon which a choice resides. In RMPL, because choices are hierarchically composed, there will always be exactly one primitive activity preceding a choice start event, and one primitive activity succeeding a choice end event. For example, in Figure 3.2, exactly one primitive activity, t10, precedes the choice start event, event 14, and exactly one primitive activity, t23, succeeds the choice end event, event 27. Furthermore, the two primitive activities that precede and succeed a choice will always reside along the same thread of execution, and hence belong to the same variable-value assignment in the OCCSP. In this case, the preceding and succeeding primitive activities belong to the initially active variable-value assignment, $\{i_A = A_A\}$. Therefore, the activity constraint for variable i_1 is: $\{i_A = v_A\} \rightarrow active(i_1)$. This encodes the constraint that if variable i_1's parent thread, $\{i_A = A_A\}$, is selected for execution, then a thread of execution subsequently needs to be selected for variable i_1. The activity constraint for variable i_2 is similar: $\{i_A = v_A\} \rightarrow active(i_2)$.

Next we demonstrate how to calculate the cost function for this LRV scenario: $f(P) = \sum c_a \mid a \in A_m \wedge (i_k = A_m) \in P$. First, we demonstrate how this cost function allows us to calculate the cost of individual variable-value assignments. For example, next, we use the cost function to calculate the total cost of all primitive activities that belong to the three separate variable-value assignments, $P = \{i_A = v_A\}$, $P = \{i_1 = A_{1a}\}$, and $P = \{i_2 = A_{2a}\}$. Notice that all timing constraints, denoted t_i, have zero cost.

$$
\begin{aligned}
f(P) = f(i_A = A_A) = \sum c_a \,\big|\, a \in A_A \wedge \{i_A = A_A\} &= c_{a1} + c_{a2} + c_{a3} + c_{a4} + c_{a5} + c_{a6} + c_{a7} + t_i \\
&= 1 + 1 + 3 + 0 + 2 + 2 + 0 + 0 \\
&= 9
\end{aligned}
$$

$$
\begin{aligned}
f(P) = f(i_1 = A_{1a}) = \sum c_a \,\big|\, a \in A_{1a} \wedge \{i_1 = A_{1a}\} &= c_{a8} + c_{a9} + t_i \\
&= 2 + 0 + 0 \\
&= 2
\end{aligned}
$$

$$
\begin{aligned}
f(P) = f(i_2 = A_{2a}) = \sum c_a \,\big|\, a \in A_{2a} \wedge \{i_2 = A_{2a}\} &= c_{a12} + c_{a13} + t_i \\
&= 0 + 2 + 0 \\
&= 2
\end{aligned}
$$

In the LRV scenario, there are four possible candidate executions:
$\{(i_A, A_A),(i_1, A_{1a}),(i_2, A_{2a})\}$, $\{(i_A, A_A),(i_1, A_{1a}),(i_2, A_{2b})\}$,
$\{(i_A, A_A),(i_1, A_{1b}),(i_2, A_{2a})\}$, and $\{(i_A, A_A),(i_1, A_{1b}),(i_2, A_{2b})\}$, as depicted in Figure 3.7.

Next, we compute the total cost of one candidate execution: $\{(i_A, A_A),(i_1, A_{1a}),(i_2, A_{2a})\}$. Notice that this candidate execution, $P = \{(i_A, A_A),(i_1, A_{1a}),(i_2, A_{2a})\}$, is simply the sum of the three cost functions developed above:

$$
f(P) = f((i_A, A_A),(i_1, A_{1a}),(i_2, A_{2a})) = f(i_A, A_A) + f(i_1, A_{1a}) + f(i_2, A_{2a}) = 9 + 2 + 2 = 13.
$$

4.2 Constructing a TPN from RMPL Expressions

In this section, we explain how Kirk recursively constructs a TPN from RMPL expressions. Kirk constructs a TPN recursively by first constructing TPNs for each sub-expression of an RMPL expression. In the following sub-sections, we define how each RMPL expression composes a single TPN out of its component sub-TPNs.

4.2.1 TPN Primitive Activity

A TPN primitive activity (Figure 4.1), $\langle string, n_i, n_j, [lb, ub], c\rangle$, is encoded as a TPN, $\langle I, V, i_A, C_A, f(P), (s,e)\rangle$, such that:

- $I = \{i\},\; V = \{\{v_i\}\},\; i_A = \{i\}.$
- There is only one discrete variable *i*, which is initially active and has value v_i .
- $v_i = (\langle string, n_i, n_j, [lb, ub], c\rangle).$
- There are no activity constraints.
- The cost function trivially corresponds to $f(i = v_i) = c$.
- The start and end events of the TPN are those of the primitive activity, (n_i, n_j) .

Hence, the TPN for a primitive activity is a tuple: $\langle \{i\}, \{v_i\}, \{i\}, \{\varnothing\}, f(i = v_i) = c, (n_i, n_j)\rangle$ with v_i = $(\langle string, n_i, n_j, [lb, ub], c\rangle)$.

4.2.2 TPN Composition Operator

The fundamental operation performed by an RMPL expression is to compose two or more TPNs together to form a new TPN. In this sub-section, we define precisely this operation.

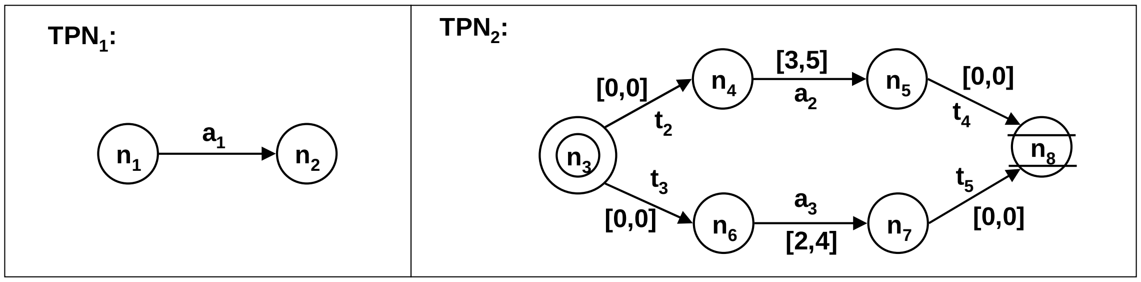


Figure 4.3: TPN$_1$ and TPN$_2$

To illustrate the concept of composing two TPNs into a new TPN, consider the two TPNs in Figure 4.3. Consider, for example, that we wish to compose TPN$_1$ and TPN$_2$

into a new TPN, called TPN$_{new}$, in which TPN$_2$ is executes immediately after TPN$_1$. This is accomplished, as shown in Figure 4.4, by placing an additional timing constraint with [0,0] time bounds, called t$_1$, between the end event of TPN$_1$, n$_2$, and the start event of TPN$_2$, n$_3$.

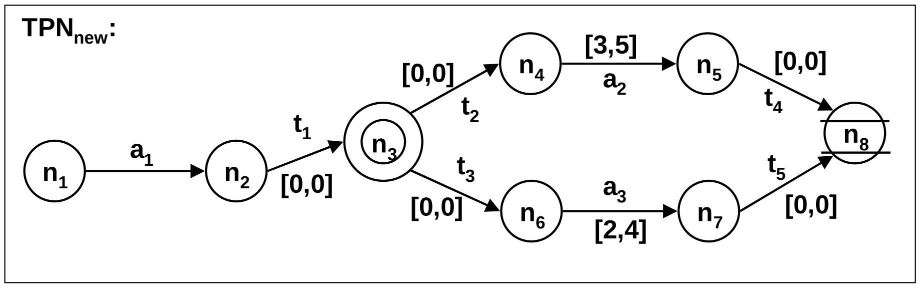


Figure 4.4: Composing TPN$_1$, TPN$_2$, and t$_1$ into a new TPN, called TPN$_{new}$.

The composition operator takes as input two or more TPNs, such as TPN$_1$ and TPN$_2$, and additional timing constraints, such as t$_1$, to compose the TPNs together. Then, it outputs a single TPN that composes each of the input TPNs together in accordance with the additional timing constraints, as shown in Figure 4.4.

In general, seven steps need to be performed to compose two or more TPNs, plus any additional timing constraints, into a new TPN:

Definition 4.3 – TPN Composition Operator

1.) Take as input two or more sub-TPNs, {TPN$_1$,TPN$_2$,…}, and any additional timing constraints, {t$_i$}, which are to be composed together into a new TPN, called TPN$_{new}$.

2.) Merge all initially active primitive activities from each TPN, as well as any additional timing constraints, into the initially active domain for TPN$_{new}$:
$TPN_{new}[(i_A, v_A)] = \{TPN_1[(i_A, v_A)] \cup TPN_2[(i_A, v_A)] \cup ... \cup \{t_i\}\}$

3.) Next, add all discrete variables, except i$_A$, from each TPN into TPN$_{new}$:

$$TPN_{new}[I - i_A] = \{TPN_1[I - i_A] \cup TPN_2[I - i_A] \cup ...\}$$

4.) Add in their respective domains as well:

$$TPN_{new}[v_i] = \{TPN_1[\exists \ v_i | \ v_i \neq v_A] \cup TPN_2[\exists \ v_i | \ v_i \neq v_A] \cup \ldots\}$$

5.) Inherit all activity constraints from each TPN:

$$TPN_{new}[C_A] = \{TPN_1[C_A] \cup TPN_2[C_A] \cup \ldots\}$$

6.) $f(\ )$, the cost function stays the same: $f(P) = \sum c_a | \ a \in A_m \wedge (i_k = A_m) \in P$.

7.) The start event, *s,* is the only time event with no incoming primitive activity. The end event, *e*, is the only event with no outgoing primitive activity. To be well-formed, a TPN must be composed such that there is only one start event, *s*, and one end event, *e*.

Next, in the following sub-sections we describe how each RMPL expression uses this composition operator, Definition 4.3, to compose well-formed TPNs.

4.2.3 Sequence

The *sequence* RMPL expression enables sequential execution of sub-expressions, as shown in Figure 4.5, by adding a timing constraint between two TPNs in sequence. For example, the additional constraint $a_1 = \langle \varnothing, n_2, n_3, [0,0], 0 \rangle$ forces events n_2 and n_3 to co-occur, thus ensuring that the execution of TPN_2 begins immediately after the execution of TPN_1 ends. While two TPNs are connected in sequence in Figure 4.5, in general, the sequence combinator can connect an arbitrary number of TPNs in sequence.

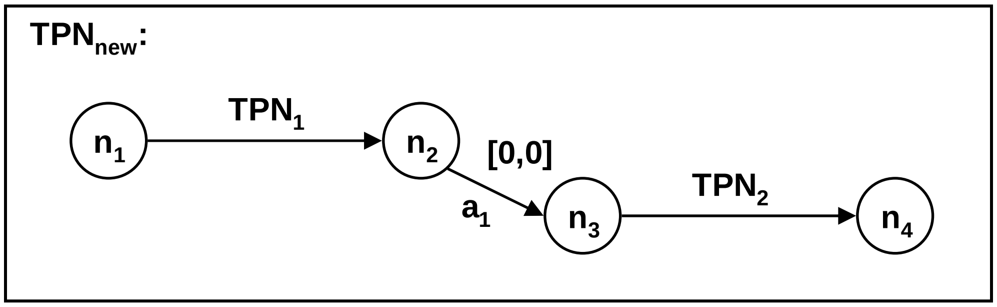


Figure 4.5: Sequence RMPL Expression.

To connect two TPNs in sequence, Kirk simply composes TPN_1, TPN_2, and the additional timing constraint, a_1, into a new TPN_{new} using the TPN composition operator (Definition 4.3).

4.2.4 Parallel

The *parallel* RMPL expression enables concurrent execution of RMPL sub-expressions; its corresponding TPN is depicted in Figure 4.6. While two TPNs are connected in parallel in Figure 4.6, in general, the parallel RMPL expression can connect an arbitrary number of TPNs in parallel. To ensure that two TPNs execute concurrently, Kirk adds additional timing constraints, $\{a_1,a_2,a_3,a_4\}$, which ensure that the two TPN's respective start and end events co-occur.

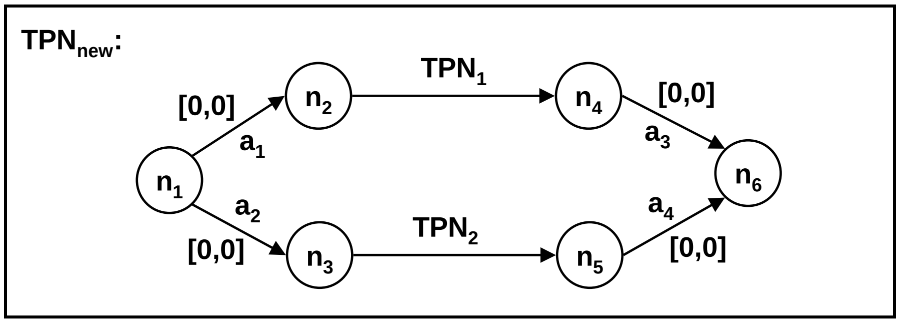


Figure 4.6: Parallel RMPL Expression.

4.2.5 Choose

The *choose* RMPL expression enables disjunctive choice between sub-expressions; its TPN is depicted in Figure 4.7. As with the parallel and sequence expressions, the choose expression allows an arbitrary number of TPNs to be composed together. The start event of a choice is indicated by a double circle, and the end event by a circle with two horizontal lines. Kirk selects only one thread of execution between a choice's start and end events. To encode a disjunctive choice between sub-expressions, Kirk introduces a new discrete variable into TPN_{new} with a domain value to represent each of the possible sub-expressions. In addition, Kirk adds additional timing constraints so that the redundant sub-expressions co-occur. For example, in Figure 4.7, to represent a new choice, Kirk creates a new TPN with six events, n_1 through n_6, four timing constraints, a_1 = $\langle \varnothing, n_1, n_2, [0,0], 0 \rangle$, a_2 = $\langle \varnothing, n_1, n_3, [0,0], 0 \rangle$, a_3 = $\langle \varnothing, n_4, n_6, [0,0], 0 \rangle$, and a_4 = $\langle \varnothing, n_5, n_6, [0,0], 0 \rangle$, one discrete variable with two discrete values representing TPN_1 and TPN_2, and an activity constraint to activate the new variable, $\{i_A = v_A\} \rightarrow active(i_k)$.

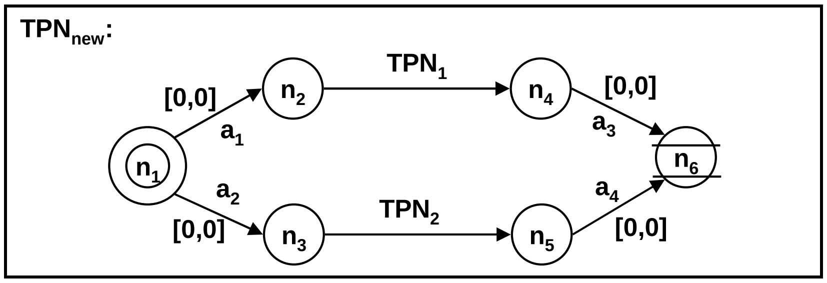


Figure 4.7: Choose RMPL Expression.

4.2.6 If Then Else

The *if then else* RMPL expression enables conditional execution, its corresponding TPN is depicted in Figure 4.8. This combinator behaves just like the *choose* combinator with a simple modification; there is a condition on each choice branch. The only difference is that two of the four timing constraints, a_1 and a_2, now have associated ask conditions: $a_1 = \langle Ask(c), n_1, n_2, [0,0], 0 \rangle$ and $a_2 = \langle Ask(not(c)), n_1, n_3, [0,0], 0 \rangle$.

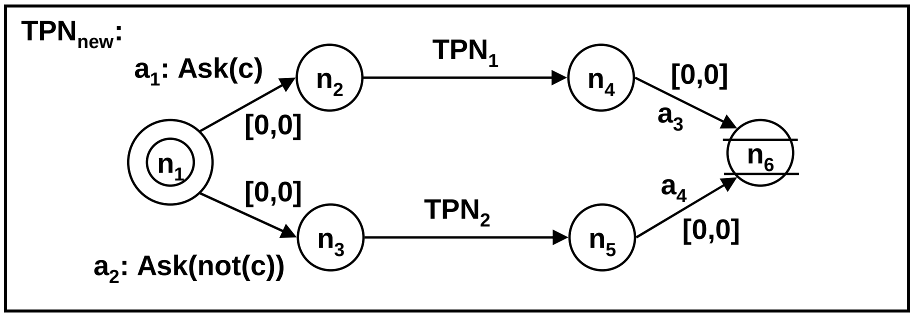


Figure 4.8: If Then Else RMPL Expression.

4.2.7 When Then

The *when then* RMPL expression enables reactive execution by composing three sub-expressions in sequence, its corresponding TPN is depicted in Figure 4.9. First, an infinite timing constraint, $a_1 = \langle \varnothing, n_1, n_2, [0,\infty], 0 \rangle$, second, a condition to be satisfied, $a_2 = \langle Ask(c), n_2, n_3, [0,0], 0 \rangle$, and third, a TPN to be executed. This combinator works by executing a noop indefinitely, a_1, until the condition *c* is satisfied, and then continuing on to execute the TPN. This combinator behaves just like the sequence combinator described in section 4.3.2.

