Generative Temporal Planning with Complex Processes

by
Jonathan Kennell

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2002

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

October 31, 2003

Author_____
Department of Electrical Engineering and Computer Science
October 31, 2003

Certified by_____
Brian C. Williams
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Generative Temporal Planning with Complex Processes

by

Jonathan Kennell

Submitted to the
Department of Electrical Engineering and Computer Science

October 31, 2003

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Autonomous vehicles are increasingly being used in mission-critical applications, and robust methods are needed for controlling these inherently unreliable and complex systems. This thesis advocates the use of model-based programming, which allows mission designers to program autonomous missions at the level of a coach or wing commander. To support such a system, this thesis presents the Spock generative planner. To generate plans, Spock must be able to piece together vehicle commands and team tactics that have a complex behavior represented by concurrent processes. This is in contrast to traditional planners, whose operators represent simple atomic or durative actions. Spock represents operators using the RMPL language, which describes behaviors using parallel and sequential compositions of state and activity episodes. RMPL is useful for controlling mobile autonomous missions because it allows mission designers to quickly encode expressive activity models using object-oriented design methods and an intuitive set of activity combinators. Spock also is significant in that it uniformly represents operators and plan-space processes in terms of Temporal Plan Networks, which support temporal flexibility for robust plan execution. Finally, Spock is implemented as a forward progression optimal planner that walks monotonically forward through plan processes, closing any open conditions and resolving any conflicts. This thesis describes the Spock algorithm in detail, along with example problems and test results.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor of Aeronautics and Astronautics

Table of Contents

List of Figures

List of Tables

Acknowledgements

The work contained in this thesis could not have been accomplished without the help and support of numerous individuals. First and foremost, my advisor, Professor Brian Williams, has been an invaluable resource. I offer Professor Williams special thanks for not only giving me sage advice, but for providing me the freedom to explore various research topics as a member of the MERS research group.

I would also like to thank our group's administrative assistant, Margaret Yoon, our group's postdocs, Greg Sullivan and Martin Sachenbacher, my fellow graduate students, I-hsiang Shu, Raj Krishnan, Judy Chen, Oliver Martin, Seung Chung, Stanislav Funiak, Tazeen Mahtab, Paul Elliott, Aisha Walcott, Andreas Wehowsky, and John Stedl, and our undergraduate research assistants, Paul Wanda, Adam Kraft, Shen Qu, Radu Raduta, and Rick Sheridan. Many technical revisions were made through our discussions, and I genuinely appreciate your friendship.

Most importantly, I would like to thank my family for their love and support. All my successes are due to the opportunities you have provided me, and I am forever grateful.

This thesis was supported by DARPA under the MICA (Mixed-Initiative Control of Automa-teams) program, contract # N66001-01-C-8075

1 Introduction

Autonomous robots are becoming an increasingly important tool for military, space exploration, and civilian applications. For example, NASA needs autonomous robots as it cannot send human explorers to remote locations in the solar system for safety and financial reasons. Furthermore, it would be advantageous to the military to be able to use expendable robots to help fight wars rather than irreplaceable human beings. In either case, successfully applying robots to achieve mission goals requires a flexible, yet robust control system.

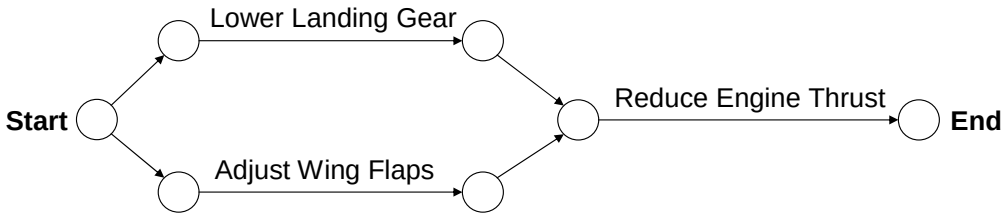


Figure 1-1 Complex Process Example

A key requirement for controlling mobile autonomous robots is the ability to express vehicle activity models as complex processes. For example, an automated landing operator for an unmanned aerial vehicle (UAV) would need to include primitives that lower the landing gear, adjust the wing flaps, and reduce engine thrust, while including timing constraints that ensure that engine thrust is not lowered until after the wing flaps and landing gear are set in place (see Figure 1-1).

To enable generative planning with complex processes, this thesis presents the Spock planner. Spock supports generative planning with complex processes via three key contributions. First, Spock's goal plans and activity models are encoded using the Reactive Model-based Programming Language (RMPL) [32]. RMPL is an innovative way for mission programmers to easily specify control programs and activity operators, because it supports a rich set of intuitive process combinators within an object-oriented framework. Second, Spock represents goal plans, plan operators, and plan candidates with a uniform representation called a Temporal Plan Network (TPN) [19]. TPNs are

significant in that they support temporal flexibility using simple temporal constraints [8], which enable dynamic scheduling and improve mission robustness. Third, Spock is implemented as a forward progression planner. When combined with a relaxed plan-graph heuristic cost estimate, this approach has been shown to support fast planning, which is a requirement for any real-world autonomous control system.

The remainder of this chapter will motivate the development of a model-based executive for mobile autonomous systems, introduce RMPL, and give an overview of the Spock generative planner.

1.1 Motivation

Achieving robust autonomous control is a challenging problem, as autonomous robots typically have hundreds or thousands of interacting components that must be controlled and monitored. To encode the relationships between system components, languages such as RAPS [11], ESL [13], and TDL [29] allow mission designers to program autonomous robots with redundant methods and goal monitoring while simultaneously expressing any necessary constraints between system components.

While these robotic execution languages work well under ideal or anticipated circumstances, a problem arises when unforeseen contingencies occur. Robotic execution languages require mission designers to hierarchically specify all operator sequences and contingencies. If a mission contingency cannot be handled via some expansion of the hierarchy, the system will fail.

Model-based programming was developed to remove dependence on pre-specified monitoring, diagnosis, and operator sequences, and to elevate programming to the specification of state evolutions [33]. In the model-based programming paradigm, a mission programmer commands an autonomous robot in terms of intended state. The specifics of achieving an intended state are delegated to a model-based executive, such as Titan [33] (see Figure 1-2). This separates a programmer's goals from the implementation achieving those goals, removing unnecessary commitments from the

planning process and thus improving the flexibility and robustness with which an autonomous robot may perform its mission.

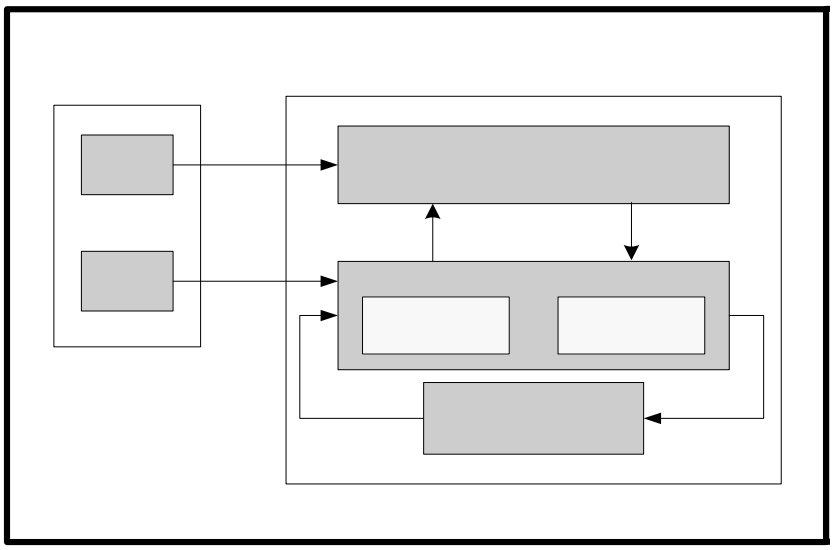


Figure 1-2 Titan Model-based Executive Architecture

The contributions of this thesis are part of a mission-level model-based executive called Kirk [19] (see Figure 1-3). Kirk is designed to control mobile autonomous robots in rich environments, such as rovers exploring the surface of Mars or unmanned aerial vehicles flying search and rescue missions.

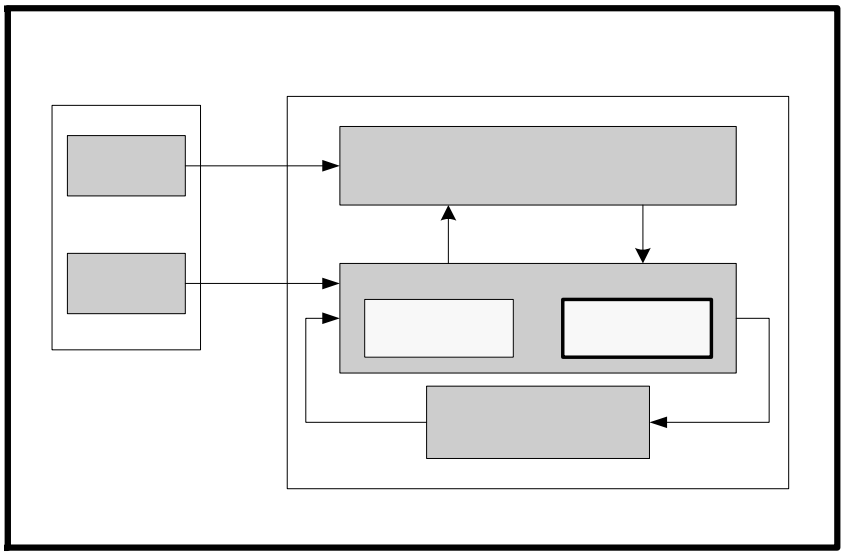


Figure 1-3 Spock Generative Planner within Kirk Model-based Executive Architecture

Mission designers program autonomous missions in Kirk at the level of intended states similar to Titan, rather than at the activity level required by RAPS. Given a goal specification and a set of mission strategies, Kirk will find and execute a safe plan, achieving the goal of robust execution for mobile autonomous robot missions.

To enable model-based programming, Kirk needs to be able to translate the intended state evolutions specified in the control program to an action plan that achieves those state evolutions. This function is provided by the Spock generative temporal planner (see Figure 1-3), and is the central contribution of this thesis.

Spock is unique for three reasons. First, it is able to construct plans by piecing together operators representing complex processes (see Figure 1-1). Second, Spock supports temporal flexibility, which allows activity operators to include durations with lower and upper-bounds. Finally, Spock is significant because it is able to generate optimal plans that minimize various cost metrics such as total mission time or the number of activities in a plan.

An overview of Spock will be given in section 1.3, after we further motivate the model-based programming paradigm through an illustrative example.

1.2 Model-based Programming Example

To demonstrate the idea behind model-based programming, consider the following example. Suppose a family is stranded and needs to be rescued from within a forest (see Figure 1-4). Before a rescue mission can be launched, two threatening forest fires must be put out. The mission commander has three unmanned aerial vehicles (UAVs) at his disposal: one autonomous rescue helicopter, and two unmanned fire-fighting aerial vehicles (UFFAVs).

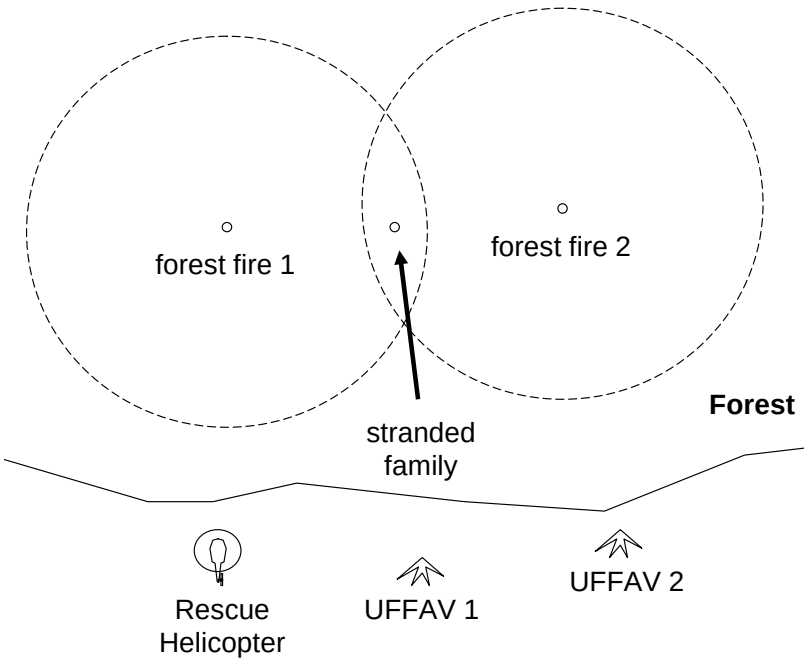


Figure 1-4 Fire Rescue Scenario

Using a robotic execution language like RAPS, the mission commander would write a program that explicitly commands each autonomous vehicle (see left side of Figure 1-5). In particular, the mission commander is responsible for encoding all contingency plans into his control program. For example, if one of the UFFAVs crashes, the program must specify a relevant contingency. If the necessary contingency is omitted by the mission programmer, then the plan will fail.

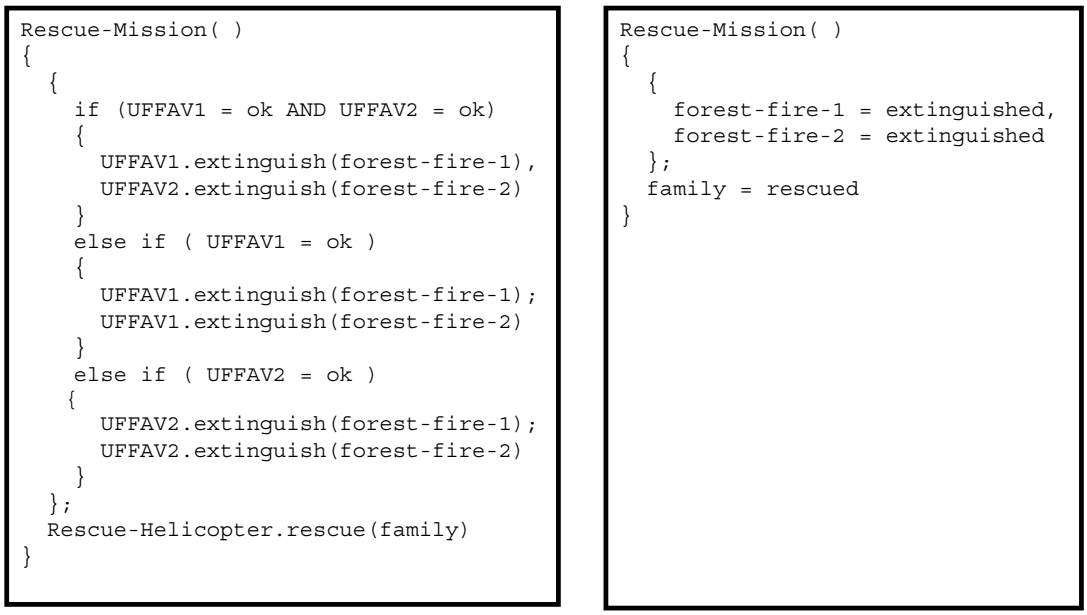


Figure 1-5 RMPL Program Paradigm Comparison

In contrast, in the model-based programming paradigm, the mission programmer only writes a control program with two sequential statements (see right side of Figure 1-5). First, the forest fires should be extinguished. Next, the stranded family should be rescued.