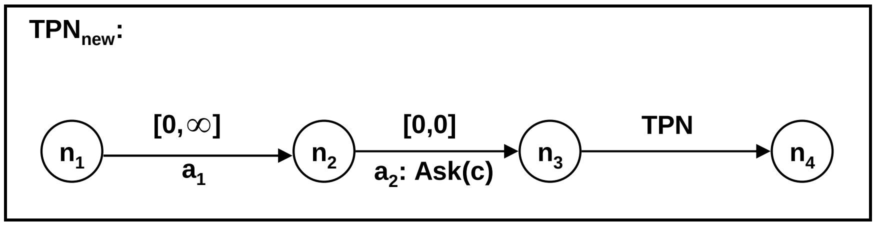


Figure 4.9: When Then RMPL Expression.

4.2.8 Do Maintaining

The *do maintaining* RMPL expression enables condition maintenance, and its corresponding TPN is depicted in Figure 4.10. To enforce a maintenance condition, the new TPN is composed such that a TPN and its maintenance condition, in the form of a state request, are forced to execute concurrently. This is accomplished with additional timing constraints between the start and end nodes of the maintenance condition, a_1

$=\langle Ask(s), n_1, n_2, [lb, ub], 0\rangle$, and the TPN. If the condition **s** is violated at runtime, Kirk halts the execution of the **TPN**. This same approach applies to the *do watching* RMPL combinator.

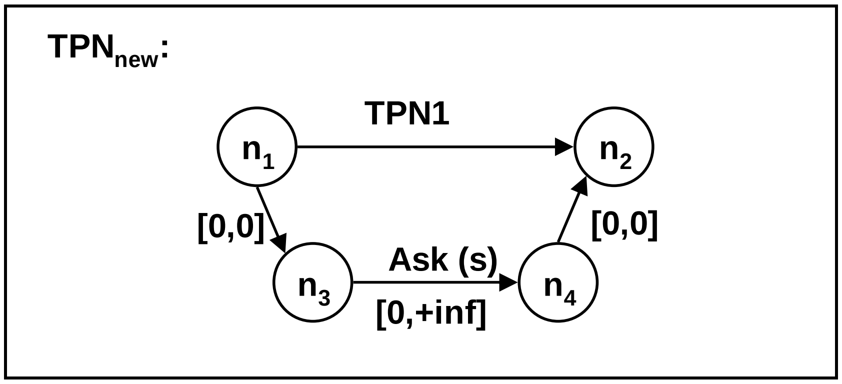


Figure 4.10: Do Maintaining RMPL Combinator

Next, we show that optimal method selection can also be encoded as an OCCSP by adding variables and values dynamically to the OCCSP encoding of a TPN.

4.3 Framing Optimal Method Selection as an OCCSP

In this section, we frame optimal method selection as solving an OCCSP. Through this contribution, optimal method selection is shown to be equivalent to finding an optimal solution to an OCCSP. As mentioned previously, this mapping is significant because it enables all past and future achievements in the field of constraint satisfaction to be applied to the problem of optimal method selection.

Optimal method selection, described in Section 3.3, occurs in four phases: candidate execution generation, temporal consistency check, symbolic consistency check, and completeness and optimality check. In the following sub-sections we describe how each of these phases can be framed as contributing to solving an OCCSP.

4.3.1 Phase 1: Candidate Execution Generation

In Definition 4.2, a TPN is encoded as an OCCSP where choices between redundant methods are encoded as discrete variable and values. Kirk’s candidate execution generation algorithm is responsible for choosing between redundant methods in order to generate candidate executions. With the redundant methods in a TPN encoded as OCCSP variables and values, Kirk’s candidate execution generation algorithm is effectively generating partial assignments to an OCCSP, thus fulfilling the “generate” aspect of the constraint satisfaction generate-and-test cycle described in Section 2.2.2.

4.3.2 Phase 2: Temporal Consistency Check

To test a candidate execution for temporal consistency, Kirk applies a temporal consistency algorithm, such as ITC, to a TPN with choices selected in search of temporal inconsistencies. When a TPN is encoded as an OCCSP, this phase corresponds to the “test” aspect of the constraint satisfaction generate-and-test cycle described in Section 2.2.2. Phase 1 of optimal method selection is just the first part of the “testing” however. A candidate execution must also pass the tests in Phase 3 and 4 of optimal method selection to be labeled as an optimal, consistent, and complete solution.

4.3.3 Phase 3: Symbolic Consistency Checking

In this section, we show that symbolic consistency checking, Phase 3 of optimal method selection, can be framed as a two-step process of 1) adding more discrete variables and values to the OCCSP encoding of a TPN, and then 2) searching the OCCSP for an optimal, complete, and consistent solution. Recall that symbolic consistency checking involves two steps:

1.) Detecting state assertion conflicts.

2.) Detecting selected methods whose maintenance conditions are not satisfied.

1.) Detecting State Assertion Conflicts:

The first step of symbolic consistency checking ensures that state assertion constraints are consistent. For example, in Figure 4.11, if the state constraints {*a* = 1} and {a = 0} overlap temporally in an execution, then the assertions are unsatisfiable. When two conflicting state assertions may overlap in time, we say they "threaten" one another. To resolve threats, Kirk adds additional timing constraints so that one follows the other.

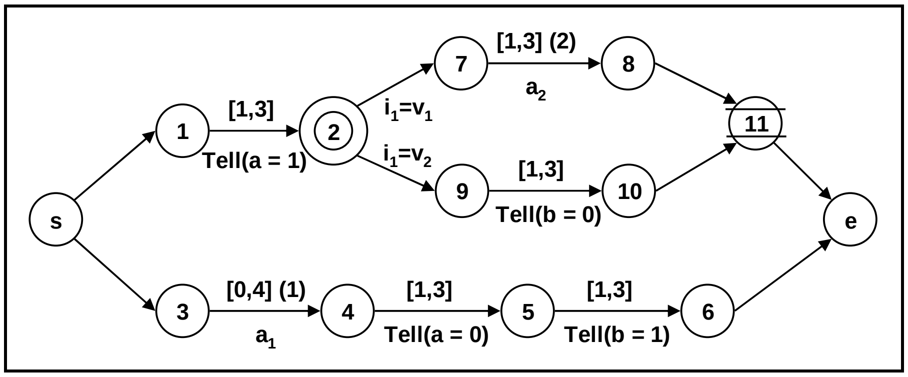


Figure 4.11: Conflicting state assertion constraints that threaten to co-occur.

For example, to resolve a state assertion threat, such as {*a* = 1} and {a = 0}, in Figure 4.11, Kirk has two possible options. The first option is to ensure that state assertion {*a* = 1} occurs before {a = 0} by adding a timing constraint, $r1 = \langle \varnothing, 2, 4, [0, \inf], 0 \rangle$, between the end event of Tell(*a* = 1) and the start event of Tell(*a* = 0), as shown in Figure 4.12 The second option is to ensure that assertion {*a* = 1} happens after {*a* = 0} by adding a timing constraint, $r2 = \langle \varnothing, 5, 1, [0, \inf], 0 \rangle$, between the end event of Tell(a = 0) and the start event of Tell(a = 1), also shown in Figure 4.12 The possible resolutions for state assertion threat {b = 0} and {b=1} are also depicted in Figure 4.12, as timing constraints r3 and r4.

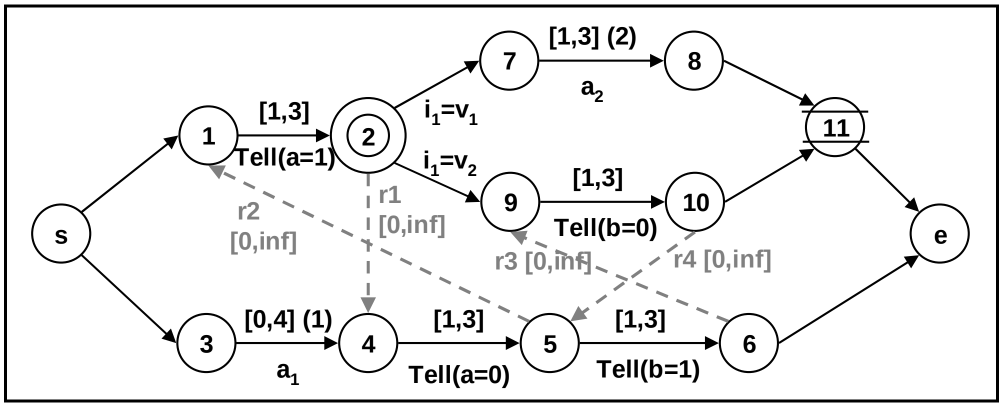


Figure 4.12: Resolving Conflicting State Assertions Framed as Solving an OCCSP

To resolve these threats, Kirk needs to consider all possible combinations of resolution constraints in order to find an optimal solution. This can be accomplished by adding a discrete choice variable to the OCCSP encoding of a TPN for each state assertion threat. The domain for each variable contains two values, one for each timing constraint that can resolve the threat. For example, two discrete variables, i_2 and i_3, are added to the OCCSP encoding of the TPN in Figure 4.12; one for each state assertion threat. The domain values for variables i_2 and i_3 are $v_2 = \{r1, r2\}$and $v_3 = \{r3, r4\}$, respectively. In addition, activity constraints are constructed for each new variable based on the variable-value assignments to which the conflicting state assertion constraints belong: $\{i_A = v_A\} \to active(i_2)$ and $\{i_A = v_A \wedge i_1 = v_2\} \to active(i_3)$. Intuitively, an activity constraint enforces the condition that a threat resolution only has to take place if both of the conflicting state assertion constraints are selected for execution.

Now, Step 1 of symbolic consistency checking is encoded into the OCCSP as well. Thus, an optimal, consistent, and complete solution to the OCCSP encoding of a TPN will resolve all state assertion threats. The optimal, complete, and consistent solution to the OCCSP encoding of the TPN in Figure 4.11 and 4.12 is $\{(i_A, v_A), (i_1, v_1), (i_2, v_1)\}$.

2.) Detecting methods whose maintenance conditions are not satisfied:

Step 2 of symbolic consistency checking ensures that all maintenance constraints in a candidate execution are satisfied. A maintenance constraint is represented in a TPN as a state request, Ask(x_i = v_i). Kirk ensures that state requests are satisfied by adding additional timing constraints into the plan which force a matching state assertion constraint to co-occur for the entire duration of the state request. A matching state assertion assigns the same variable and value as the state query, for example, Tell(foo = bar) is a matching state assertion for the state query Ask(foo = bar).

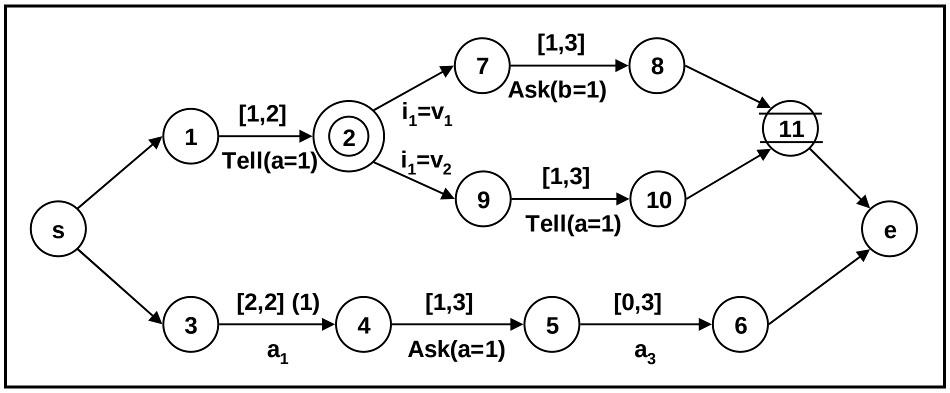


Figure 4.13: Examples of state requests, Ask(a = 1) and Ask(b = 2).

Kirk has as many options to satisfy a state request as there exist matching state assertions in the TPN to cover it. For example, in Figure 4.13, the Ask(a = 1) state request can be satisfied by forcing either of the two Tell(a = 1) state assertions to co-occur in the TPN, while the Ask(b = 2) state request has no satisfying state assertions in the TPN. Forcing a state assertion and a state request to co-occur, such as Ask(a = 1) and Tell(a = 1), is accomplished via two timing constraints. One to ensure that the state assertion begins before the state request begins, $r1 = \langle \varnothing ,1,4,[0,\inf],0\rangle$, and one to ensure that the state assertion ends after the state request ends, $r2 = \langle \varnothing ,4,2,[0,\inf],0\rangle$. There are two possibilities for satisfying the state request *Ask*(a=1) in Figure 4.13, both of which are shown in Figure 4.14.

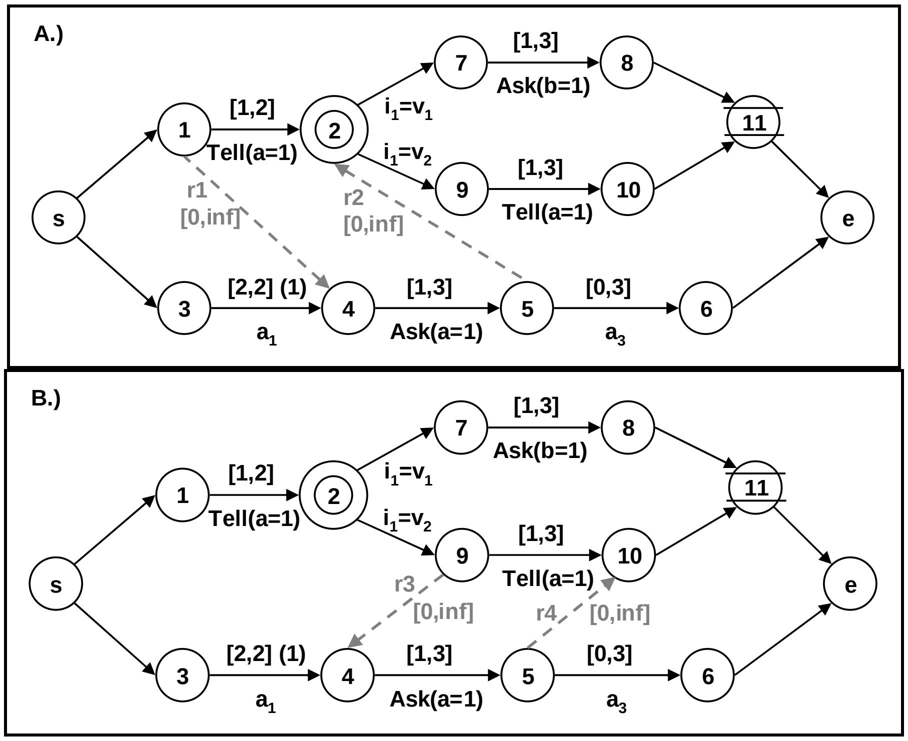


Figure 4.14: Satisfying state requests framed as OCCSP variables and values.

In order to find an optimal solution that satisfies all state requests, Kirk must consider all possible combinations of matching state requests and state assertions. This can be accomplished by adding a discrete choice variable to the OCCSP encoding of a TPN for each state query that needs to be satisfied. The domain for each variable contains a value for each matching state assertion in the TPN, and each value contains the timing constraints required for the matching state assertion to cover the state query. For example, two discrete variables, i_2 and i_3, are added to the OCCSP encoding of the TPN in Figure 4.13, one for each state request, Ask(a=1) and Ask(b=1), respectively. Variable i_2 has two values, $v_2 = \{A_{2a}, A_{2b}\}$ one for each matching state assertion, Tell(a=1), while variable i_3 has no values, $v_3 = \{\varnothing\}$ because there are no matching state assertions, Tell(b=1). The domains for variable i_2 contain the timing constraints required for the matching state assertion to cover the state query, $v_2 = \{A_{2a}, A_{2b}\} = \{\{r1, r2\}, \{r3, r4\}\}$. In addition, activity constraints are constructed for each new variable based on the variable-value assignments to which the matching state assertion and state query belong: $\{i_A = v_A\} \rightarrow active(i_2)$ and $\{i_1 = v_1\} \rightarrow active(i_3)$.

In addition, note that matching state assertion constraints may or may not be available depending on the threads of execution which are selected. For example, the Tell(a=1) constraint between events 9 and 10 in Figure 4.14 is only available to cover the Ask(a=1) constraint if the variable-value assignment $\{i_1 = v_2\}$ is selected. Therefore, each time Kirk reaches Phase 3 of the optimal method selection routine, the state assertion constraints available to cover any given state request constraint can change. The availability of a state assertion constraint depends on whether or not the thread upon which the state assertion resides has been selected for execution. To encode changes in the values of state request variables during optimal method selection, we can define activity constraints for each value of an OCCSP variable. For example, the activity constraints for each value in Figure 3.12 are: $\{i_A = v_A\} \rightarrow active((i_2, v_1))$ and $\{i_1 = v_2\} \rightarrow active((i_2, v_2))$.

If a state request variable, such as i_2 or i_3 is activated, but none of its values are activated, then that variable is self-inconsistent. Intuitively, this implies that there are no matching Tell constraints available in the candidate execution in which to cover an Ask constraint, such as Ask(b=1), and it is trivially inconsistent. If this occurs, the candidate execution is not satisfiable, and it must be passed back to Phase 1 in attempts to deactivate the self-inconsistent Ask constraint, or activate a satisfying Tell constraint to cover the Ask constraint.

Now, Step 2 of symbolic consistency checking is encoded into the OCCSP as well. Thus, an optimal, consistent, and complete solution to the OCCSP encoding of a TPN will satisfy all maintenance constraints. The optimal, complete, and consistent solution to the OCCSP encoding of the TPN in Figure 4.13 is $\{(i_A, v_A), (i_1, v_2), (i_2, v_2)\}$.

4.3.3 Phase 4: Completeness and Optimality Check

As discussed in Section 3.3, a candidate execution is complete if it makes a choice for every choice point reached by a selected thread of execution, starting from the beginning of the TPN. In an OCCSP encoding of a TPN, the hierarchical dependencies among choice points are encoded as activity constraints. Therefore, finding an OCCSP solution that satisfies all activity constraints is equivalent to ensuring completeness by making a choice for every choice point reached in the TPN. Similarly, an optimal execution is an execution with the lowest total cost, where the total cost is equal to the sum of all activity costs selected for execution. Encoding a TPN as an OCCSP does not change this. Optimality is still calculated by summing the costs of all activities selected for execution. The only difference is that, now, candidate executions and their associated costs are grouped into variable-value assignments, and are calculated from the cost function: $f(P) = \sum c_a \mid a \in A_m \wedge (i_k = A_{km}) \in P$. To find an optimal, complete, and consistent solution to an OCCSP encoding of a TPN, Kirk still performs an *incumbent* search, keeping track of the lowest cost, complete, and consistent execution found so far. Then, when no more candidate executions, in the form of OCCSP partial assignments, exist with a lower cost than the incumbent, the incumbent is returned, in the form of an optimal, complete, and consistent assignment of values to OCCSP variables, as an *optimal, complete, and consistent execution*.

Next, in Chapter 5, we extend Dynamic Backtracking to Solve OCCSPs.

Chapter 5 – Extending Dynamic Backtracking to Solve OCCSPs

In this chapter, we improve upon the state-of-the-art in solving OCCSPs by developing a conflict-directed OCCSP solver. To accomplish this, we extend Dynamic Backtracking (DB) to solve OCCSPs via four extensions to Ginsberg's original algorithm [13]. This new algorithm, called Conditional Dynamic Backtracking Branch and Bound (CondDB-B+B), employs infeasibility conflicts, suboptimality conflicts, and dynamic variable reordering, to quickly prune suboptimal and infeasible portions of the search space in order to quickly arrive at an optimal solution. While the pedagogical focus of this chapter is to extend the DB algorithm to solve OCCSPs, the ideas developed in this chapter more generally apply to extending all backjumping-based algorithms to solve OCCSPs.

To extend Dynamic Backtracking to solve Optimal Conditional CSPs, we augment the DB algorithm to handle activity constraints and soft constraints. This is accomplished via four extensions to the DB algorithm, which are summarized below, and described in detail in the following sections. The first three extensions address the problem of searching systematically and dynamically over conditional variables, while the last extension converts DB from a satisfaction algorithm into an optimization algorithm.

1.) A total variable ordering, I_O, for searching over conditional variables, and a conditional variable instantiation function.

2.) A modified backjumping resolution step which accounts for the behavior of conditional variables.

3.) A recursive check to remove deactivated variables from the partial solution when backjumping occurs.

4.) A branch-and-bound search optimization framework augmented to construct minimal suboptimal conflicts.

The pseudocode for this algorithm is presented as Algorithm 5.1 below, with each change to Ginsberg's original algorithm (DB) highlighted in grey and annotated with a superscript number indicating the extension it belongs to. In the following sections we give a detailed description of each of the four extensions.

Algorithm 5.1 – Conditional Dynamic Backtracking Branch and Bound (CondDB-B+B)

```
1. Set $P = \emptyset$, $I = I_I$. Set $E_i = \emptyset$ for each $i \in I$. (1) Take as input the total variable
   ordering, $I_O$. (4) Set the incumbent solution $N = (\emptyset, \infty)$.

2.a. (4) If $\hat{P} = I_A$, and $f(P) < f(N)$, P is the new incumbent solution. Set $N = (P, f(P))$.

2.b. (4) If $\hat{P} = I_A$, set $E_i = E_i \cup (i \neq v, \hat{P} - i)$. Otherwise, select a variable (1) $i = applyNextAC(\ )$
     (Function 5.1) and set $E_i = E_i \cup$ (4) $\varepsilon_O(P, i)$. (Definition 5.1)

3. Set $L = V_i - \hat{E}_i$. If J is nonempty, choose an element $v \in L$. Add $(i, v)$ to P and return
   to step 2.

4. If L is empty, we must have $\hat{E}_i = V_i$; let E be the set of all variables appearing in the
   explanations, T, of each elimination explanation, $(i \neq v, T)$ for each $v \in \hat{E}_i$, (2) plus all
   of the variables appearing in variable i's activating constraint, AC. (Proposition 5.1,
   OCCSP Backjumping Resolution Step)

5. If $E = \emptyset$, (4) return the incumbent, N. Otherwise, let $(j, v_j)$ be the last entry in P such
   that $j \in E$. Remove $(j, v_j)$ from P and for each variable $k \in P$ which was assigned a
   value after j, remove from $E_k$ any eliminating explanation that involves j. (3) Call
   removeUnsupportedVars( j , P ), and set,

   $$E_j = E_j \cup ^{(4)} \varepsilon_O(P, j) \cup \{(j \neq v_j, E \cap \hat{P})\}$$

      so that $v_j$ is eliminated as a value for j because of the values taken by variables in
   $E \cap \hat{P}$. Now set i = j and return to step 3.
```

(1) Extension #1, (2) Extension #2, (3) Extension #3, (4) Extension #4

5.1 Extension #1: A Systematic Method for Searching over Conditional Variables

This extension outfits DB with the basic machinery to search systematically over conditional variables. In previous work, systematic methods for searching over conditional variables have been developed [12,36]. One such method is to construct a directed graph from the conditional dependencies between variables [36]. Then, from this graph, a total variable ordering, I_O, can be derived which together with a conditional variable instantiation function results in a systematic search over the conditional variables. This method, called CondBT, is described in detail in [12] and [36].

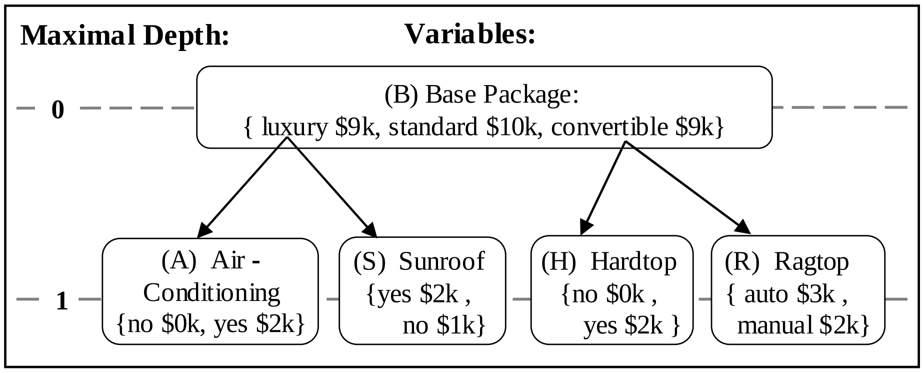
In order to enable DB to search systematically over conditional variables, we interleave the CondBT and DB algorithms. At first glance, it may appear that CondBT's strict variable ordering strategy is incompatible with DB's dynamic variable reordering technique. However, in actuality, the two merge quite nicely for the following reasons; CondBT restricts the order in which variables are added to the partial solution, while DB restricts the order in which variables are removed from the partial solution. CondBT is in charge of picking which unassigned variable should be instantiated next (in order to ensure a systematic search over the conditional variables), and DB is in charge of rearranging, reassigning, and unassigning variables once they have been instantiated (in order to perform Dynamic Backtracking).

Next, we summarize CondBT's variable instantiation strategy in four steps:

Step 1 - Create a Dependency Graph. The dependencies between a OCCSP's activity constraints, C_A, can be represented in the form of a directed graph, called a *dependency graph*, where the root node is defined as the set of all initially active variables, I_I. For example, the dependency graph for the car buyer example in Section 2.3.5 is shown in Figure 5.1, and the initially active variable is Base Package. Constructing a dependency graph helps to identify:

1.) Variables that will not become activated and can be removed (reachability check).
2.) Cycles in the activity constraints (cycle check).

3.) A partial order in which the activity constraints should be applied and retracted during search.



Maximal Depth:
Variables:
0
(B) Base Package: { luxury $9k, standard $10k, convertible $9k}
1
(A) Air - Conditioning {no $0k, yes $2k}
(S) Sunroof {yes $2k , no $1k}
(H) Hardtop {no $0k , yes $2k }
(R) Ragtop { auto $3k , manual $2k}

Figure 5.1: Dependency graph for the car configuration task in Section 2.3.5.

Step 2 – Eliminating Cycles in the Dependency Graph

Once a CCSP's dependency graph is constructed, any cycles in the graph must be eliminated by clustering the cyclic elements into a super-node. After all cycles have been collapsed, the new graph is called the *reduced dependency graph*, or RDG. The RDG will always be a directed acyclic graph (DAG). If a dependency graph contains no cycles, it is trivially an RDG, and also a DAG. Within the scope of this thesis, we will only encounter OCCSPs that are guaranteed by construction to contain no cycles in their dependency graphs, so Step 2 will always be trivially satisfied. For example, the car buyer example in Figure 5.1 contains no cycles so it is trivially the RDG.

Step 3 – Derive a Total Ordering, I_O

The RDG implies a partial ordering in which the activity constraints of an OCCSP should be applied and retracted during search. To determine the implied partial ordering, an integer value is defined for each node in the RDG, called the *maximal depth.* The maximal depth for each RDG node is defined as the number of nodes appearing above it in the RDG. If there happens to be more than one path into an RDG node, then the longest path must be taken as that node's maximal depth. For example, the maximal depth of each variable in the car buyer example is shown in Figure 5.1, and the implied partial ordering is: $\langle \{B\}, \{A, S, H, R\} \rangle$.

As defined by Gelle and Faltings [12], any two nodes with the same maximal depth are *incomparable*, and the order in which their corresponding activity constraints are applied is arbitrary. Thus, any total ordering, I_O, that obeys the implied partial ordering, is valid. For example, two valid total orderings for the car buyer example are $\langle B, A, S, H, R \rangle$ and $\langle B, A, S, R, H \rangle$.

Step 4 – The *CondBT* Algorithm

The *CondBT* algorithm enforces a sound and complete search over the conditional variables via two additions: a total variable ordering, I_O, and a function *applyNextAC()*. The function *applyNextAC( )* works by instantiating only *active* variables, and simply skips over variables that are *not active*, and is described in Function 5.1.

Function 5.1 **-** *applyNextAC( )*

This function simply scans I_O from beginning to end and returns the first variable, v, which satisfies two conditions.

1.) The variable *must not* belong to the current partial solution, P.

2.) The variables activating condition *must* be satisfied.

5.2 Extension #2: OCCSP Backjumping Resolution Step

As Extension #2, we augment the DB Backjumping Resolution Step (Proposition 2.1) to account for conditional variables. To do this, we inform the backjumping resolution step that a variable may be removed from the problem via conceding any one of the activation conditions used to instantiate it. Thus, when backjumping occurs, the activation conditions responsible for a variable presently being active are also added to the newly resolved conflict. This modified resolution step is described below.

Proposition 5.1 **- OCCSP Backjumping Resolution Step**

Let i be a variable with domain, $V_i = \{v_1, v_2, \ldots, v_m\}$, activity constraint $AC \to active(i)$, and let $P_1, P_2, \ldots, P_m$ be partial solutions that do not include *i*. If,

$$P_1 \cup \{(i, v_1)\}, P_2 \cup \{(i, v_2)\}, \ldots, P_m \cup \{(i, v_m)\}$$

are all conflicts, then,

$$P_1 \cup P_2 \cup \ldots \cup P_d \cup AC$$

is also a conflict.

Note that this new conflict can be resolved by removing variable i from the problem via conceding any one of its activation conditions, AC.

5.3 Extension #3: Checking for Deactivated Variables

Extension #3 is more straightforward than the previous two. When CondDB backjumps to a variable and changes its value, it is possible for that reassignment to deactivate other variables in the partial solution. In response, those variables must also be unassigned, removed from the partial solution, and the eliminating explanations depending on those variables must be erased. This is accomplished via the recursive function *removeUnsupportedVars(v,P)*.

Function 5.2 – removeUnsupportedVars(j , P)

```
for each variable $i \in \hat{P}$,
    if $i$'s activating constraint depends on the reassigned variable $j$, unassign variable $i$,
    remove $(i, v_i)$ from P, and for each variable k assigned a value after i, remove from $E_k$
    any eliminating explanation that involves i, by calling removeUnsupportedVars( i , P ).
```

5.4 Extension #4: A Branch-and-Bound Framework

To extend DB from a satisfaction algorithm to an optimization algorithm, we integrate Branch-and-Bound (B+B) search [37] into the DB algorithm. In addition, we augment B+B to construct minimal suboptimal conflicts. A B+B framework consists of three key attributes: an incumbent, a cost function, and a pruning mechanism. Branch-and-Bound Search (B+B) augments standard chronological backtracking (Section 2.2.3) search with two simple modifications:

1.) The search doesn't terminate when the first solution is found, but instead searches until the optimal solution is found.
2.) The lowest-cost solution found so far, called the incumbent, is stored and used to efficiently prune away sub-optimal portions of the search space.

To merge branch-and-bound search with Dynamic Backtracking we augment Ginsberg's elimination mechanism, ε, (Definition 2.5) to also prune candidate solutions with a higher cost than the incumbent solution. Therefore, we define a new *elimination mechanism* for the OCCSP, called ε_O , as follows:

Definition 5.1 **- OCCSP Elimination Mechanism,** $\varepsilon_O(P,i)$. We define a new *elimination mechanism* ε_O for the OCCSP as a function which accepts as arguments a partial solution, P , and a variable $i \notin \hat{P}$ and returns a (possibly empty) set $E_i = \varepsilon_O(P,i)$ of eliminating explanations for i . An elimination mechanism tries to extend a partial solution, P , by assigning each possible value for a variable i , and returns a reason for each value assignment which along with P is inconsistent or suboptimal given the problems constraints, C_C and C_A. If the extended partial solution is suboptimal, (has a cost greater than the current incumbent, N), $f(P \cup i = v) \geq f(N)$, then a subset of the extended partial solution, $M \subseteq P \cup i$ is returned as the reason for inconsistency, since its extension will be a *suboptimal* solution, $(i \neq v, \hat{M})$. Where M is determined by the Function 5.3, *minSubOptimalConflict(P,f(N))*.

Function 5.3 – minSubOptimalConflict(P , f(N))

Let $C_E \subseteq C_S$ be the cost associated with each variable value assignment expressed in the partial assignment $(i = v) \in P$, and thus contributors to $f(P)$. Let $\langle C_E \rangle$ be an ordered list of each assignment $\langle C_E \rangle = \langle i_j = v_j, i_k = v_k, \ldots \rangle$ such their associated costs, $f(i = v)$, are ordered from greatest to least. Let $\langle C_k \rangle$ be the first k elements of $\langle C_E \rangle$ such that their combined cost exceeds $f(N)$. Return $\langle C_k \rangle$ as a minimal suboptimal conflict.

This concludes our description of the four extensions to DB in order to solve OCCSPs, next we provide an example of CondDB-B+B solving an OCCSP.

5.5 Taking CondDB-B+B for a Test-Drive

In this section we give an execution trace of the CondDB-B+B algorithm solving the car buyer example. Initially, we assume *CondDB-B+B* receives as input, $I_I = \langle B \rangle$ and $I_O = \langle B, A, S, H, R \rangle$. In Figure 5.2, we show at each search step the partial solution, P, its cost, $f(P)$, the cost of the current incumbent, $f(N)$, the elimination explanations for each variable, E_i, and also the set of active variables which are not assigned values, $I_A - \hat{P}$.

Step 0 – *CondDB-B+B* is initialized with $P = \varnothing$, $\hat{E}_i = \varnothing$, $I_A = I_I = \langle B \rangle$, $I_O = \langle B, A, S, H, R \rangle$, and $N = (\varnothing, \infty)$.

Step 1 - $\hat{P} \neq I_A$, so the function *applyNextAC( )* is called and returns variable B. All three of B's value assignments are consistent with the constraints, C_C, and it is assigned the value 1. The new variable-value assignment $B = 1$ activates variables A and S.

Step 2 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable A. Both of A's value assignments are consistent with the constraints C_C, and it is assigned the value 1.