Given this simple control program, the model-based executive determines that it must use the two UFFAVs to extinguish the forest fires, and it must use the autonomous rescue helicopter to rescue the family. It proceeds to automatically generate a plan that applies this knowledge. During mission execution, should one of the UFFAVs crash, the model-based executive can re-examine the goal of extinguishing the forest fires and autonomously derive an alternate plan. For example, if one of the UFFAVs crashes before its target forest fire is extinguished, the model-based executive might generate an alternate plan that instructs the remaining UFFAV to extinguish both forest fires. Because the model-based executive is free to deduce its own planning solutions, unforeseen contingencies will only cause plan failure in the case where no possible solution exists.

1.3 The Spock Generative Planner

To enable model-based programming for mobile autonomous systems, this thesis provides the Spock generative planner. Spock supports generative planning with complex processes through its input language, RMPL [32], a temporally-flexible representation for control programs, operators, and plans called a Temporal Plan Network [19], and an optimal forward progression planning algorithm.

RMPL is an innovative way for mission programmers to specify control programs and activity operators, because it supports a rich set of intuitive process combinators within an object-oriented framework. This approach improves upon other robotic execution languages by allowing mission designers to program in terms of intended state evolutions as opposed to explicit sequences of specific activity operators.

TPNs are significant in that they support temporal flexibility using simple temporal constraints [8], which enable dynamic scheduling and improve mission robustness. This representation supports fast planning as it enables the use of efficient graph-based algorithms for determining plan consistency and cost.

The selection of a forward progression planning architecture is motivated by existing planners such as FF [16] and HSP [6]. These planners have achieved fast generative planning by coupling forward progression planning with a relaxed plan-graph heuristic cost estimate. Furthermore, forward progression planners support optimality metrics, such as time, that are not possible in other planning architectures.

Given an input control program, such as the one on the right side of Figure 1-5, Spock will return an executable solution plan by combining the control program with an environment model and activities from an activity library. To do this, Spock is implemented as a forward progressing optimal planner that walks monotonically forward through plan processes, closing any open conditions and resolving any conflicts.

The rest of this chapter will review in greater detail Spock's three main features: the RMPL control program and activity modeling language, the TPN operator and plan-space representation of processes, and Spock's forward progression planning algorithm.

1.4 The Reactive Model-based Programming Language for Rich Activity Operators and Goal Specifications

An important feature of Spock is that it supports rich activity operators and goal specifications, in order to allow mission designers flexibility in modeling robot behaviors and mission scenarios. This is achieved by building upon the Reactive Model-based Programming Language [32].

To use Spock, a mission programmer writes a control program using the Reactive Model-based Programming Language (RMPL). RMPL uses a process algebra to describe the intended state evolutions of a system similar to executable specification languages like Esterel [4]. The RMPL language allows programmers to specify concurrent processes by combining primitive commands or state assignments using parallel and sequential compositions, non-deterministic choice, pre-emption, and conditional execution.

```
Group-Enroute()[l,u] = {
   choose {
      do {
         Group-Fly-Path(PATH1_1,PATH1_2,PATH1_3,TAI_POS)[l*90%,u*90%];
      } maintaining PATH1_OK,
      do {
         Group-Fly-Path(PATH2_1,PATH2_2,PATH2_3,TAI_POS)[l*90%,u*90%];
      } maintaining PATH2_OK
   };
   {
      Group-Transmit(FAC,ARRIVED_TAI)[0,2],
      do {
         Group-Wait(TAI_HOLD1,TAI_HOLD2)[0,u*10%]
      } watching ENGAGE_OK
   }
}
```

Figure 1-6 Example RMPL Control Program

An example RMPL control program is shown in Figure 1-6. Primitive expressions within RMPL are either command executions (such as Group-Fly-Path) or state assertions (such as PATH1_OK). Expressions are combined, as demonstrated in the example, using non-deterministic choice (choose), parallel composition (,), sequential composition (;), activity time-bounds ([l,u]), and pre-emption (do-maintaining and do-watching). RMPL supports modularity through inheritance, encapsulation, and abstraction via adherence to the object-oriented paradigm.

In the context of this thesis, RMPL is innovative in that it is used to describe complex processes that are the operators in a planning problem. This allows the Spock planner to incorporate rich activities that represent real-world behaviors, such as the automated landing operator in Figure 1-1.

Finally, RMPL is a language for describing both control programs and activity operators. The full RMPL syntax is supported for Kirk's control programs. However, because Spock is not a conditional planner, Spock's activity operators use a subset of the RMPL language that includes activity timing, state assertion, sequential composition, parallel composition, and the do-maintaining combinator. This subset omits conditional statements such as non-deterministic choice and the if-then combinator. However, the resulting execution language is still sufficiently expressive for representing complex processes within Spock's activity operators.

1.5 Flexible Time-bounds and Temporal Plan Networks

An important feature of Spock is that the plans it produces are temporally flexible. While some temporal planners use activities with fixed durations, a planner that supports temporal flexibility uses time-bounds that express a range of durations, which is an essential trait in the context of real-world applications.

Flexible time-bounds are motivated by the fact that the real-world does not usually go according to schedule. If an activity is supposed to take 10 minutes, it may actually finish in 9 minutes or in 11 minutes. Planners that rely on rigid schedules must either

include slack to compensate for these unexpected occurrences, or must re-plan frequently during mission execution.

In contrast, by using a plan representation that supports flexible time-bounds, Spock can schedule mission activities dynamically, eliminating the need for plan slack or excessive replanning. Spock's internal plan representation that supports temporal flexibility is called the Temporal Plan Network (TPN) [19], and is a central contribution of this thesis.

A Temporal Plan Network [19] is a graphical depiction of a process representing plans in plan-space. When a mission programmer finishes writing an RMPL control program, the program is converted into a Temporal Plan Network, which is a graph that represents the space of possible concurrent threads of execution specified within the RMPL control program. The model-based executive then operates on this TPN during the planning process.

TPNs are superior to other temporally-flexible plan representations, such as timelines, because their graph-based representation enables the use of fast network algorithms that efficiently evaluate plan consistency and correctness, as well as perform dynamic scheduling.

As TPNs are a compact representation of the data contained in an RMPL program, all of the constraints, primitive activities, and open conditions expressed in an RMPL program have a direct mapping when encoded as a TPN.