Search Step	P	f(P)	f(N)	$\widehat{I_A - P}$	E_i	B	A	S	H	R
0	{ ∅ }	-	∞	{ B }	1					
					2					
					3		▨	▨	▨	▨
1	{B=1}	9	∞	{ A,S }	1	✓				
					2					
					3		▨	▨	▨	▨
2	{B=1, A=1}	9	∞	{ S }	1	✓	✓			
					2					
					3		▨	▨	▨	▨
3	{B=1, A=1, ... }	9	∞	{∅}	1	✓	✓	{∅}		
					2			{∅}		
					3		▨	▨	▨	▨
4	{B=2}	10	10	{∅}	1	{∅}				
					2	☑				
					3		▨	▨	▨	▨
5	{B=3}	9	10	{ H,R }	1	{∅}				
					2	{∅}				
					3	✓	▨	▨	▨	▨
6	{B=3, H=1}	9	10	{ R }	1	{∅}			✓	
					2	{∅}			{B}	
					3	✓	▨	▨	▨	▨
7	{B=3, H=1, ... }	9	10	{∅}	1	{∅}			✓	{B}
					2	{∅}			{B}	{B}
					3	✓	▨	▨	▨	▨
8	{B=∅}	9	10	{ H,R }	1	{∅}				
					2	{∅}				
					3	{∅}	▨	▨	▨	▨
Return the Incumbent Solution, N = ({B=2} , 10).										

Figure 5.2: Solving the car buyer OCCSP with CondDB-B+B

Step 3 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable S. Both of $S's$ value assignments are *NOT* consistent with the constraints, C_C, so it is not added to the partial solution. The buyer does not like a sunroof! Since S is self-inconsistent, each of its elimination explanations are empty, $\{\varnothing\}$. $E_S = E_S \cup (S \neq 1, \{\varnothing\}) \cup (S \neq 2, \{\varnothing\})$.

Step 4 – $L = \varnothing$, so backjump, $E = \{\varnothing \cup \varnothing \cup B\} = \{B\}$. (Proposition 5.1) Let $(j, v_j) = (B,1)$. Remove $(B,1)$ from *P*, and erase any elimination explanations involving *B*. Call *removeUnsupportedVars(B,P),* which removes A from P. Set $E_B = E_B \cup (B \neq 1, \{\varnothing\})$, and set $B = 2$. Now, $\hat{P} = I_A$ and $f(P) < f(N)$, so that P is the new incumbent. Set $N = (\{B = 2\}, 10)$.

Step 5 - $\hat{P} = I_A$ so $E_B = E_B \cup (B \neq 2, \hat{P} - B)$ and $(B,3)$ is added to P. The assignment $\{B = 3\}$ activates H and R.

Step 6 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable H. The assignment $H = 2$ is pruned as suboptimal, $f(P \cup \{H = 2\}) \geq f(N)$. Add $H = 2$ to P.

Step 7 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable R. Both of $R's$ value assignments are pruned as suboptimal. Note that the conflict minimization function (Function 5.3) eliminates H=1 from each conflict, since the cost of the assignments to B and R alone (without even considering H) already comprise suboptimal conflicts.

Step 8 - $L = \varnothing$, so backjump, $E = \{\varnothing\}$. An empty conflict was produced. Return the incumbent, $N = (\{B = 2\}, 10)$. Search Success!!

Next, we sketch out proofs of completeness and termination for the CondDB-B+B algorithm.

5.6 Sketching Out Proofs of Completeness and Termination

We sketch proofs of completeness and termination for the CondDB-B+B algorithm, based on a small set of modifications to Ginsberg's proofs for Dynamic Backtracking [13]. Just as in [13], we begin by making the following assumptions about all elimination mechanisms:

- They are *correct.* We assume that suboptimal and infeasible partial assignments are always identified correctly (No false positives or negatives).

- They are *complete*. This part requires no modification from [13].
- They are *concise*. If a partial solution is suboptimal *and* infeasible, record only one of the two conflicts. Specifically, always record infeasibility.

5.6.1 Completeness

For CondDB-B+B we sketch completeness by proving three sufficient conditions:

1.) Retained eliminating explanations are always valid, and invalid eliminating explanations are always dropped.

2.) All possible combinations of conditionally activated variable assignments are considered during the search (The entire OCCSP search space).

3.) The search is guaranteed to find an optimal, complete, and consistent assignment of values to variables, if one exists.

Condition #1. To prove that retained eliminating explanations are always valid, we first need to prove that only valid eliminating explanations are added into the problem in the first place. There are two places in which CondDB-B+B adds new elimination explanations: the elimination mechanism (Definition 5.1) and the OCCSP backjumping resolution step (Proposition 5.1). Firstly, the elimination mechanism, as stated above, is assumed to only produce valid elimination explanations. Secondly, the OCCSP backjumping resolution step (Proposition 5.1) is defined in such a way that it creates a new valid elimination explanation from existing valid elimination explanations. Next, we point out that eliminating explanations are not altered after being created, so to complete our proof of condition #1, it suffices to show that valid eliminating explanations are always dropped as soon as they become invalid. Analogously to DB, the CondDB-B+B algorithm is guaranteed by construction to drop eliminating explanations as soon as they become invalid. This guarantee comes from the fact that all eliminating explanations are checked for validity every time a variable is unassigned and removed from the partial assignment, which is the only condition in-which an elimination explanation can become invalidated.

Condition #2. This condition is enforced by Extension #1 to DB, the CondBT( ) algorithm presented Section 5.1, which utilizes a total ordering, I_O, and a conditional variable instantiation function, *applyNextAC( )* to systematically search over the conditional variables.

Condition #3. Branch-and-bound style search is proven to find the optimal, complete, and consistent solution, if a solution exists. This is true even during dynamic variable reordering of the search tree simply because the cost function does not dependent upon the order of assignments in a partial solution, just what those assignments are.

5.6.2 Termination

Ginsberg proves termination for DB [13] by showing that the deductive consequences of all eliminating explanations grow monotonically as the DB algorithm proceeds. The same termination proof can be applied to CondDB-B+B. To state the argument intuitively, imagine an OCCSP inundated with so many activity constraints so that all variables always remain active. This is simply a CSP, and the original termination proof applies. If we reduce the number of activity constraints in the problem, the deductive consequences of all eliminating explanations still grows monotonically as search progresses, the only difference is that the valid search space is smaller, because not all combinations of conditional variables will be active at the same time. Thus, the same termination proof still applies. By pruning suboptimal sub-trees in addition to infeasible sub-trees we only strengthen the case for termination.

Chapter 6 – Improving TPN Search Efficiency

In this chapter, we develop four novel methods for improving TPN search efficiency.

1.) We use temporally inconsistent and sub-optimal partial executions, called *conflicts*, to guide optimal method selection towards an optimal, complete, and consistent execution.

2.) We introduce a technique for computing tight lower bounds during Branch and Bound search of TPNs that enables early pruning of sub-optimal partial executions.

3.) We define a relaxed union operator that enables early detection of temporal conflicts in a TPN.

4.) We develop a method to focus temporal conflicts by identifying and removing irrelevant choices, in the form of variable-value assignments, from the conflict.

6.1 – Using Conflicts to Guide Optimal Method Selection

We begin by describing prior work on detecting temporal conflicts in a TPN, in Section 6.1.1. Then, in Section 6.1.2, we explain how inconsistent and sub-optimal partial executions, called conflicts, can be used by a conflict-directed candidate execution generator to guide optimal method selection towards an optimal, complete, and consistent execution.

6.1.1 Detecting Temporal Conflicts in TPNs

Prior work [38,39] has enabled incremental conflict detection and resolution in TPNs via an algorithm called ITC. ITC is an incremental shortest path algorithm that is augmented to detect temporal conflicts in a TPN by looking for self loops in its set of support [39].

A temporal conflict is an over-constrained sub-network of activities in a TPN. For example, a temporal conflict is shown in Figure 6.1, with grey and black striped arrows. The top thread of execution with choice assignments {i_1=v_1} and {i_2=v_1} is constrained to end before the thread of execution with choice assignment {i_3=v_1} is allowed to end. When ITC detects a temporal conflict, such as the one in Figure 6.1, it returns the set of all time events involved in the conflict: {d,c_1,l,m,f,c_2,p,q,r,j,i,h,u,,t,c_3,e}.

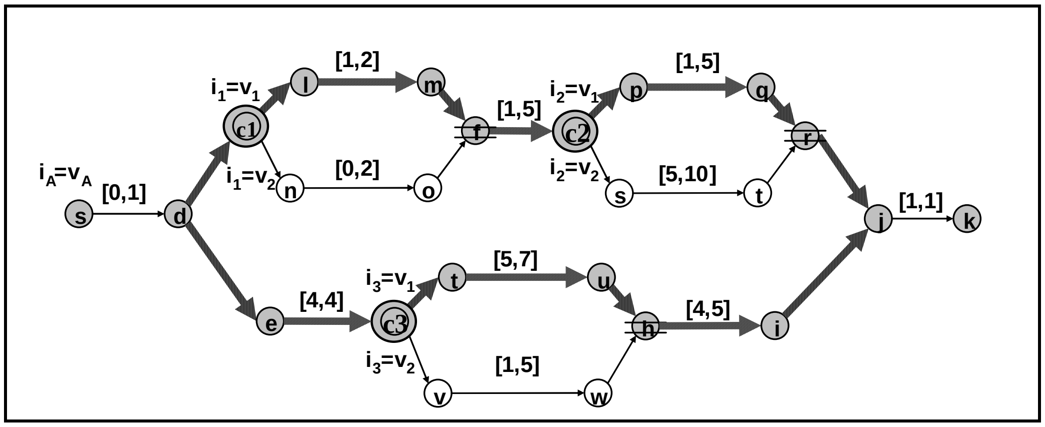


Figure 6.1: A Temporal Conflict, { i_A=v_A, i_1=v_1, i_2=v_1, i_3=v_1}.

6.1.2 Using Conflicts to Guide Optimal Method Selection

In this section, we explain how temporally inconsistent and sub-optimal partial executions, called *conflicts*, can be used to guide optimal method selection towards an optimal, complete, and consistent execution. To implement this capability we make two significant changes to Kirk's optimal method selection architecture. First, we augment each testing phase of optimal method selection, Phases 2 through 4, with the capability to return conflicts in the cases of inconsistency and sub-optimality, as depicted in Figure 6.2. Second, we develop a conflict-directed candidate execution generator for Phase 1 of optimal method selection that stores conflicts and only generates candidate executions that avoid all stored conflicts. The conflict-directed candidate execution generator is depicted in Figure 6.3. Next, we describe the method in which conflicts are returned from Phases 2 through 4. Then, we describe how the conflicts are used to guide optimal method selection.

To simplify the process of returning conflicts, phases 2 through 4 convert the conflicts into a uniform format before sending them to Phase 1. This uniform format is an OCCSP representation of a conflict of the form $\{ i_x = v_x , i_y = v_y , \ldots \}$ where each variable-value assignment represents a choice assignment in the TPN that contributes to the conflict. In Phases 2 and 3, the ITC algorithm detects conflicts as a set of time events that contribute to a temporal conflict, as described in Section 6.1.1. The process of converting a set of conflicting time events into the format described above is straightforward, and is described in Function 6.1, below. In Phase 4, a candidate

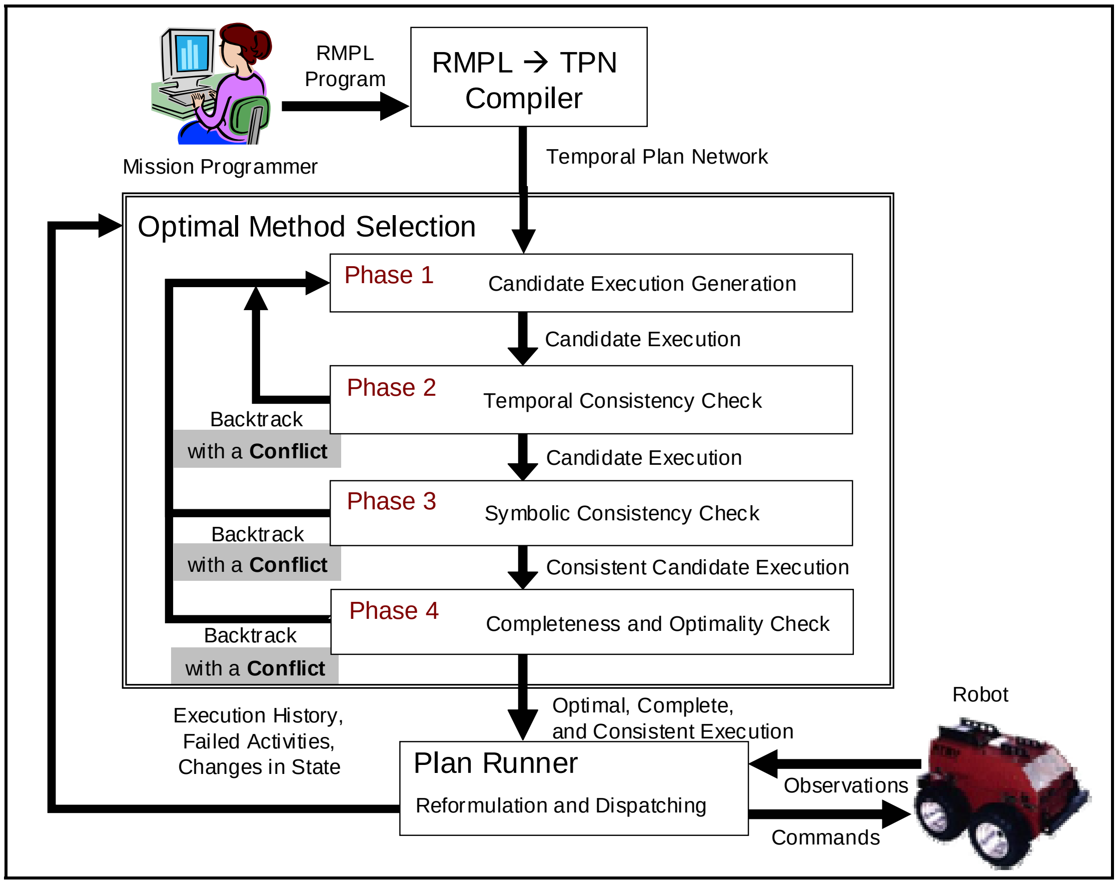

RMPL Program
RMPL → TPN Compiler
Mission Programmer
Temporal Plan Network
Optimal Method Selection
Phase 1
Candidate Execution Generation
Candidate Execution
Phase 2
Temporal Consistency Check
Backtrack with a Conflict
Candidate Execution
Phase 3
Symbolic Consistency Check
Backtrack with a Conflict
Consistent Candidate Execution
Phase 4
Completeness and Optimality Check
Backtrack with a Conflict
Execution History, Failed Activities, Changes in State
Optimal, Complete, and Consistent Execution
Robot
Plan Runner
Reformulation and Dispatching
Observations
Commands

Figure 6.2: Conflict-directed Optimal Method Selection

Function 6.1 **– timeEventsToVarValAssignments (*T, OCCSP_TPN*)**

```
Let T be a set of time events, and let OCCSP_TPN be an OCCSP encoding of a TPN.
Initialize the variable value assignments to return to null: $C = \{\emptyset\}$.
For each time event, $t \in T$,
    Let A be the set of activities in OCCSP_TPN such that time event t is either its
    start or end event.
    For each $a \in A$,
        If a's start event is t, let v be a's end event.
        Else, let v be a's start event.
        If v also belongs to T,
            Let $\{i_k=v_k\}$ be the variable value assignment that activates a in the
            OCCSP_TPN. Add $\{i_k=v_k\}$ to C.
Return C.
```

Function 6.2 **– suboptimalExecutionToVarValAssignments (*S, N*)**

```
Let S be a suboptimal candidate execution, and let N be the current incumbent.
Let S' be the OCCSP variable value assignments of each choice in the suboptimal
candidate execution. Let N' be the OCCSP variable value assignments of each
choice assigned in the current incumbent. Let f(N') be the cost of N'.
Initialize the variable value assignments to return to null: $C = \{\emptyset\}$.
C = minSubOptimalConflict( P' , f( N' ) ) (Function 5.3)
Return C.
```

execution is tested for completeness and optimality. A suboptimal candidate execution is one whose total cost, which is the sum of all activity costs in the candidate execution, exceeds that of the best cost, complete, and consistent execution found so far, called the incumbent. A suboptimal candidate execution is converted into OCCSP variable value assignments using Function 6.2, above. An incomplete candidate execution is one in-which more choices need to be made to fulfill the mission objectives. An incomplete candidate execution can be detected by looking for unassigned choice points along the selected threads of execution in a TPN, or alternatively, by checking that the activity constraints, C_A, in the OCCSP encoding of a TPN are not all satisfied. An incomplete candidate execution is a consistent candidate execution that is passed back to Phase 1 for more choices to be selected. Since incomplete candidate executions are not inconsistent or suboptimal, they do not return a conflict when backtracking.