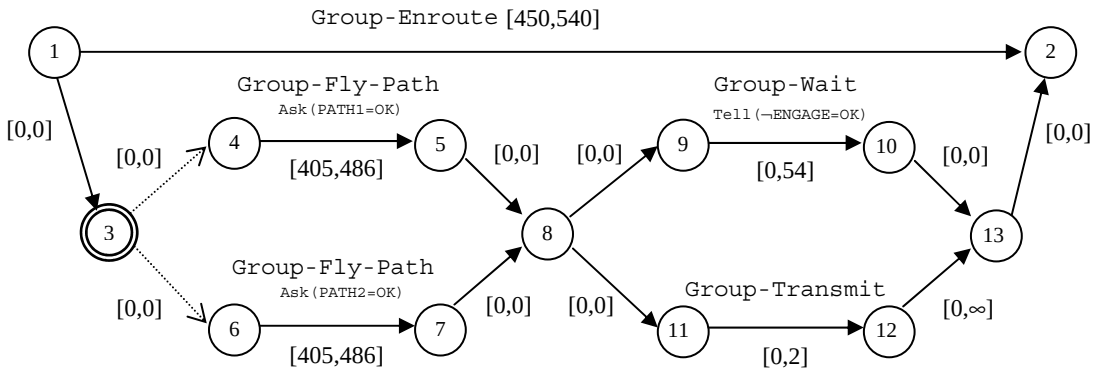


Figure 1-7 Example Temporal Plan Network

An example TPN corresponding to the RMPL control program example shown in Figure 1-6 is shown in Figure 1-7. In this example, the non-deterministic choice is represented by Node 3, while parallel composition is demonstrated at nodes 1 and 8. All arcs in this TPN are labeled with lower and upper time-bounds.

As a temporal plan representation, nodes in a TPN represent events in time, while arcs correspond to episodes (or intervals) between those events. The episodes in a TPN have simple temporal constraints bounding their duration, allowing activities to have flexible durations. Additionally, TPN episodes contain primitive actions as well as state assignments in the form of Ask and Tell constraints, which are used in TPN planning to represent open conditions and the activity effects that close those open conditions, respectively.

Finally, TPNs add support for decision nodes, which allow the network to express non-deterministic choice as part of the plan-space representation. When Kirk's strategy selection algorithm searches a TPN for a consistent sub-graph to return as its solution plan, it is searching over the space of choices among these decision nodes. While Kirk's strategy selection algorithm supports the full TPN specification, Spock itself does not support decision nodes, as it does not perform conditional planning. However, Spock utilizes the rest of the TPN constructs in order to create a uniform representation for its control programs, activity operators, and internal plan candidates.

1.6 Forward Progression Planning

Spock is an optimal forward progression planner. When given a goal plan in the form of a control program TPN, Spock walks monotonically forward through the TPN processes, closing any open conditions (Asks) and resolving any conflicts (mutually exclusive Tells). This approach is appropriate for model-based programming of mobile autonomous systems, because it plans optimally, generatively, and within a framework that supports fast planning. The remainder of this section will argue the importance of these three features, and motivate the selection of Spock's forward progression architecture.

Optimality is essential when controlling real-world autonomous systems. For example, a Mars observation satellite has a limited amount of fuel with which it can guide its orbital insertion. If a control system executes a sub-optimal orbital insertion plan, it may use too much fuel. If the plan uses all of the available fuel, this could cause orbital insertion to fail, resulting in vehicle loss. At best, a sub-optimal plan will result in a shorter mission duration, because fuel is consumed that would otherwise have been used to maintain orbit throughout the science mission.

Because optimality is so important, any planner that is to be used to control real hardware must be capable of finding globally optimal solutions over various cost metrics, such as total mission time and resource consumption. This motivates Spock's use of a forward progression planning architecture, as graph-based planners can only optimize for the total number of plan steps (or layers) [21].

Hierarchical task network planners can perform optimal fast planning using temporally-flexible operators [26]. However, these planners rely on explicit hierarchical expansion rules that must be encoded by a mission programmer. The model-based programming paradigm avoids pre-compiled expansion rules, and thus a generative planning architecture like forward progression is needed.

Finally, to be credible and useful in real-world applications, a planner must be able to quickly solve reasonably sized problems. This also motivates Spock's use of a forward

progression planning architecture, as forward progression search combined with informative heuristic cost estimates has recently been shown to be an effective way to achieve fast planning [6][14][16].

In the style of FF [16] and HSP [6], Spock has the capability of including a relaxed plan heuristic cost estimate that will upgrade its search algorithm from a uniform-cost search to a much faster informed search. Given a plan candidate, relaxed plan graphs that reach the goal can be constructed in polynomial time using simplified constraint rules. These relaxed graphs serve as an admissible estimate of the remaining cost in a candidate plan, allowing a planner's search algorithm to focus on plan candidates that are more likely to yield optimal solutions.

In the future work section, this thesis describes a possible relaxed plan heuristic cost function that should accelerate Spock's search algorithm to achieve fast planning.

Spock performs generative temporal planning using a forward progression planning algorithm. This approach is motivated by the need for Spock to be fast, optimal, and generative. These capabilities are uniquely provided by the forward progression family of planning algorithms, justifying the choice of Spock's architecture.

1.7 Thesis Layout

This thesis first presents in Chapter 2 a brief background of the field of planning. Next, Chapter 3 introduces Spock's input language, the Reactive Model-based Programming Language. Chapter 4 describes Temporal Plan Networks, which are the graphs corresponding to RMPL code and serve as Spock's control program, activity, and plan candidate representation. Chapter 5 explains the Spock generative planning algorithm in detail, including several examples. The final chapter describes the implementation and provides test results of Spock's performance, along with concluding remarks and a discussion of future work.

2 Related Work

The planner described in this thesis builds upon the fields of constraint-based interval planning and forward progression planning. Furthermore, as Spock was being designed, various alternative methods for achieving fast planning were evaluated, including hierarchical task network planning and graph-based planning. This chapter describes these various approaches to planning and explains the decision to use a heuristic-guided forward progression design.

2.1 Constraint-based Interval Planning

Spock's internal plan representation, the Temporal Plan Network (TPN), inherits from constraint-based interval plan representations [30]. Similar to constraint-based interval plans, a TPN contains episodes of state assignments that have interval durations with flexible time-bounds. However, TPNs differ with regard to how these episodes are combined to describe complex processes. This section gives an overview of constraint-based interval planning, and highlights the specific features incorporated by the Spock generative planner along with essential differences.

Planning for real-world systems requires using a realistic representation of time. Constraint-based interval planners address this need by using plan actions with interval durations. To this rich notion of time, constraint-based interval planners add constraints between action intervals that allow the expression of mutual exclusion relationships as well as preconditions that must hold before, during, or after a particular action interval [30].

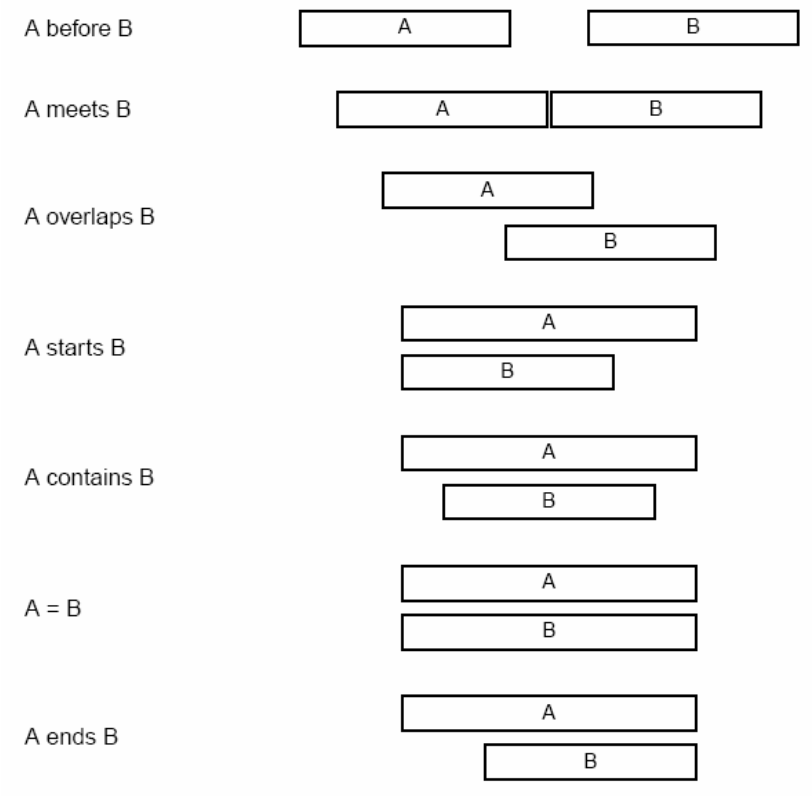


Figure 2-1 Allen's Interval Relationships [1][30]

Intervals within a constraint-based interval planner are often ordered using Allen's basic interval relationships: before, meets, overlaps, starts, contains, equals, and ends [1] (see Figure 2-1). These relationships are used by a planner to constrain the execution of two related actions to ensure that open conditions are satisfied, or that conflicting intervals do not co-occur. Furthermore, Allen's relationships are used when a programmer writes an activity model to describe complex interactions within system processes.

Constraint-based interval planners, such as HSTS [17], usually plan using a goal-directed search. Planning begins with an initial plan that contains open conditions. The planner closes those open conditions by adding actions from its action library. As each action is added to the plan, threat resolution ensures that any conflicting state assignments do not co-occur. When all of the open conditions in a plan have been closed, the planner returns the plan as a solution.

In a constraint-based interval plan, the duration of an action is specified with temporal flexibility through an upper and lower time-bound. To check for conflicts among an interval plan's temporal constraints, the start and end-points for each interval in the plan are represented with variables that can be constrained using the interval durations embedded in the plan [30]. These constraints are represented using a constraint network, such as a Simple Temporal Network [8] or distance graph [2], which allows consistency to be checked using efficient graph-based algorithms [8]. Spock uses a similar temporal representation in terms of Simple Temporal Networks [8].

Constraint-based interval planners usually describe concurrent processes through a fixed set of timelines. We instead build these processes through a process algebra, which allows processes to naturally fork and recombine. Constraint-based interval planners also include a representation for describing continuous resource utilization. However, this falls outside the scope of Spock.

2.2 Hierarchical Task Network Planning

While designing Spock's planning algorithm, several architectures were considered. This section will discuss hierarchical task networks, and explain why this design pattern was not selected for Spock's planning algorithm.

All planners attempt to achieve fast planning by reducing the amount of search space that is explored. Hierarchical task network (HTN) planners increase speed by searching a plan-space that is restricted to plan candidates which are guaranteed to be complete. While this limits their flexibility, it also makes them fast by eliminating a large portion of the search space. Examples of HTN planners include SHOP2 [24], Aspen [25], and Kirk's strategy selection algorithm [19].

When using an HTN planner, a programmer uses a library of macro operators, which can be decomposed into other macros, primitive operators, or some combination of the two. Additionally, there may be a choice between several alternative decompositions of a single macro operator, which introduces a non-deterministic branch and a need for a search component.

In HTN planning, mission programmers initiate the planning process after specifying an initial plan. The initial plan contains macros that need to be decomposed by the HTN planner using the macro library. When an HTN planner has decomposed all the macros from the control program into consistent primitive operators, planning is complete.

While HTN planners can be very efficient, their reliance on pre-specified macro decompositions limits their flexibility and puts additional programming demands on the mission designer. In the spirit of model-based programming, Spock should be able deduce solution plans without pre-specified rules. This requirement motivates the use of a generative planning approach rather than one of task decomposition.

2.3 Graph-based Planning

As opposed to HTN planning, generative planning solves a planning problem by combining a set of plan actions to achieve the planning goals. This section will discuss graph-based planning, which is one of today's leading architectures for solving generative planning problems.

Graph-based planners, such as Graphplan [5], Blackbox [18], and LPGP [21], all utilize a structure called a plan-graph. Plan-graphs compactly represent the plan-space for a given planning problem, allowing graph-based planners to solve planning problems without exploring the entire space of plan candidates (see Figure 2-2).

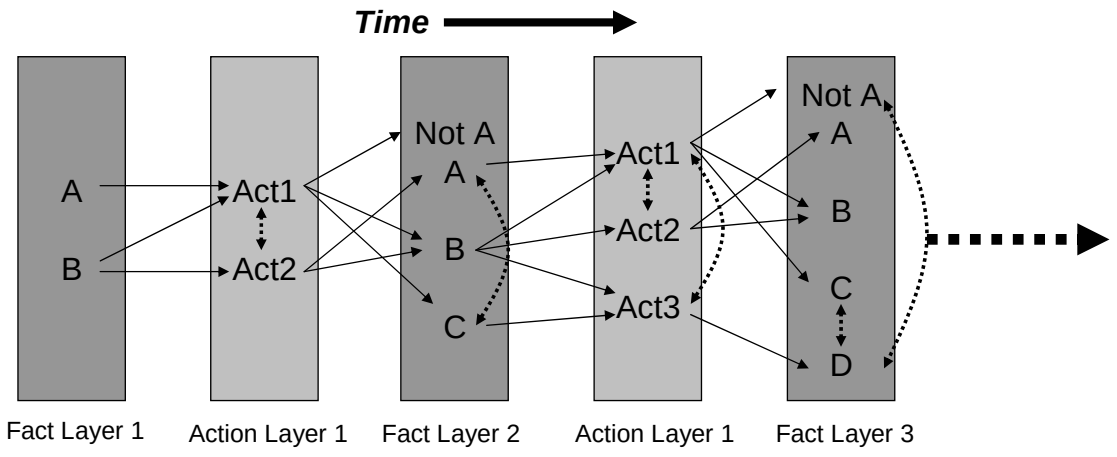


Figure 2-2 A Plan Graph

A plan-graph contains alternating fact and action layers, increasing with time. The facts in a given fact layer represent an upper bound on the set of all facts that could, in theory, be achieved at the time of that fact layer. That is, if a fact is not included in a particular fact layer, it is not attainable by the corresponding point in time.

Plan-graphs also track mutual exclusion relationships (or conflicts) among the facts in each fact layer. While each fact in a given fact layer can be achieved via some path in the plan-graph, each mutual exclusion relationship indicates that two facts cannot be achieved simultaneously without violating plan consistency and completeness. A graph-based planner therefore knows that it should only search its plan-graph to find a solution when all of the goals in the plan-graph become pair-wise consistent. This is how graph-based planners achieve their speed: they avoid searching the subset of the plan-graph where the goals cannot be simultaneously achieved.

Graph-based planners perform very well when the facts in a planning problem are mutually exclusive on a pair-wise basis. This is because plan-graphs only keep track of mutual exclusion relationships between pairs of facts. However, sometimes facts are consistent on a pair-wise basis, but mutually exclusive in larger groupings. For example, a robot with two arms may be able to move any two objects in one time-step, but cannot move a group of three or more objects in a single time-step. In this case, the planner begins searching the plan-graph before a solution exists. When it discovers that no

solution exists in the plan-graph, the planner adds additional fact and action layers to the plan-graph, and continues its search.

When facts in a planning problem are mutually exclusive in triples or larger groupings, a plan-graph has no ability to predict the existence of a complete solution plan. Thus, the planner becomes less efficient, as it searches regions of the plan-space that do not contain a solution.

Another concern of graph-based planners is their limitation regarding optimality. Graph-based planners return the first plan they discover that achieves their planning goals. Because they search in increasing order of plan length, plans with fewer actions are preferred. Thus, when cost metrics involve resources other than action quantity, plan-graph planners cannot perform optimal planning.