Next, we describe the changes to Phase 1, candidate execution generation, in order to use conflicts to guide optimal method selection. To enable conflict-directed candidate generation, we augment Kirk's candidate execution generator to operate directly on the OCCSP encoding of a TPN, from Chapter 4. Through this approach, any OCCSP algorithm, such as Conditional Dynamic Backtracking Branch and Bound (CondDB-B+B) from Chapter 5, can be applied directly to the problem of optimal method selection. A depiction of this architecture is shown in Figure 6.3. The candidate execution generator takes as input a TPN, which is then encoded as an OCCSP, and conflicts, in the form of OCCSP variable-value assignments. Any OCCSP search algorithm, such as

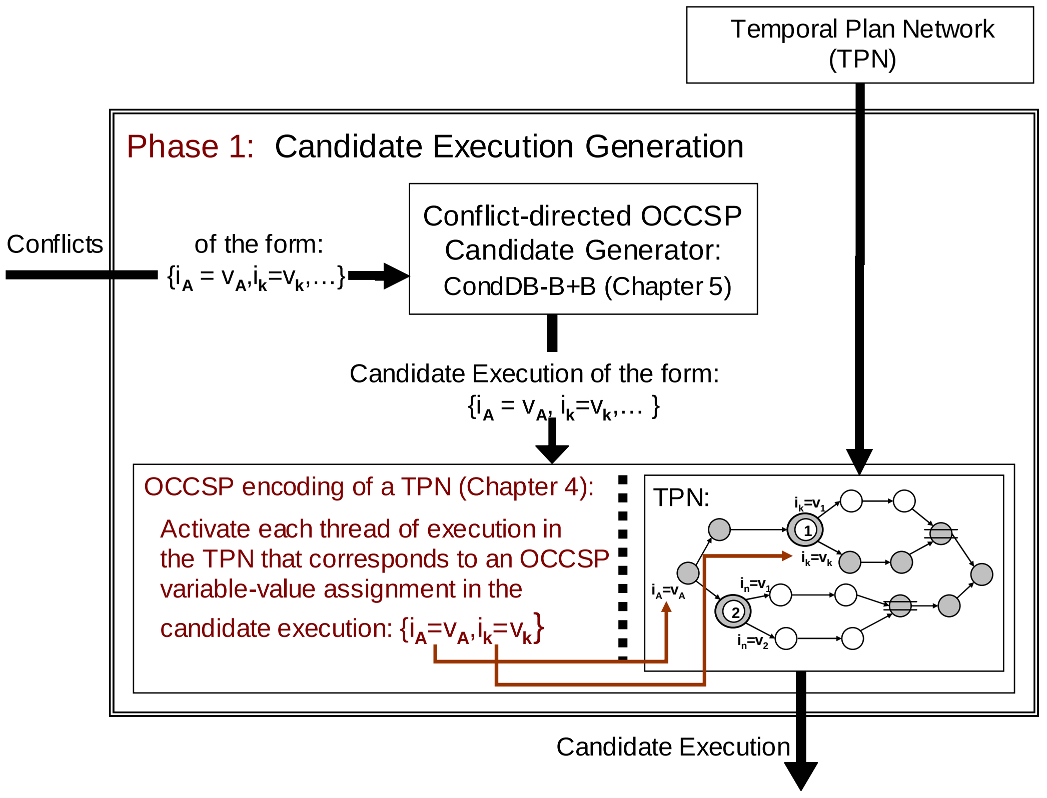

Temporal Plan Network (TPN)
Phase 1: Candidate Execution Generation
Conflicts
of the form: {i_A = v_A, i_k=v_k,…}
Conflict-directed OCCSP Candidate Generator: CondDB-B+B (Chapter 5)
Candidate Execution of the form: {i_A = v_A, i_k=v_k,… }
OCCSP encoding of a TPN (Chapter 4):
Activate each thread of execution in the TPN that corresponds to an OCCSP variable-value assignment in the candidate execution: {i_A=v_A, i_k=v_k}
TPN:
i_k=v_1
i_k=v_k
i_A=v_A
i_n=v_1
i_n=v_2
1
2
Candidate Execution

Figure 6.3: Conflict-directed Candidate Execution Generation

CondDB-B+B can then be used to perform candidate execution generation. A candidate execution comes from the candidate generator in the form of variable value assignments, of the form { $i_x = v_x$, $i_y = v_y$, … }. A candidate execution is then selected from the TPN by activating each thread of execution that corresponds to an OCCSP variable-value assignment, as depicted in Figure 6.3. Then the candidate execution is passed on to Phases 2 through 4.

Next, we describe a technique that improves the ability of Branch and Bound search to prune suboptimal partial executions.

6.2 – A Technique for Computing Tight Lower Bounds

In this section, we improve the pruning power of Branch and Bound search of TPNs by introducing a technique for computing tight lower bounds. A lower bound is a cost associated with a partial execution, and is an underestimate of the lowest cost complete and consistent execution extending from that partial execution. In Branch and Bound search of TPNs, the lower bound is used to prune partial executions that are guaranteed to be suboptimal. The tightness of a lower bound refers to how close it estimates the *actual* lowest cost complete and consistent execution that extends from a partial execution. A tight lower bound improves Branch and Bound's ability to detect and prune suboptimal

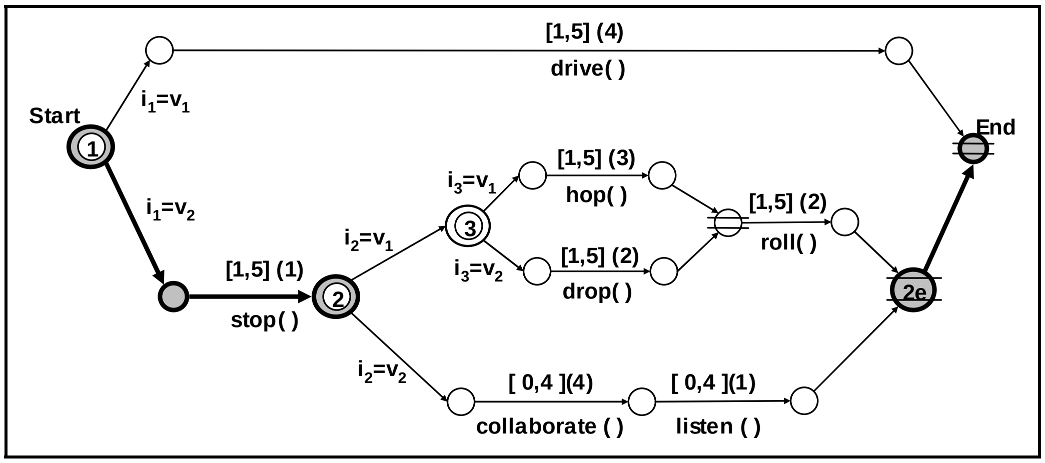


Figure 6.4: A Partial Execution, {i_1=v_2}.

partial executions. As an example, consider the TPN in Figure 6.4. Suppose that we start with the partial execution $\{i_1 = v_2\}$, highlighted in grey and bold. The *actual* lowest cost of a complete and consistent execution extending partial execution $\{i_1 = v_2\}$ is 5, and corresponds to the choice assignments $\{i_1 = v_2,\ i_2 = v_1,\ i_3 = v_2\}$, or analogously, the execution thread "stop( )", "drop( )", and "roll( )". One valid lower bound is simply the cost of a partial execution itself, which for $\{i_1 = v_2\}$ is 1. This lower bound is not very tight, however, and significantly underestimates the actual cost of extending a partial execution to a complete and consistent execution, which for $\{i_1 = v_2\}$ is 5. Next, we introduce a technique for computing tighter lower bounds.

As a tighter lower bound for Branch and Bound search of TPNs, we compute the lowest cost, complete, but not necessarily consistent, execution that extends a partial execution. This lower bound is effective for two reasons:

1) it often closely underestimates the cost of the complete *and consistent* execution with lowest cost, and

2) it has an attractive time complexity; linear in the number of choices in a TPN.

To compute this lower bound in linear time, a pre-processing step is required. We pre-compute the lowest cost thread of execution through each choice assignment in a TPN. For example, in Figure 6.5, we place next to each choice assignment a number in brackets and highlighted in grey, such as $\{i_1 = v_2\}$<5>, which represents the lowest cost of a thread of execution through that choice assignment. These costs can be computed recursively, as described in Function 6.3, by calling *lowestCost*(i_A,v_A) on a TPN's base choice. The computational complexity of this pre-processing step is linear in the number of activities in a TPN.

Function 6.3 **– lowestCost(*i,v_i*)**

```
Initialize the lowest cost, c, of ( $i$ , $v_i$ ) to zero, c = 0.
For each variable j activated by the assignment ( $i = v_i$ ),
        $m = m + MINIMUM\{\ f( i = v_i ) + lowestCost( j , v ) ,\ for\ each\ v \in v_j\ \}$
```

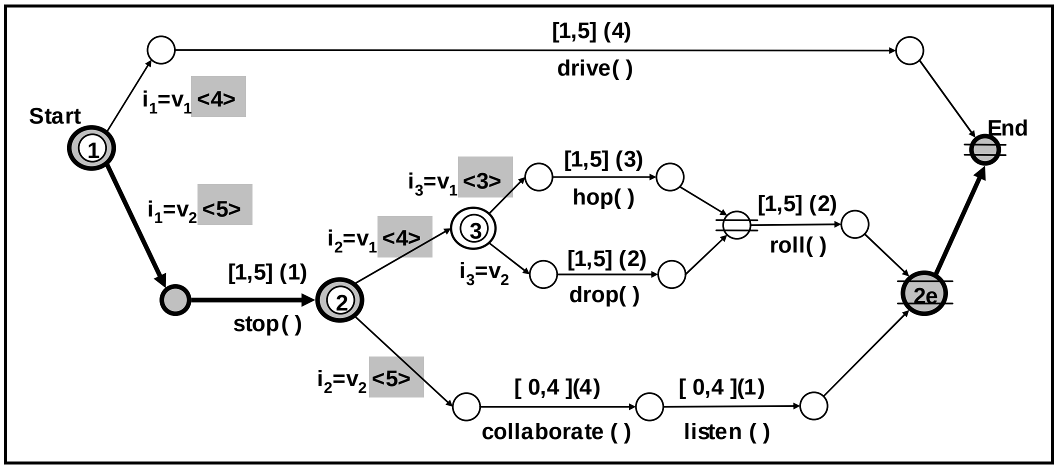


Figure 6.5: A Pre-processing Step for Computing Tight Lower Bounds.

After the pre-processing step is completed, the lower bound for a partial execution is calculated as described in Function 6.4. We use the partial execution {i_1=v_2} as a running example in describing Function 6.4.

Function 6.4 – computeLowerBound(P)

1.) Take as input a partial execution, P. For example, {i_1=v_2}.
2.) Initialize the lower bound to zero, *lb = 0.*
3.) Add to *lb* the cost of the partial execution. For example, *lb = 0+ f(i_1=v_2) = 1.*
4.) For each choice point in the partial execution without a value assigned, such as choice point 2, pick the lowest cost estimate of extending that choice point to a complete execution, and add it to *lb*. For example, *lb = 1+ MINIMUM{4,5} = 5.*
5.) Return *lb.*

To demonstrate the effectiveness of the tight lower bound (Function 6.4), we solve the TPN from Figure 6.4 using Branch and Bound (B+B) search first without the tight lower bound, and then with it. The search tree for B+B without the tight lower bound is shown in Figure 6.6a, while the search tree for B+B with the tight lower bound is shown in Figure 6.6b.

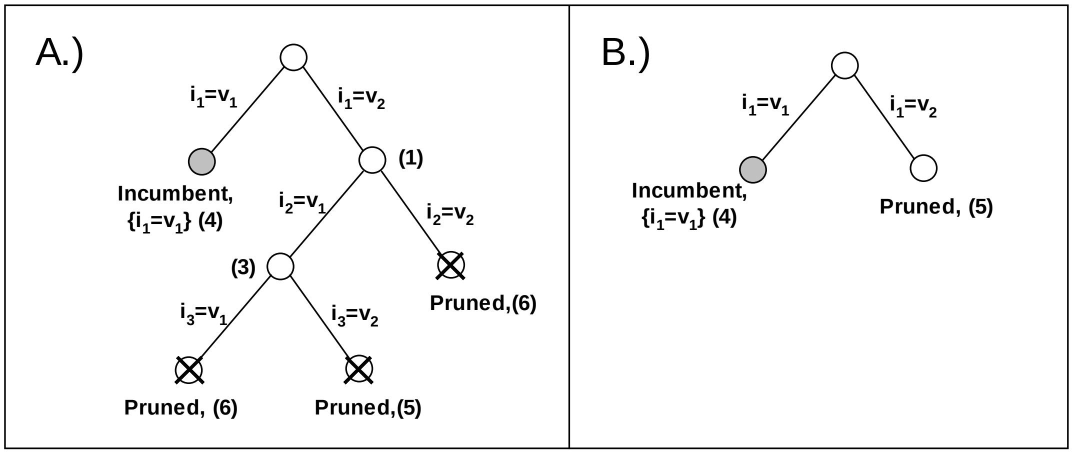


Figure 6.6: Branch and Bound Search with and without tight lower bounds.

In this example, tight lower bounds (Function 6.4) added significant pruning power to the B+B algorithm, reducing the number of search iterations from 6 to 2. This performance improvement scales well to larger TPNs, as demonstrated in our empirical results in Chapter 7.

6.3 – A Relaxed Union Operator that Enables Early Detection of Temporal Conflicts

In this section, we define a relaxed union operator that enables early detection of temporal conflicts by bounding the feasible durations of choice expressions in a TPN, where a bound is represented as a single interval constraint. These bounds are added into a TPN at compile time, enabling the early detection of temporal conflicts. For example, consider the TPN in Figure 6.7a. Using a relaxed union operator for TPNs, which is defined in Section 6.3.1, we pre-compute a bound on the feasible durations of the choice C. This bound is represented in the form of a single interval constraint, called Crelaxed, as depicted in Figure 6.7b, and indicates the shortest and longest possible durations for choice C. Adding this new constraint into the TPN between choice C's start and end events allows us to detect immediately that this TPN will be inconsistent no matter what choice we make for C. Adding these relaxed bounds into a TPN, in general, saves an

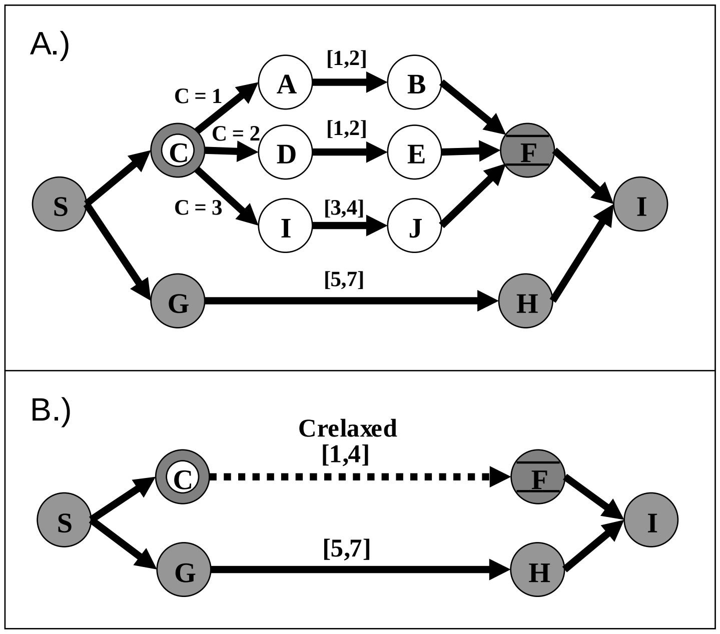


Figure 6.7: A Relaxed Constraint: The Feasible Bound on a Choice Expression.

exponential amount of time during search, while the time complexity for pre-computing them is linear in the number of activities in a TPN. We conclude this section by defining the relaxed union operator for TPNs.

6.3.1. Defining the Relaxed Union Operator for TPNs

Next, we define the *relaxed union operator* for TPNs that extends the *union operator* defined by Dechter, Meiri, and Pearl [7] for temporal constraint networks. Let $T = \{I_1,...,I_l\}$ and $S = \{J_1,...,I_m\}$ be metric timing constraints which imply sets of allowable intervals of a real variable *t*. Both of these operators are depicted in Figure 6.8.

Definition 6.1 **– The Union Operator:** The *union* of *T* and *S*, denoted $T \cup S$, admits only values that are allowed by either one of them, $T \cup S = \{I_1,...,I_l, J_1,...,J_m\}$, as indicated in Figure 6.4.

***Definition 6.2* – The Relaxed Union Operator:** The *relaxed union* of *T* and *S*, denoted $T \tilde{\cup} S$, admits any value between the highest and lowest values of $T \cup S$, as indicated in Figure 6.4.

In Figure 6.8, the highest value of $T \cup S$ is 8 and the lowest value of $T \cup S$ is 0. Therefore, $T \tilde{\cup} S$ admits any value between 0 and 8. The relaxed union operator is not a new concept. It has been commonly employed in the area of *interval arithmetic* [1] to retain tractability when performing arithmetic over large sets of intervals. What is novel, is to apply this operator to tractably pre-compute temporal conflicts in a TPN.