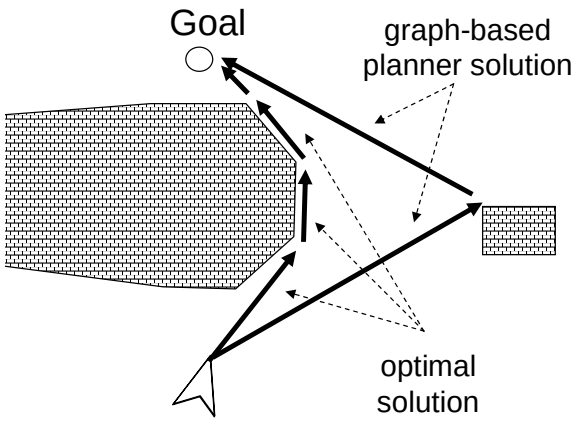


Figure 2-3 Example of Graph-based Planner Sub-optimality

An example of graph-based planner sub-optimality is shown in Figure 2-3. In this scenario, a vehicle is instructed to move to a goal waypoint while avoiding obstacles in the environment. In this domain, all movements must be between obstacle vertices or scenario waypoints along straight line trajectories. For this problem, the optimal solution involves four move commands, ordering the vehicle to trace the side of the large obstacle. Unfortunately, a graph-based planner minimizes the number of layers in a plan, returning a sub-optimal solution because it requires only two move commands.

While graph-based planners do provide the generative search property that Spock needs, they have significant limitations regarding optimal planning. As Spock is intended for use on real-world problems involving expensive robot hardware, optimal planning over metrics such as time is an important feature that must be included. This leads us to conclude that the graph-based planning architecture is insufficient for satisfying Spock's feature specification.

2.4 Forward Progression Planning

We have discussed non-generative planning architectures, as well as generative architectures with limited support for optimal planning. Now we will discuss forward progression planning, which supports optimal planning as well as generative planning.

Forward progression planners and backward propagation planners both perform a search over the entire plan-space. Forward progression planners begin at some initial state and search towards the goal state, while backward propagation planners begin at the goal and search towards the initial state. These approaches allow for expressive plan actions and have the ability to plan optimally for arbitrary cost metrics, however, they are also inherently slower than HTN or graph-based planners.

One way of optimizing forward chaining planners is to use expansion rules, as demonstrated by TLPlan [3]. Expansion rules inform the planner such that it avoids searching redundant or wasteful candidate solutions, thus reducing the search branching factor and increasing planning speed. Unfortunately, these expansion rules are inappropriate for Spock as they violate the spirit of model-based programming.

Recently, some forward progression planners, such as FF [16] and HSP [6], have shown dramatic performance improvements by using relaxed plan-graphs to calculate admissible heuristic cost estimates. A relaxed plan-graph is constructed in a manner similar to a plan-graph, except that mutual exclusions are ignored. This property allows the relaxed plan-graph to act as an admissible heuristic estimate when trying to determine the cost to the goal for a particular planning state.

With the relaxed plan-graph heuristic cost estimate, a forward progression planner uses an informed search process, as opposed to a uniform cost search process. This improves planner efficiency by focusing the search toward solution states, thus reducing the number of states that must be explored in a given planning problem. Spock's design as a forward progression planner was chosen with the intent to eventually utilize relaxed plan-graph heuristic cost estimates as a mechanism for achieving fast planning.

Finally, another method of achieving fast planning when using a forward progression plan representation is through local search. While local-search or repair-based planners do not use a forward progression planning algorithm, they generally operate on plan representations similar to those used in forward progression planning. An example of a local-search planner is LPG [14]. LPG plans by using a randomized local search algorithm similar to WalkSAT [27], called WalkPlan. LPG is quite fast, however, its randomized search means that it is not optimal, often returning plans with obviously wasteful sub-sequences.

Spock's planning algorithm is implemented using a forward progression design. As described in this chapter, forward progression planners are generative, enabling Spock to automatically deduce command sequences that achieve a programmer's mission goals. Additionally, forward progression planners support optimal planning for various cost metrics, and can be accelerated via relaxed-graph cost heuristic estimates.

3 The Reactive Model-based Programming Language

Controlling complex autonomous systems is a difficult task. Autonomous aerial vehicles and robotic spacecraft can have thousands of hardware components, each of which needs to be monitored or controlled at all times. To help manage the inherent complexity of autonomous systems control, mission programmers have traditionally relied on programming languages such as RAPS [11], ESL [13], and TDL [29]. These languages help model the relationships between various robot states by incorporating features such as concurrency, metric constraints and durations, functionally redundant choice, contingencies, and synchronization. The benefit of these languages is that they allow mission programmers to create models of autonomous systems that accurately reflect the hardware being controlled.

While existing languages have proven to be useful through their ability to model the activities of real-world autonomous systems, they do little to address the massive complexity inherent in such devices. A modern spacecraft or unmanned aerial vehicle has hundreds of sensors and actuators, all of which must be constantly monitored or commanded. Because of this large number of inter-dependent variables, managing the complexity of these systems is quite similar to managing the complexity of a modern software project. As such, a robotic execution language that includes features of modern programming languages, such as abstraction, inheritance, and encapsulation, is needed to ensure that vehicle models can be programmed quickly with minimal human error. To meet this demand, we introduce the Reactive Model-based Programming Language.

RMPL is a rich language for describing activity models of autonomous reactive systems [32]. Designed to help manage complexity, RMPL is object-oriented and supports high-level programming features such as abstraction, encapsulation, and inheritance. Moreover, RMPL is a process algebra that includes combinators supporting concurrency, pre-emption, conditional execution, non-deterministic choice, maintenance conditions, state assertion, and activity timing. These combinators make RMPL programming quick and easy, while still allowing the expression of all desired constraints.

This chapter first provides a brief overview of RMPL and its syntax. Next an example will be shown, followed by an in-depth discussion of RMPL's combinators and its role in the Spock generative planner.

3.1 RMPL Overview

The Reactive Model-based Programming Language, RMPL, is a high-level language used to describe activity models of autonomous reactive systems. To support encapsulation and abstraction, RMPL is object-oriented, and thus RMPL code is contained in object methods with the following structure:

```
Method-Name(arguments){method body}
```

All RMPL methods have a name, as well as two important specification sections: the arguments list and the method body.

As required by any functional programming language, the arguments list in an RMPL method contains variables that the method body uses to customize its behavior. For example, a Move method might take a start and end position as arguments, allowing the method to determine the proper trajectory and temporal bounds for the specified move activity.

The RMPL method body is coded using a process algebra consisting of a set of combinators that supports conditional execution, concurrency, pre-emption, maintenance conditions, state assertion, activity timing, and non-deterministic choice (not all of which are supported by Spock).

3.2 Example Scenario with RMPL Program

To illustrate the combinators in RMPL, we present the following scenario. A family hiking in the woods is threatened by a nearby forest fire. The decision is made to send an autonomous rescue helicopter to recover the family. Simultaneously, another autonomous helicopter will be sent to fight the forest fire. For safety purposes, the family

should only be rescued after the nearby flames have been extinguished. We can encode this scenario with the RMPL code in Figure 3-1.

```
Rescue-Helicopter.Retrieve(group g) // activity 1
{ // activity / method body
  do pickup(g) maintaining { threat = low } [300,+INF];
  g = safe
}

Fire-Helicopter.Extinguish-Fire(location loc) // activity 2
{ // activity / method body
  do {
    if (retardant = present) then
      drop-retardant()
    else
      call-for-assistance()
  } watching { fire = controlled };
  threat = low
}

Rescue-Family() // control program
{ // method body
  { // thread 1
    Rescue-Helicopter.fly-to(rescue-point);
    Rescue-Helicopter.Retrieve(family)[400,500];
    Rescue-Helicopter.fly-to(hospital);
  },
  { // thread 2
    Fire-Helicopter.fly-to(forest-fire);
    Fire-Helicopter.Extinguish-Fire(forest-fire)[300,400];
    Fire-Helicopter.fly-to(base);
  },
  [0,1200]
}
```

Figure 3-1 Example RMPL Program

This example contains three RMPL methods: two macro activity declarations (Rescue-Helicopter.Retrieve and Fire-Helicopter.Extinguish-Fire), and a top-level program (Rescue-Family). The macro activity declarations are high-level methods that are called by the top-level program, while the other methods referenced in the RMPL code (in lowercase) are primitive activities understood by the system executive.

The Rescue-Helicopter.Retrieve activity method demonstrates "do-maintaining" maintenance conditions, sequential composition, and episode timing. The first statement in the method body, `"do pickup(g) maintaining { threat = low } [300,+INF],"` executes the pickup primitive activity for at least 300 seconds, given that the threat condition remains low. This statement is sequentially combined with the state assertion, `"g = safe,"` which asserts that the group being rescued, g, is indefinitely safe once the pickup activity is complete.

The next activity method, Fire-Helicopter.Extinguish-Fire, demonstrates do-watching maintenance conditions, sequential composition, and conditional execution. The first root-level statement in the method body, `"do {…} watching { fire = controlled },"` instructs the system to fight the fire until the fire is under control. The interior of this statement, `"if (retardant = present) then drop-retardant() else call-for-assitance(),"` tells the system how to fight the fire. Specifically, it says to drop retardant on the fire if possible, and otherwise call for help when retardant is not available. This complex statement is combined using sequential composition with the goal state assertion, `"threat = low,"` which informs the system that the environment is safe once the fire has been extinguished.

This example also includes a top-level program, "Rescue-Family," which is the primary method that directs the execution of the rescue mission. The top-level program demonstrates sequential and parallel composition, macro activity calls, and episode timing.

The body of the "Rescue-Family" method contains two parallel threads of execution that are both constrained to take no more than 1200 seconds to execute. The first sequence commands the rescue helicopter to fly to the rescue point, retrieve the family in 400-500 seconds, and finally fly to the hospital to drop off any injured people. The second sequence commands the fire helicopter to fly to the forest fire, extinguish it in 300-400 seconds, and then return to base.

3.3 RMPL Combinators

This section presents each RMPL combinator and describes its semantics. The list of such combinators is shown in Figure 3-2.

```
A := A [l,u] |
     c |
     A; A' |
     A, A' |
     { A } |
     if c then A |
     when c then A |
     do A maintaining c |
     do A watching c |
     choose { A, A', … }
c := assignment to state variable
```

Figure 3-2 RMPL Combinators Supported by Kirk's Strategy Selection Algorithm

Note that Spock only supports a subset of RMPL, as it does not allow conditional plan operators. The subset of RMPL combinators supported by Spock is thus listed in Figure 3-3

```
A := A [l,u] |
     c |
     A; A' |
     A, A' |
     { A } |
     do A maintaining c
c := assignment to state variable
```

Figure 3-3 RMPL Combinators Supported by Spock

3.3.1 Episode Timing - A [l,u]

Given an RMPL sub-activity, A, the statement A[l,u] informs the executive that the episode, or interval, during which the activity occurs must take at least l time-units and

no more than u time-units. This construct can be used to constrain the durations of activity episodes, or the episodes between activities.

Note that, by default, an episode has time-bounds of [0,+INF]. Moreover, if an episode is constrained by more than one set of time-bounds, the intersection of those bounds is used.

3.3.2 State Assertion - c

RMPL is a language for interacting with hidden state. Thus, it needs a mechanism for asserting assignments to state variables. This mechanism is state assertion. Within RMPL activity code, a programmer can assert the value of a state variable by simply writing the state variable $x_i = v_{ij}$, where x_i is a declared variable and v_{ij} is an element of x_i's domain.

Note that, as RMPL is a language for describing the evolution of state variables through time, every state variable assignment has a corresponding episode during which it persists.

3.3.3 Sequential Composition - A;A'

Programmers frequently want to constrain two activities such that one occurs immediately after another. In this situation, the sequential composition construct is used. For example, the code { cook( ); eat( ) } would instruct a system to perform the cook activity, and then immediately execute the eat activity.

3.3.4 Parallel Composition - A,A'

RMPL includes a parallel composition construct to allow the expression of concurrent activities. Parallel activities are constrained to begin and end at the same time. For example, the code { sneeze( ), close-eyes( ) } would instruct a system to simultaneously begin the sneeze and close-eyes activities, and then simultaneously end both activities.

3.3.5 Conditional Execution - if c then A [else A']

RMPL's conditional execution construct, if-then, allows sub-activities to be executed when a specified state assignment is true. This construct, along with the other control statements, is particularly important as it enables RMPL to react to environmental conditions. For example, a programmer might encode the program "if (environment = safe) then fly-mission( ) else abort( )."

Note that if-then only requires a state assignment to hold at the beginning of the embedded activity. That is, after the activity begins, the state assignment is free to change. The combinator that maintains a state assignment throughout the execution of an activity is do-maintaining.

Also, note that the if-then combinator is only supported within Kirk's strategy selection algorithm, and not within Spock.

3.3.6 Pre-emptive Execution - when c then A

Another type of control statement is when-then. When a programmer wants a particular sub-activity to be executed every time a particular state assignment holds, he can use a when-then. For example, suppose a programmer wants to implement a simple obstacle-avoidance routine that halts a robot's motors whenever its proximity sensors register an object within a certain threshold. This obstacle-avoidance routine might be coded as "when (distance = below-threshold) then all-stop( )".

Note that the when-then combinator is only supported within Kirk's strategy selection algorithm, and not within Spock.

3.3.7 Maintenance Conditions - do A maintaining c, do A watching c

One of the most important activity constraints for programming autonomous vehicles is that of maintenance conditions. Frequently, mission programmers want to encode execution sequences with maintenance (or guard) conditions that require a particular state

assignment for the duration of the activity. To express these guard conditions in RMPL, programmers use the do-maintaining construct. For example, to express the constraint that a thruster only be fired while its fuel is pressurized, an RMPL programmer might write “do fire-thruster( ) maintaining (fuel = pressurized)”.

3.3.8 Non-deterministic Choice - Choose { A, A’, … }

RMPL also includes support for non-deterministic contingency selection. This allows mission programmers to specify functionally-redundant procedures that improve robustness by encoding contingency sequences. To encode a non-deterministic choice, one uses the choose construct followed by a list of possible execution threads. For example, to encode the scenario where a UAV selects from a series of three surveillance targets, an RMPL programmer would encode the following, “{choose { fly-over (target1) }, { fly-over (target2) }, { fly-over (target3) } }”.

Note that the choose combinator is only supported within Kirk’s strategy selection algorithm, and not within Spock.

3.4 RMPL Subsumption of PDDL+ Operators

The planning community has established the Planning Domain Description Language (PDDL+) as a standardized format for encoding planning problems [12]. PDDL+ was developed to be a flexible format for encoding primitive operators. PDDL+ supports durative actions, start pre-conditions and effects, invariant conditions and effects, and end pre-conditions and effects.

An important claim of this thesis is that RMPL is an expressive language for describing operators. To argue this claim, we note that with the addition of a single combinator to RMPL, arbitrary PDDL+ operators can be encoded using RMPL.