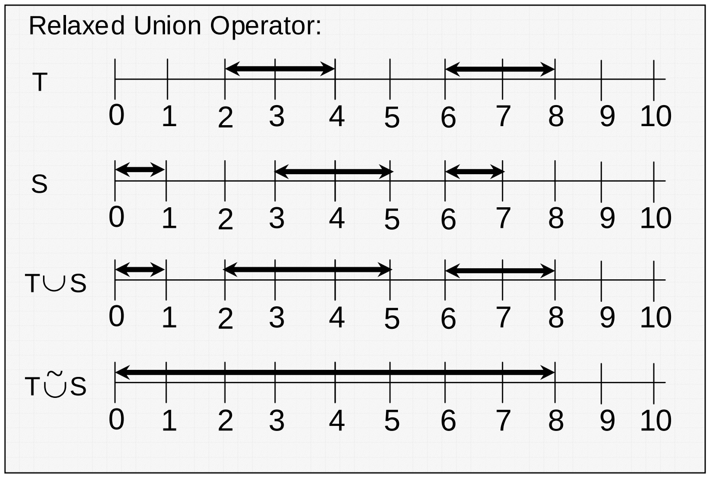


Figure 6.8: The Union and Relaxed Union Operators

6.4 – Extracting Focused Temporal Conflicts from a TPN

First, in section 6.4.1, we explain how to extract focused temporal conflicts from a TPN by identifying and removing irrelevant variable-value assignments from a conflict. Then, in Section 6.4.2, we extend upon the ideas of Section 6.4.1 and develop an algorithm that identifies and removes large sets of irrelevant variable value assignments from a temporal conflict.

6.4.1 Identifying Irrelevant Variable-Value Assignments

A key observation emanating from this thesis is that irrelevant variable-value assignments can be identified and eliminated from a conflict returned by ITC. The reason is that there are disjunctive constraints in a TPN that are not captured in a candidate execution when it is tested for consistency by ITC, but that can be used to eliminate variable-value assignments from a conflict after being returned by ITC. This is an important observation because removing irrelevant variable-value assignments from conflicts dramatically improves the performance of conflict-directed search algorithms.

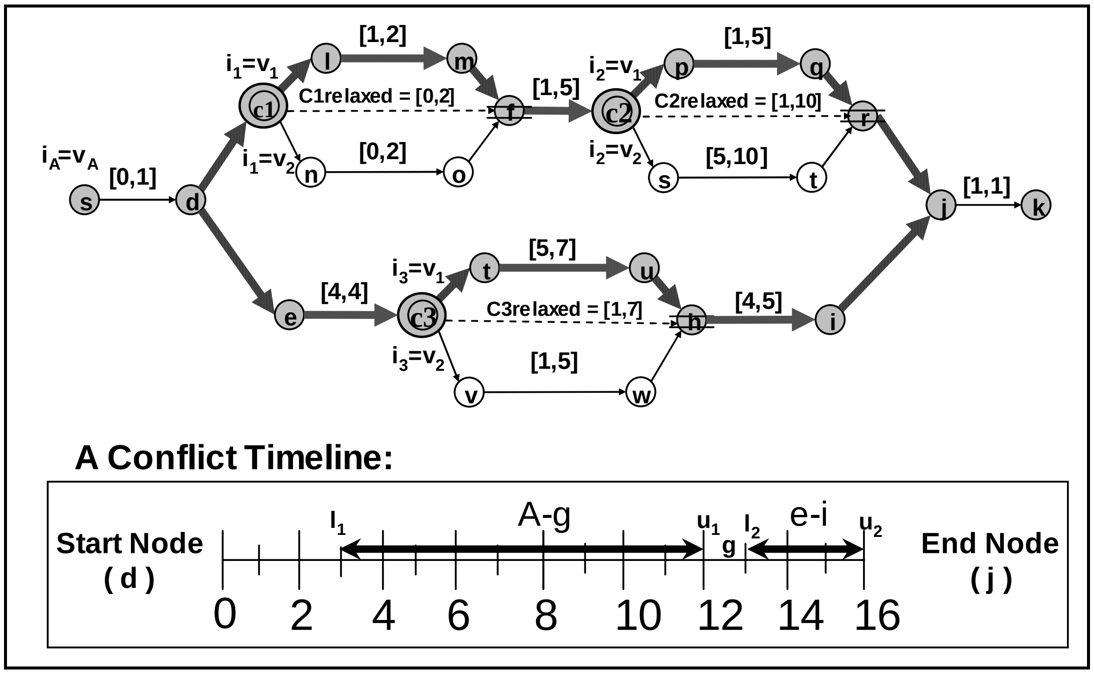


Figure 6.9: An Alternative Way to View of a Temporal Conflict: A Conflict Timeline.

In order to detect irrelevant variable value assignments, in Figure 6.9, we provide an alternative way to view conflicts, called a conflict timeline. A conflict timeline represents with double headed arrows the allowable durations of two conflicting parallel threads of execution, such as the parallel threads c_1-r and e-i in Figure 6.9. Next, we define several terms related to a conflict timeline that will be useful when explaining how to detect irrelevant variable-value assignments.

Definition 6.3 **– A Conflict Timeline**

(we use the conflict in Figure 6.6 as a running example)

The start and end events of the two conflicting threads of execution: {d,j}.

The short duration thread: { c1-r }.

The long duration thread: { e-i }.

The short duration thread's upper bound, $u_1 = 12$.

The long duration thread's lower bound, $l_2 = 13$.

The temporal gap: $g = l_2 - u_1 = 13 - 12 = 1$.

To resolve a conflict, the temporal gap, g, as defined in Definition 6.3, needs to become less than or equal to zero so that the two parallel threads, c_1-r and e-i, can begin and end simultaneously. To reduce the temporal gap, g, we can attempt to make different choices to the variables involved in the conflict, C_1, C_2, and C_3. An alternative choice is only beneficial, however, if it contributes towards eliminating the temporal gap, g. We define a variable-value assignment as 'irrelevant' if no alternative choice for that variable helps to resolve the conflict. For example, in Figure 6.9, variable C1 is irrelevant because the alternative choice for C1, $\{C_1 = v_2\}$, has the same upper bound as the current choice, $\{C_1 = v_2\}$, and does not contribute to resolving the temporal conflict, by reducing the temporal gap, g, between the two conflicting threads of execution. Thus, we can remove the variable-value assignment $\{C_1 = v_2\}$ from the conflict, because the remaining assignments, $\{C_2 = v1 , C_3 = v_1\}$ still infer a temporal conflict regardless of the choice for variable C_1.

More precisely, we define a variable-value assignment as 'irrelevant' if the absolute value of the difference between its relaxed bound and its actual bound, called the *temporal slack*, is less than the *temporal gap*, g. If a variable-value assignment resides along a conflict's *short duration thread* then we compare its upper bound, for example $|ub_{Arelaxed} - ub_{A1}| = 2 - 2 = 0,$ and if a variable-value assignment resides along a conflict's *long duration thread* then we compare its lower bound, for example $|lb_{Crelaxed} - lb_{C1}| = 5 - 1 = 4.$ For example, the temporal slack for variable-value assignment $C_1 = v_1$ is less than the temporal gap, $0 < 1$, indicating that C1 = v1 is irrelevant, since no alternative choice for A can possibly resolve the temporal conflict.

The temporal slack for variable-value assignment $C_2 = v_1$ is greater than the temporal gap, 4 > 1, indicating that C = 1 is *not* extraneous because an alternative choice for C exists that can resolve the temporal conflict.

In the next section, we develop a systematic method for identifying large sets of irrelevant variable-value assignments in a temporal conflict.

6.4.2 Identifying Large Sets of Irrelevant Variable-Value Assignments

In this section we develop an algorithm called, *extractTemporalConflict( )*, that can identify and remove a large set of irrelevant variable-value assignments from a temporal conflict. To motivate the importance of this approach, consider the temporal conflict highlighted in grey in Figure 6.10. We will be able to remove a large set of the variable-value assignments from this conflict.

To begin, we calculate the temporal slack for each variable-value assignment involved in the conflict, as described in Section 6.4.1:

{A = 1}: $\left|lb_{Arelaxed} - lb_{A1}\right| = 7 - 3 = 4,$ {B = 1}: $\left|lb_{Brelaxed} - lb_{B1}\right| = 1 - 0 = 1,$

{C = 1}: $\left|lb_{Crelaxed} - lb_{C1}\right| = 1 - 0 = 1,$ {D = 1}: $\left|lb_{Arelaxed} - lb_{D1}\right| = 1 - 0 = 1,$

{E = 1}: $\left|ub_{Erelaxed} - ub_{E1}\right| = 6 - 5 = 1.$

The *temporal slack* for each variable-value assignment in the conflict is less than the *temporal gap* of 5. Therefore, any of the variable-value assignments could be considered 'irrelevant' and removed from the conflict. The trick here becomes, how do we remove the largest number of 'irrelevant' variables from the conflict in order to best focus the temporal conflict? The answer to this question is described in Function 6.5, below.

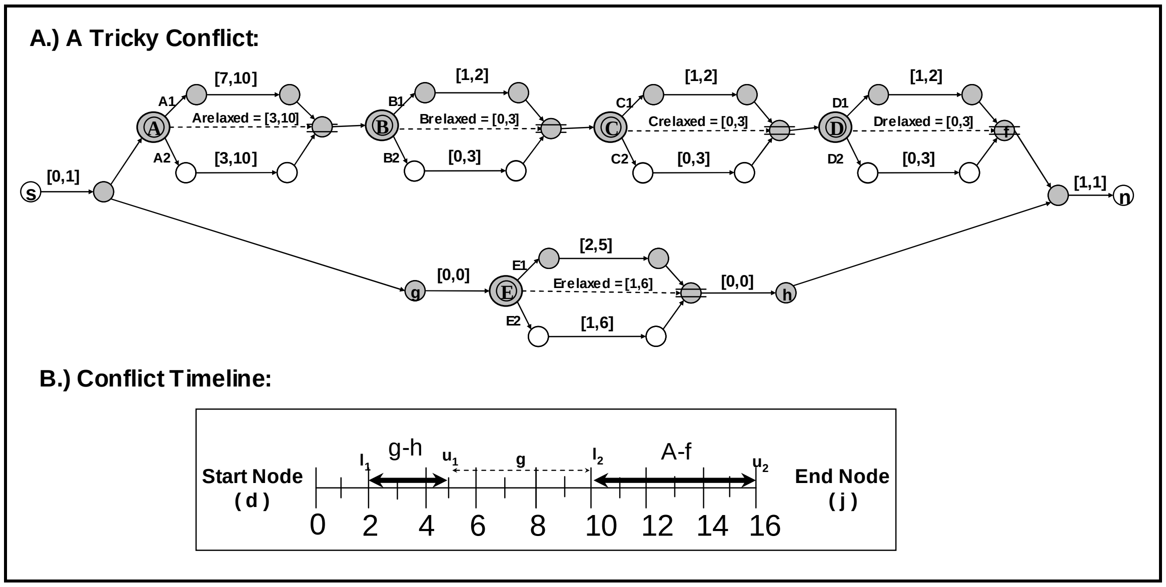


Figure 6.10: A Tricky Conflict.

Function 6.5 **– ExtractTemporalConflict( ):**

Our strategy for identifying the largest set of irrelevant assignments given a temporal conflict is simple:

1.) Order the assignments from least to greatest according to their *temporal slack* values.

- For example, ({B=1}=1 , {C=1}=1 , {D=1}=1 , {E=1}=1 , {A=1}=4).

2.) Remove the remaining assignment with the *least* temporal slack until the *temporal gap* becomes less than or equal to zero. Then, return the remaining assignments as a focused conflict. For example,

- Remove, {B=1}, such that the temporal gap, g = 5 – 1 = 4.
- Remove, {C=1}, such that the temporal gap, g = 4 – 1 = 3.
- Remove, {D=1}, such that the temporal gap, g = 3 – 1 = 2.
- Remove, {E=1}, such that the temporal gap, g = 2 – 1 = 1.
- Try to remove, {A = 1}, but the temporal gap, g = 1 – 4 = -3, becomes negative. Return the remaining assignments as a focused conflict: {A=1}.

Next, we develop an extension to the extractTemporalConflict( ) function that enables it to handle nested choices in a temporal conflict.

Extending extractTemporalConflict( ) to Handle Nested Choices:
To handle the more general case, where nested choices can exist in a temporal conflict, we make one simple addition to the extractTemporalConflict( ) function. In addition to ordering the assignments from greatest to least, we add an additional constraint so that nested choice assignments are removed from the conflict before their parent assignments. This ensures that the largest irrelevant set of assignments is removed from a conflict. The algorithm is presented again in Function 6.6, with the extension highlighted in grey:

***Function 6.6* – ExtractTemporalConflict( ) (with extension to handle nested choices)**

1.) Order the assignments from least to greatest according to their *temporal slack* values.

2.) Remove the remaining assignment with the *least* temporal slack *that* in addition has no children assignments remaining in the conflict, until the *temporal gap* becomes less than or equal to zero. Then, return the remaining assignments as a focused conflict.

Next, we provide empirical results that demonstrate the significance of the enhancements developed in this chapter, as well as implementation details, a discussion of future work, and conclusions.

Chapter 7 – Empirical Results, Discussion, Implementation, Future Work, and Conclusions

7.1 Overview

In this chapter we present empirical results, a discussion of the results, implementation details, possible directions for future work, and concluding remarks.

7.2 Empirical Results and Discussion

In this section, we present empirical results for each of the contributions developed in this thesis. First, in Section 7.2.1, we describe a random TPN generator that was used to generate TPNs for the empirical evaluation. Then, in section 7.2.2 we describe the algorithms that were evaluated. In Section 7.2.3 we present the empirical results along with a discussion of the results.

7.2.1 Random TPN Generator

A random TPN generator was developed to test Kirk's performance on a wide variety of TPNs. The random TPN generator varies the generated TPNs over two dimensions:

1) the number of choices in parallel, called the branching factor (b)
2) the depth of nested choices, called the nest level (n)

Figure 7.1 shows a typical randomly generated TPN, and the black arrows represent each parameter that can be varied to change the dimension of TPN being generated. In the experiments, the activity time bounds were adjusted until the randomly generated plans were approximately 50% solvable, which means they are on the phase transition where difficult problems are most common [2].

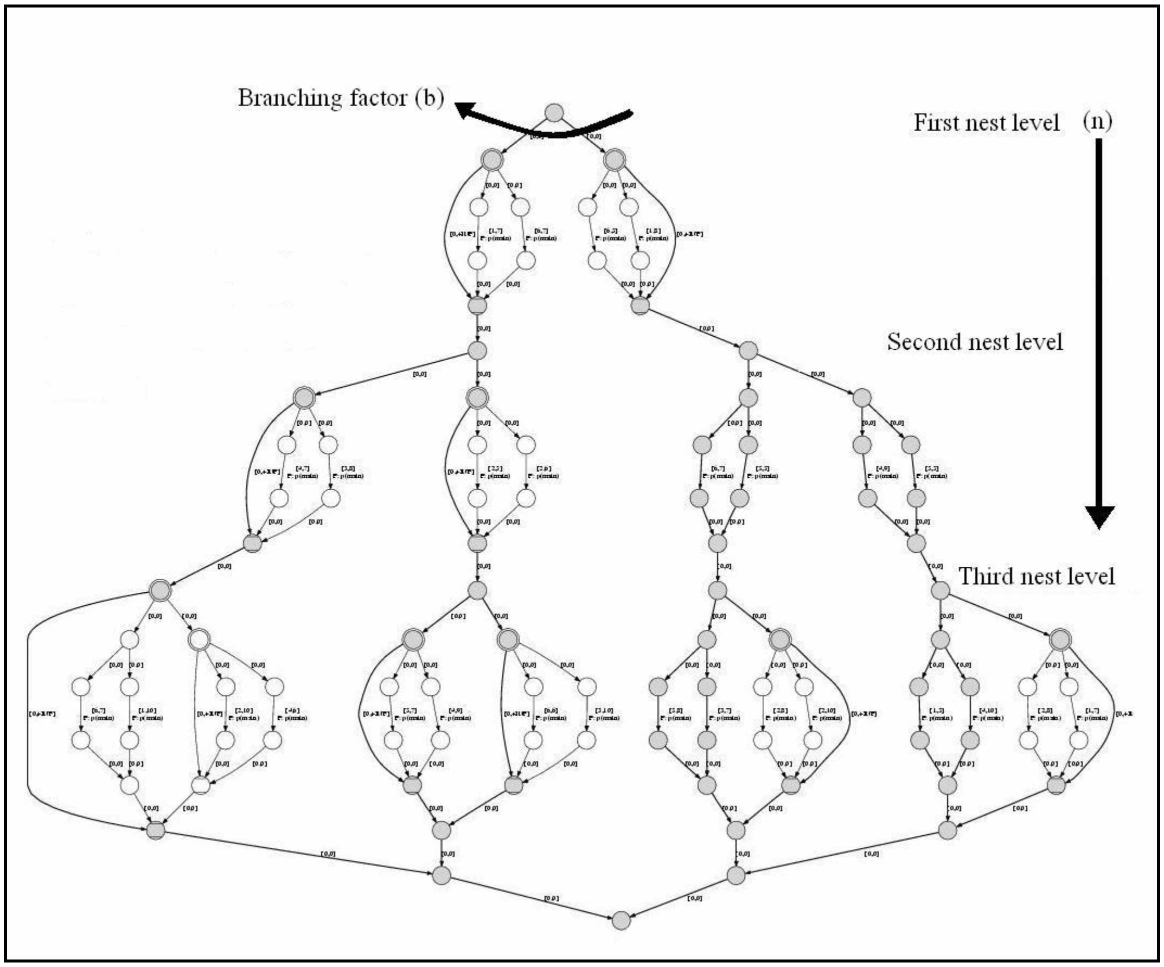


Figure 7.1: Two Dimensions of Complexity in a TPN

7.2.2 The Algorithms

The central contribution of this thesis is a novel algorithm for performing optimal method selection, called Conditional Dynamic Backtracking Branch and Bound (CondDB-B+B). In addition, we develop three novel enhancements that speed up the CondDB-B+B algorithm when applied to TPN search:

- A tight lower bound for Branch and Bound search of TPNs. (Section 6.2)

- A relaxed union operator that enables early detection of temporal conflicts in TPNs. (Section 6.3)
- A method to extract focused temporal conflicts by eliminating irrelevant variable-value assignments. (Section 6.4)

To measure the performance benefits of these enhancements separately, we define two algorithmic variants of the CondDB-B+B algorithm that incorporate these enhancements. Additionally, for benchmarking purposes, we define a standard Branch-and-Bound algorithm augmented to handle conditional variables, called CondBT-B+B. Again, to test each enhancement separately, we define two algorithmic variants to CondBT-B+B. A description of each algorithm is provided below:

1.) CondDB-B+B – This algorithm, as described in Section 6.1, employs the contributions in Chapters 4 and 5 to enable conflict-directed, and memory-bounded candidate execution generation with dynamic variable re-ordering to quickly prune sub-optimal and inconsistent partial executions. In addition, this algorithm stores conflicts, and only generates partial executions that avoid all stored conflicts.

2.) CondDB-B+B_TB – This algorithm extends CondDB-B+B with a method for computing tight lower bounds, as described in Section 6.2, that improves the pruning power of Branch and Bound search of TPNs.

3.) CondDB-B+B_RB – This algorithm extends CondDB-B+B with a relaxed union operator that bounds the feasible durations of choice expressions in a TPN, as described in Section 6.3, and that enables extraction of focused temporal conflicts, as described in Section 6.4. These enhancement enable early and accurate detection of temporal conflicts in a TPN.

4.) CondBT-B+B – This algorithm a standard Branch and Bound search algorithm augmented to handle conditional variables.

5.) CondBT-B+B_TB – This algorithm augments CondBT-B+B with a method for computing tight lower bounds, as described in Section 6.2.

6.) CondBT-B+B_RB – This algorithm augments CondBT-B+B with a method for computing relaxed bounds, as described in Section 6.3.

7.2.3 Empirical Results and Discussion

The speed of optimal method selection in Kirk was tested with six different candidate generation algorithms, as described above in Section 7.2.2, in two separate experiments. The two separate experiments each independently vary a dimension of TPN complexity, as described in Section 7.2.1.

Experiment #1: Varying the Number of Choices in Parallel (b)

For the first experiment, the depth of nested choices (n) was fixed at 4, and the number of choices in parallel (b) was varied from 6 to 20. In addition, the domain size for each choice was fixed at 3, and all activities were assigned a cost of 0. We record the number of partial executions generated by each algorithm before finding an optimal, complete, and consistent execution, while capping individual tests at 5000 candidates generated. The average case performance of each algorithm is presented in Figure 7.2, and the performance of the individual test cases is presented in Figure 7.3.

There are two important trends in the data. Firstly, both of the dynamic backtracking algorithms, CondDB-B+B and CondDB-B+B_RB, significantly outperformed their chronological counterparts, by up to two orders of magnitude. Secondly, the relaxed bounds (RB) enhancements doubled the performance of their respective algorithms. In addition, since all activity costs in this experiment were zero, the tight bounds (TB) enhancement is not applicable, and the performance of the CondDB-B+B_TB and CondBT-B+B_TB would be identical to that of the CondDB-B+B and CondBT-B+B algorithms, respectively. Hence, they were not tested in this experiment.

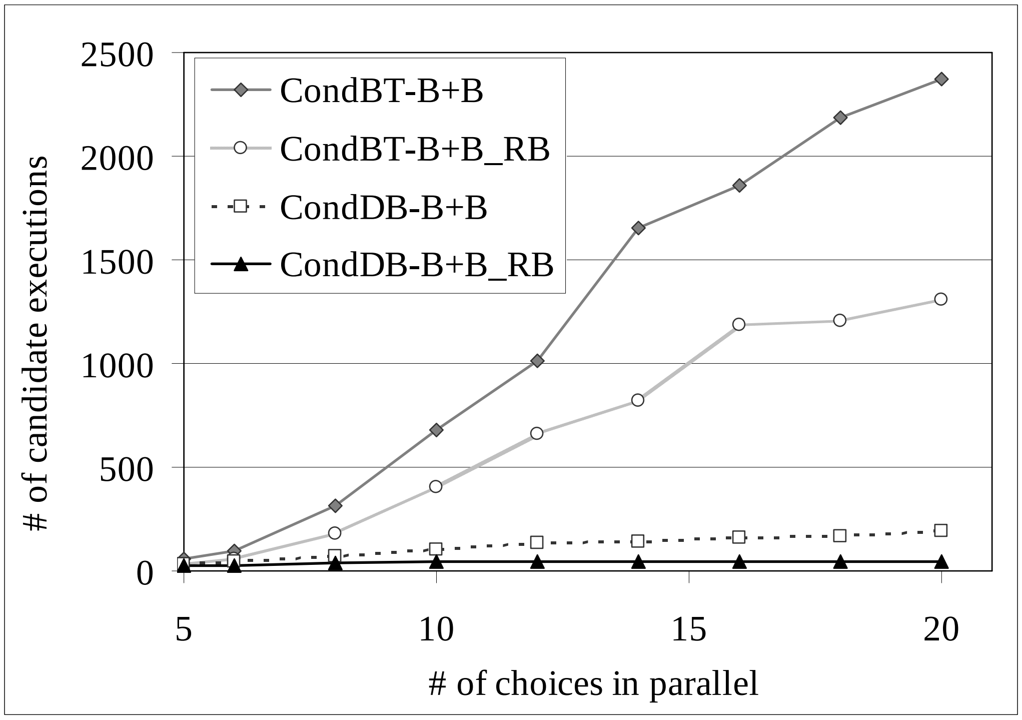


Figure 7.2: Average Case Performance for Experiment #1

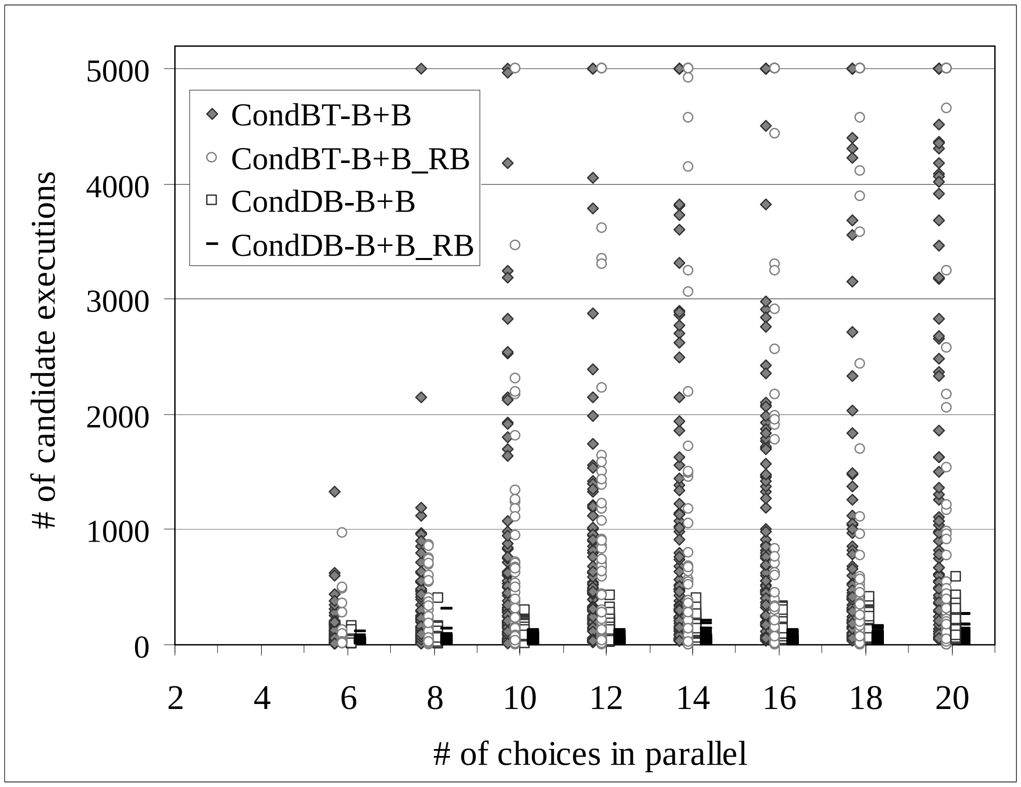


Figure 7.3: Individual Test Cases for # of Choices in Parallel (b)

Experiment #2: Varying the Number of Nested Choices (n)

For the second experiment, the depth of nested choices (n) was varied from 1 to 6, the number of choices in sequence (s) was fixed at 1, and the number of choices in parallel (b) was fixed at 2. In addition, the domain size for each choice was fixed at 3, and the cost of each activity was selected randomly from 1 to 10 with a uniform distribution. In this experiment, the nested choices formed a uniform depth tree of depth (n), so that the number of variables grew at each step by approximately 3^(n) as n varied from 1 to 6. We recorded the number of partial executions generated by each algorithm before finding an optimal, complete, and consistent execution. The average case performance of each algorithm is presented in Figure 7.4, and the performance of the individual test cases is presented in Figure 7.5.

In the data, we see the same general trends as in Experiment #1. Firstly, both of the dynamic backtracking algorithms, CondDB-B+B and CondDB-B+B_TB, significantly outperformed their chronological counterparts. Secondly, this time, the tight bounds (TB) enhancements doubled the performance of their respective algorithms. The relaxed bounds (RB) enhancements were not tested in this experiment.

Another important trend in the data, that is applicable to both experiments, is worth mentioning. In the individual test cases, Figures 7.3 and 7.5, the dynamic backtracking algorithms all have significantly smaller variances than their chronological counterparts. Thus, in addition to being faster at optimal method selection, the dynamic backtracking algorithms are also more predictable. Predictability is an important characteristic for autonomous robots that interact with elderly or injured people, and that operate as part of a larger engineered system.

These empirical results demonstrate that the contributions developed in this thesis significantly improve the speed and predictability at which robots can perform optimal method selection in response to changes in their environment and health status. Implementation details and concluding remarks are provided in the next two sections.

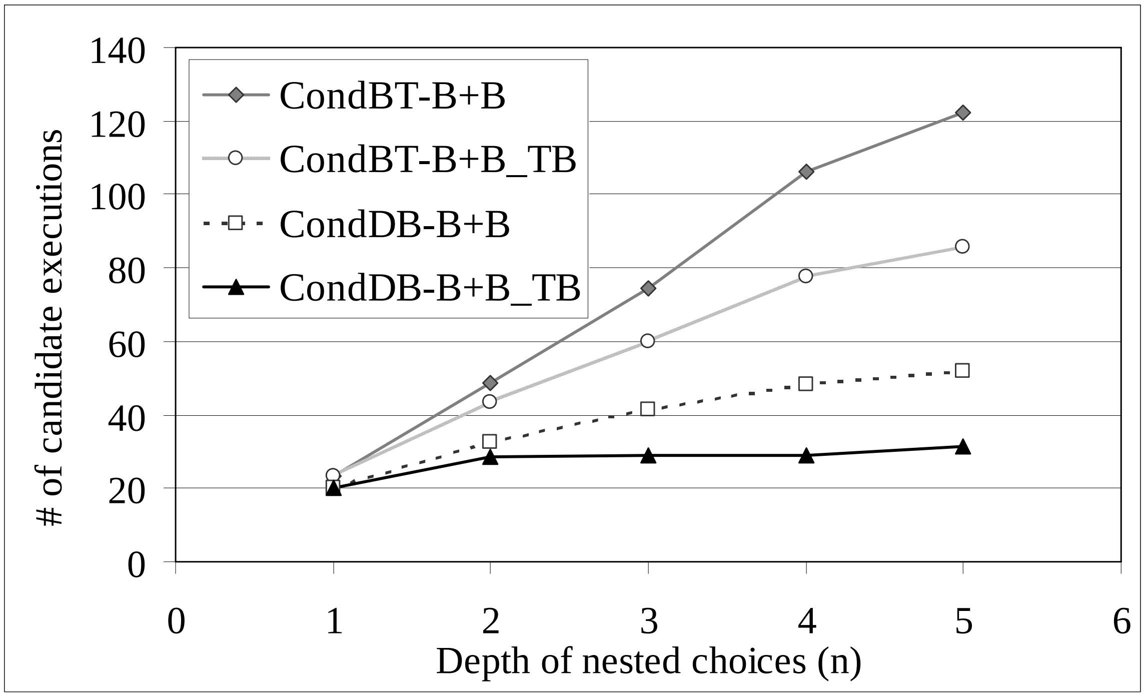


Figure 7.4: Average Test Results for # of Nested Choices (n)

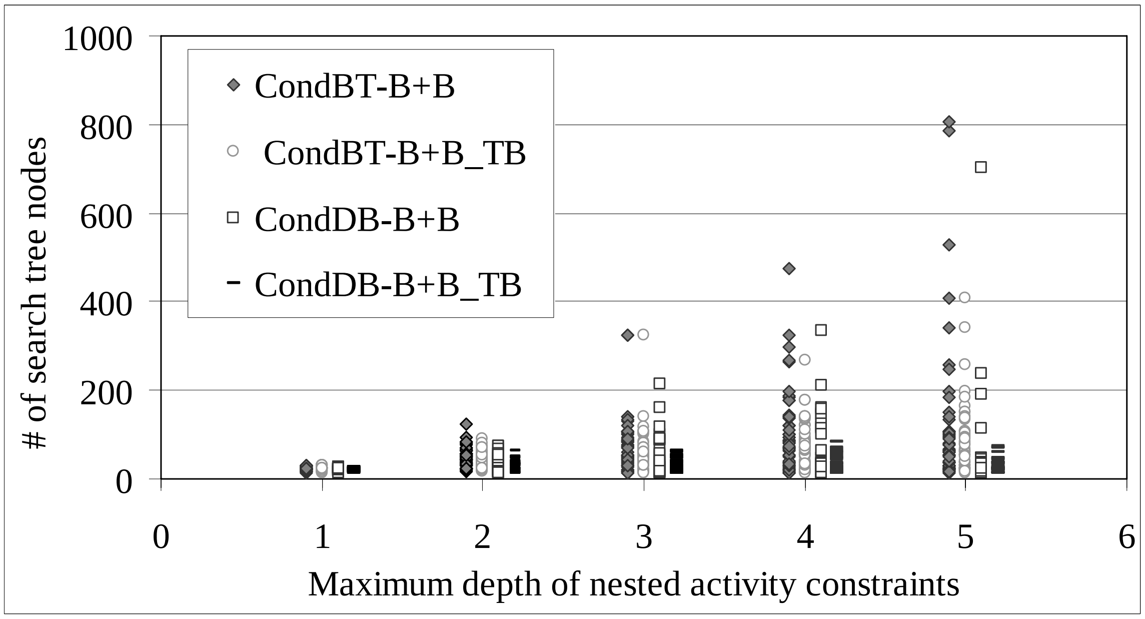


Figure 7.5: Individual Test Results for # of Nested Choices (n)

7.3 Implementation Details

All of the algorithms developed in this thesis were implemented in C++ and integrated into the Kirk temporally-flexible contingent plan executive. The random TPN generator was also implemented in C++. Kirk's main planning loop was augmented so that the user can select which algorithm, CondBT-B+B or CondDB-B+B, is to perform optimal method selection. In addition, the relaxed bounds (RB) and tight bounds (TB) enhancements are presented as options to the user.

Kirk is integrated into a rover test bed of ATRV and ATRV Jr. rovers. To test the algorithms developed in this thesis for correct integration with Kirk, three rover demos were performed using the CondBT-B+B and CondDB-B+B algorithm to perform optimal method selection. The plans executed during these demos were small, with at most a few choices, hence there were no significant differences in runtime performance between algorithms. The performance differences start to become noticeable when there are more than 5 to 10 choices in a plan.

7.4 Future Work

We suggest two directions for future work:

1.) Improving the temporally-flexible contingent planning architecture.

2.) Integrating additional OCCSP search techniques into Kirk.

7.4.1 Improving the Temporally-Flexible Contingent Planning Architecture

In this section, we describe two important and open issues regarding the temporally-flexible contingent planning architecture described in this thesis, which is depicted again in Figure 7.7 for convenience. The two open issues are listed below, and then described in detail in the following two sub-sections:

1.) Incremental plan modification in response to changes

2.) Failure recovery and clean-up

1.) Incremental plan modification in response to changes

As shown in Figure 7.6, each time a change in state occurs, optimal method selection is re-invoked, and a new optimal feasible plan is generated. This approach can become problematic, however, if changes in state occur too often. To deal with frequent state changes, the current architecture needs a mechanism to distinguish between state changes that are important versus changes that are not. In this way, a robot can invoke optimal method selection only when absolutely necessary, minimizing its risk of failing to meet near-term mission objectives or incurring damage from the environment while waiting on a new optimal feasible plan.

Another drawback to the current architecture is that each time optimal method selection is invoked, all conflicts are thrown away, and optimal method selection is started from scratch. The reason for this is that when a state change occurs, some or all of the conflicts may no longer be valid. To retain and utilize valid conflicts, two contributions are needed: 1) A mechanism for incrementally marking conflicts as valid and invalid as state changes occur, and 2) a candidate execution generation algorithm that can handle conflicts dropping out and reappearing seemingly at random to the algorithm.

A third drawback to the current approach is one of inconsistency between

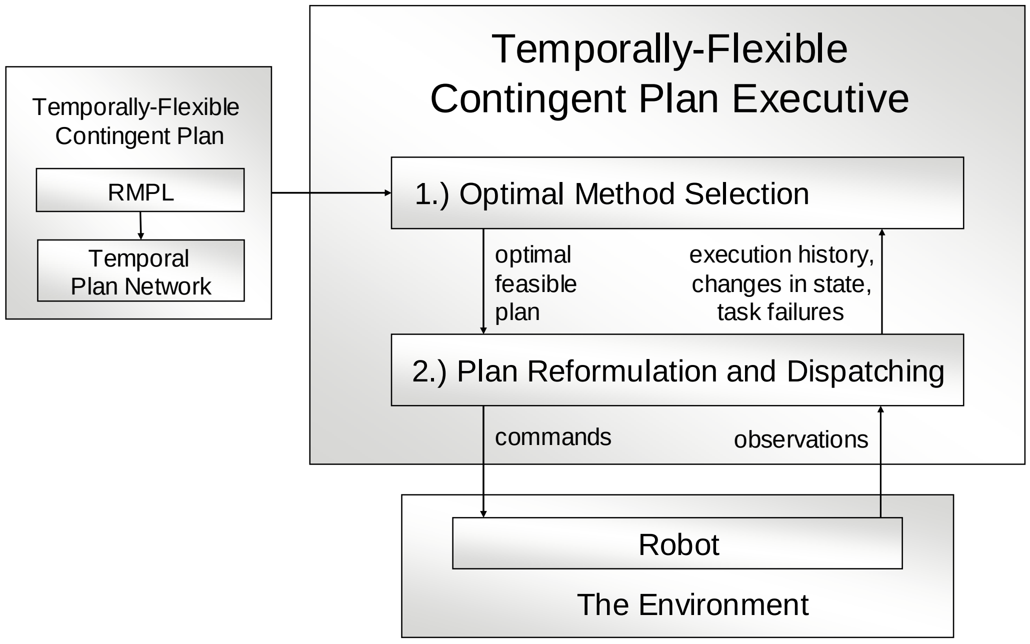

Temporally-Flexible Contingent Plan
RMPL
Temporal Plan Network
Temporally-Flexible Contingent Plan Executive
1.) Optimal Method Selection
optimal feasible plan
execution history, changes in state, task failures
2.) Plan Reformulation and Dispatching
commands
observations
Robot
The Environment

Figure 7.6: Temporally-Flexible Contingent Planning Architecture.

successively generated plans. In the current architecture, there is nothing stopping optimal method selection from gratuitously changing the plan in response to a minor state change. The intuition here is that there may be more value in selecting a new plan that closely resembles the old plan, even if it happens to be slightly more costly than the global optimum. As a suggestion to address this issue we make an interesting connection; with optimal method selection encoded as an OCCSP (Chapter 4) the issue of finding a plan that closely resembles the preceding plan becomes one of finding a stable solution to a dynamic and recurrent CSP [46,47]. We anticipate that past and future advances in this sub-field of constraint satisfaction can be leveraged to improve the temporally-flexible contingent planning architecture presented in Figure 7.6. In fact, one of the most important contributions of this thesis may be to sketch a link between temporally-flexible contingent planning and finding stable solutions to dynamic and recurrent CSPs.

Another suggestion to improve consistency between successively generated plans is to associate a cost with communicating changes in a plan among multiple collaborating robots. If this communication cost is considered as part of the objective function there would be a built-in reluctance to make major changes gratuitously. This makes the most sense in a distributed planning system with multiple robots, and would be an interesting direction for future work.

2.) Failure Recovery and clean-up

Another issue that needs to be addressed is how to recover gracefully from failures that require clean-up. By clean-up, we are referring to activities that need to be performed to adequately recover from a failure. For example, consider the LRV-deployment example from Chapter 1. Consider the first activity in the plan, shown in Figure 7.7a, remove(insulation-blanket). Suppose that if robot #1 fails to remove the blanket, we would like for robot #2 to try to remove the blanket. This can be encoded in a TPN as shown in Figure 7.7b. To incorporate these types of recovery mechanisms into RMPL, a contingency handling framework was devised as a part of the DARPA Self Regenerative

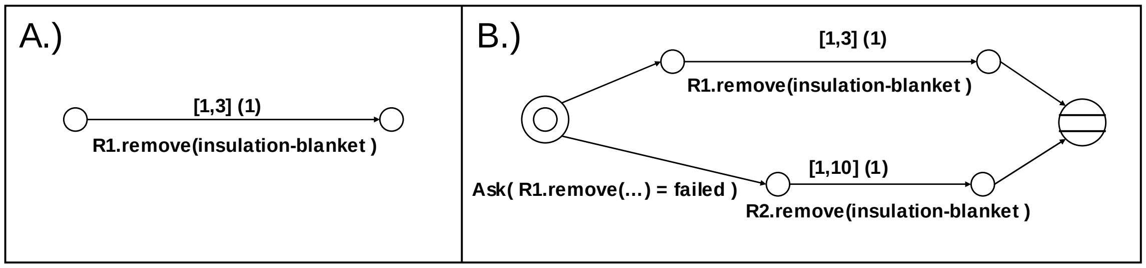

A.)
[1,3] (1)
R1.remove(insulation-blanket)
B.)
[1,3] (1)
R1.remove(insulation-blanket)
[1,10] (1)
Ask(R1.remove(...) = failed)
R2.remove(insulation-blanket)

Figure 7.7: A Contingency Plan for the R1.remove(insulation-blanket) Activity.

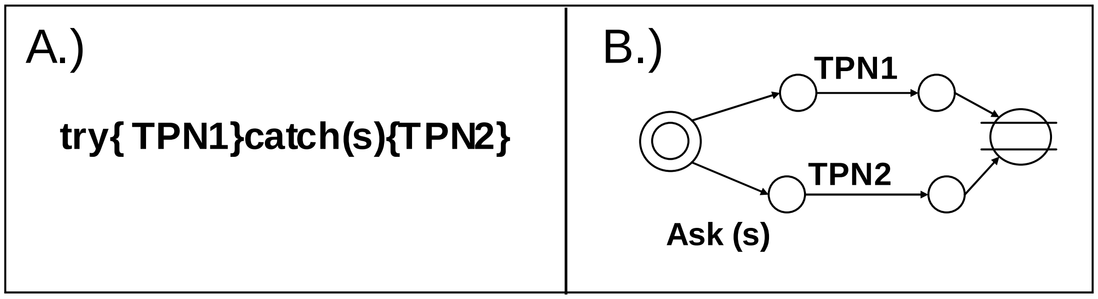


Figure 7.8: Contingency Handling Operator: RMPL to TPN.

Systems program. The basic RMPL construct for contingency handling, and its associated TPN representation are presented in Figures 7.8a and 7.8b, respectively. To fully support this type of construct will require two modifications to Kirk. Firstly, the plan dispatcher needs to be augmented to return failed activity information. For example, if the activity R1.remove(…) fails during execution, the plan dispatcher needs to pass this information back to optimal method selection. Currently, however, the plan dispatcher only monitors timing constraints, so maintenance conditions and explicit task failure information, such as the condition specified in Figure 7.7b, are not supported. In addition, Kirk needs to be augmented so that a thread of execution upon which a failed activity resides cannot be selected again for execution.

7.4.2 Incorporating other OCCSP Search Techniques

Several OCCSP search techniques exist that could potentially improve the speed at which Kirk performs optimal method selection. Keppens [17] defines an A* algorithm and an order of magnitude preference logic for OCCSPs, while Sabin [36] extends forward checking and arc-consistency to OCCSPs. In addition, techniques such as using valued nogoods [6] could be used to improve performance at the expense of more memory.

7.5 Conclusions

This thesis develops a novel temporally-flexible contingent plan executive that selects alternative methods quickly and optimally in response to changes in a robot's health status and environment. To enable fast and optimal method selection, this thesis makes six key contributions:

1.) We frame optimal method selection as an OCCSP. (Chapter 4)

2.) We extend fast CSP search algorithms, such as Dynamic Backtracking and Branch-and-Bound Search, to solve OCCSPs. (Chapter 5)

3.) A candidate execution generator that uses temporally inconsistent and sub-optimal partial executions, called *conflicts*, to guide optimal method selection to an optimal, complete, and consistent execution. (Section 6.1)

4.) A tight lower bound for Branch and Bound search of TPNs. (Section 6.2)

5.) A relaxed union operator that enables early detection of temporal conflicts in TPNs. (Section 6.3)

6.) A method to extract focused temporal conflicts by eliminating irrelevant variable-value assignments. (Section 6.4)

These contributions build upon the ideas of conflict-directed search and optimal heuristic search, which reason on the structure of a problem to guide the search towards an optimal and consistent solution. Experiments on an autonomous rover test-bed and randomly generated plans demonstrate that these contributions significantly improve the speed at which robots can perform optimal method selection in response to changes in their health status and environment.

Bibliography

[1] G. Alefeld and J. Herzberger. "Introduction to Interval Computations", Academic Press, 1983.

[2] A. Baker. Intelligent backtracking on the hardest constraint problems. *Journal of Artificial Intelligence Research*, 1995.

[3] D. Beasley, M. Mathews, and M. Schwager, NASA Announces Telerobotic Construction Competition. *NASA Press Release: 05-417, Dec. 2, 2005.*

[4] A. Cesta and A. Oddi, 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem, *3rd Workshop on Temporal Representation and Reasoning*.

[5] T. Cormen, C. Leiserson and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[6] P. Dago and G.Verfaillie. Nogood Recording for Valued CSPs. In *Proceedings of ICTAI, 1996.*

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

[8] M. Do and S. Khambhampati. Solving planning-graph by compiling it into csp. *In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 2000.

[9] R. Firby. An investigation into reactive planning in complex domains. *Proceedings of the 6th National Conference on AI, Seattle, WA, July 1987,* 1987.

[10] J. Gashnig. Performance Measurement and Analysis of Certain Search Algorithms, Tech. Rept. CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[11] E. Gat. “Esl: A language for supporting robust plan execution in embedded autonomous agents.” *AAAI Fall Symposium: Issues in Plan Execution,* Cambridge, MA, 1996.

[12] E. Gelle and B. Faltings. Solving mixed and conditional constraint satisfaction problems. Constraints, 8(2):107–141, 2003.

[13] M. Ginsberg, Dynamic backtracking, *Journal of Artificial Intelligence Research 1*, 1993, p.25-46.

[14] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *AI*, 14:263–314, 1980.

[15] M. Ingham, R. Ragno and B. Williams, “A Reactive Model-based Programming Language for Robotic Space Explorers,” Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey, Montreal, Canada, June 2001.

[16] J. Kennell. "Generative Temporal Planning with Complex Processes." M. Eng. Thesis, Massachusetts Institute of Technology, October 2003.

[17] Keppens and Shen. Compositional Model Repositories via Dynamic Constraint Satisfaction with Order-of-Magnitude Preferences. JAIR 2004.

[18] P. Kim, B. Williams, and M. Abramson. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. In *Proceedings of IJCAI-2001, Seattle, WA*, 2001.

[19] P. Kim. "Model-based Planning for Coordinated Air Vehicle Missions." M. Eng. Thesis, Massachusetts Institute of Technology, August 2000.

[20] A.Mackworth, Consistency in networks of relations. *Artificial Intelligence 8*, p. 99-118, 1977.

[21] I. Miguel. Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning. PhD diss., The Univ. of Edinburgh, 2001.

[22] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In Proceedings of the 8th National Conference on Artificial Intelligence, pages 25–32. MIT Press, 1990.

[23] M. Moffitt and M. Pollack. Temporal Preference Optimization as Weighted Constraint Satisfaction. In Proceedings of the 21st National Conference on Artificial Intelligence. (AAAI-06), Boston, MA, July 2006.

[24] R. Murphy, J. Casper, and M. Micire. "Potential Tasks and Research Issues for Mobile Robots in RoboCup Rescue." RoboCup Workshop 2000.

[25] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in temporal reasoning for autonomous control systems. In *Autonomous Agents*, 1998.

[26] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proc. Of Sixth Int. Conf. on Principles of Knowledge Representation and Reasoning* (KR '98), 1998.

[27] N. Muscettola, P. Nayak, B. Pell, and B. Williams. August 1998. "Remote Agent: To Boldly Go Where No AI System Has Gone Before." Artificial Intelligence 103(1-2):5-48.

[28] K. Myers. *CPEF: A continuous planning and execution framework*. AI Magazine, 20(4):63-70, 1999.

[29] National Aeronautics and Space Administration. The Vision For Space Exploration, 2004. *NASA Press Release: NP-2004-01-334-HQ, Jan. 14, 2004.*

[30] D. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. "Total-Order Planning with Partially Ordered Subtasks." In *IJCAI-2001*. Seattle, August, 2001.

[31] B. Peintner and M. Pollack 2004. Low-cost addition of preferences to DTPs and TCSPs. In *Proceedings of the 19th National Conference on Artificial Intelligence*, 723–728.

[32] M. Pollack, L. Brown, D. Colbry, C. McCarthy, C. Orosz, B. Peintner, S. Ramakrishnan, and I. Tsamardinos, Autominder: An Intelligent Cognitive Orthotic System for People with Memory Impairment, *Robotics and Autonomous Systems*, 2003.

[33] Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problems. *Comp. Intelligence*, 9(3):268-299.

[34] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS)*, Noordwijk, The Netherlands, June 1999.

[35] M. Sabin and E. C. Freuder. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In Web-published papers of the CP’98 Workshop on Constraint Problem Reformulation, Pisa, Italy, October 1998.

[36] M. Sabin. Towards More Efficient Solution of Conditional Constraint Satisfaction Problems. Ph.D. Thesis, University of New Hampshire, Durham, NH 03824, U.S.A., 2003.

[37] Schiex, T., et. al. 1995. Valued constraint satisfaction problems. In *Proceedings of IJCAI'95*, 631-637.

[38] I. Shu. "Enabling Fast Flexible Planning through Incremental Temporal Reasoning." M. Eng. Thesis, Massachusetts Institute of Technology, September 2003.

[39] Shu, I., Effinger, R., Williams, B., "Enabling Fast Flexible Planning through Incremental Temporal Consistency with Conflict Extraction". *ICAPS*, 2005.

[40] R. Simmons. A task description language for robot control. In proceedings of the Conference on Intelligent Robots and Systems (IROS), Victoria Canada, 1998.

[41] R. Stallman and G. Sussman. *Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis.* Artificial Intelligence, 9:135--196, 1977.

[42] J. Stedl. "Managing Temporal Uncertainty Under Limited Communication: A Formal Model of Tight and Loose Team Communication." S.M. Thesis, Massachusetts Institute of Technology, September 2004.

[43] Titus N., and Ramani K. "Design Space Exploration using Constraint Satisfaction", Configuration Workshop at the 19th International Joint Conference on Artificial Intelligence - IJCAI 2005, Edinburgh, Scotland, July 31st2005, pp.31 – 36.

[44] G. Verfaillie and T. Schiex, Dynamic Backtracking for Dynamic Constraint Satisfaction Problems, In *Proceedings of ECAI*, 1994.

[45] A. Walcott. "Unifying Model-Based Programming and Path Planning Through Optimal Search." S.M. Thesis, Massachusetts Institute of Technology, May 2004.

[46] R. Wallace and E. Freuder, Stable solutions for dynamic constraint satisfaction problems. *In Proc. 4th International Conference on Principles and Practice of Constraint Programming*, Pisa, Italy, 1998, pp. 447-461.

[47] R. Wallace and E. Freuder. Representing and coping with recurrent change in dynamic constraint satisfaction problems. *In CP'99 Workshop on Modelling and Solving Soft Constraints, 1999.*

[48] A. Wehowsky. "Safe Distributed Coordination of Heterogeneous Robots Through Dynamic Simple Temporal Networks." S.M. Thesis, Massachusetts Institute of Technology, May 2003.

[49] B. Williams, V. Gupta. Unifying Model-based and Reactive Programming in a Model-based Executive. *Proceedings of the 10th International Workshop on Principles of Diagnosis,* Scotland, June 1999.

[50] B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems. *Proceedings of IEEE: Special Issue on Modeling and Design of Embedded Software*, 9(1):212-237, 2003.

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

[52] B. Williams, et. al., “Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration.” Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey, Montreal, Canada, June 2001.