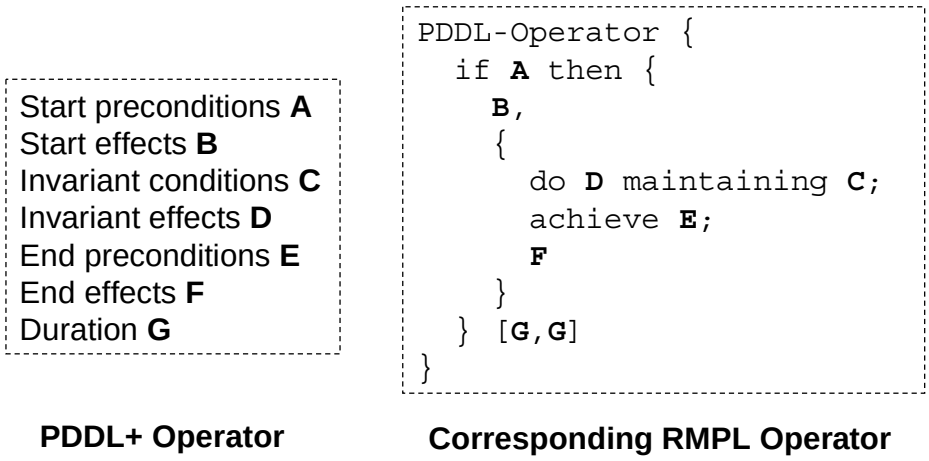


Figure 3-4 RMPL Subsumption of PDDL+ Operators

A standard PDDL+ operator takes the form shown on the left side of Figure 3-4. With the addition of an achieve combinator that denotes an activity operator sub-goal, we see that any PDDL+ operator can be represented in RMPL. PDDL+ operators are accepted as an expressive operator format. As RMPL can express arbitrary PDDL+ operators, we conclude that RMPL is also an expressive format for encoding plan operators.

3.5 RMPL for the Spock Generative Planner

Our research on Spock has focused on the generation of unconditional plans. To achieve this focus, we do not allow plan operators to include conditional expressions (if-then, do-watching, etc.) or non-deterministic choice. Instead, Spock's operators are coded using a subset of the RMPL language consisting of activity timing, state assertion, sequential composition, parallel composition, and the do-maintaining combinator.

Furthermore, the control program, activities, and environment model in a planning problem have different roles. The control program represents a mission designer's planning goals. The activities in the activity library correspond to plan operators that have effects which achieve a control program's open conditions. Finally, the environment model corresponds to a set of state assignments that cannot be changed. Because of this semantic difference, the interpretation of RMPL state assignments varies between these inputs.

In a control program, constraints are meant to indicate intended state assignments, or planning goals. Thus, state assertion in an RMPL control program corresponds to a request that a particular assignment be achieved for some period of time. For example, in a control program, the state assertion "light = on [10,+INF]" would signify a request that the model-based executive achieve the state where the light is on for at least 10 time units.

In activities and environment models, a state assertion corresponds to an effect. Thus in these inputs, an RMPL state assertion corresponds to an explicit assertion that a particular assignment hold for some period of time. For example, in an activity, the state assertion "light = on [10,+INF]" would signify an assertion (or operator effect) that the light is in fact in the "on" state for at least 10 time units.

3.6 Conclusion

The Reactive Model-based Programming Language is an effective tool for mission programmers that allows them to express constraints while efficiently managing complexity. Rooted in proven execution and modern object-oriented languages, RMPL is a process algebra that enables programmers to easily encode arbitrarily complex activity models and mission control programs.

4 Temporal Plan Networks

RMPL allows a programmer to specify complex processes in terms of the evolution of state variables. To enable fast planning, we convert RMPL programs into equivalent graph structures called Temporal Plan Networks [19].

TPNs are useful in that they compactly encode the space of possible state evolutions expressed by an RMPL program. Once a program has been converted to a TPN, it can be processed using efficient network algorithms to perform search, scheduling, and to check temporal consistency.

This chapter presents an overview of Temporal Plan Networks, gives an example TPN based on the RMPL example from Chapter 3, and describes the mapping from RMPL combinators to TPN constructs.

4.1 TPN Overview

Temporal Plan Networks are inspired by the history-based process representations used in qualitative physics [15] and concise histories [31], and by interval representations from constraint-based interval planning [30]. As such, the episodes (or arcs) in a TPN represent state variable assertions and requests that hold for a given interval of time. The end-points of these episodes are called events, which are represented in the TPN using graph vertices. To be temporally flexible, a TPN's episodes are bound with simple temporal constraints that include both a lower and upper-bound for the corresponding interval of time (or episode). To encode state queries and assertions, episodes are labeled with Ask and Tell constraints, respectively. Episodes can also be labeled with primitive activity operators. Finally, TPNs add decision nodes, which allow non-deterministic choice within the plan representation (but note that decision nodes are not allowed by Spock).

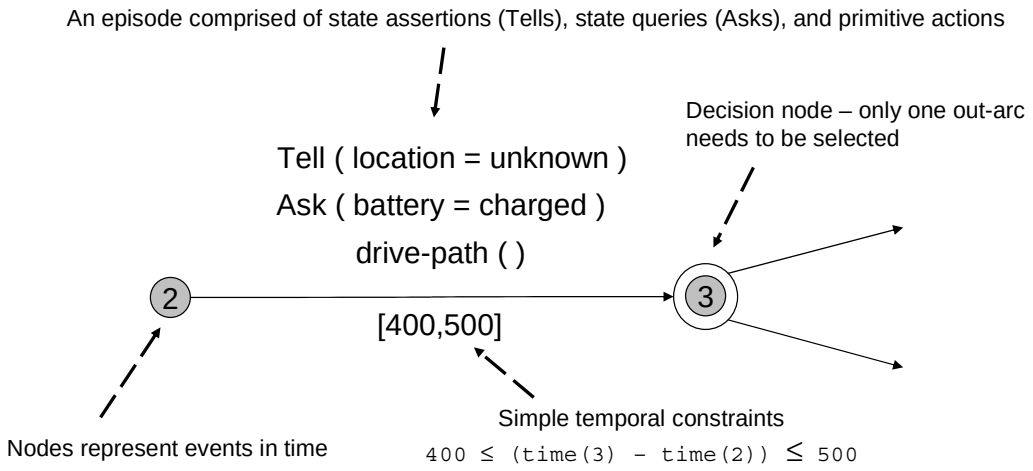


Figure 4-1 Temporal Plan Network Constructs

Figure 4-1 illustrates the constructs in a Temporal Plan Network. In this example, nodes 2 and 3 represent events in time, while the arc from Node 2 to Node 3 represents the episode during which the `drive-path` primitive action is being executed. The label `[400,500]` below the arc represents the time-bounds attached to the episode. These time-bounds constrain the episode between events 2 and 3 to take at least 400 and not more than 500 time units.

A state assertion and state request are also attached to the episode arc. `Tell (location = unknown)` asserts that the system's location variable is undefined for the duration of the `drive-path` episode, while `Ask (battery = charged)` requests that the system achieve the state where the battery is charged in order to ensure that the robot does not run out of power during the episode's execution.

Finally, Node 3 is a decision node. This means that the model-based executive must select only one of its out-arcs for execution. Note that the end event of an episode does not have to be a decision node, and that the start event of an episode is allowed to be a decision node. Lastly, we reiterate that TPNs within the Spock planner do not include decision nodes, as Spock does not perform conditional planning.

4.2 Example TPN

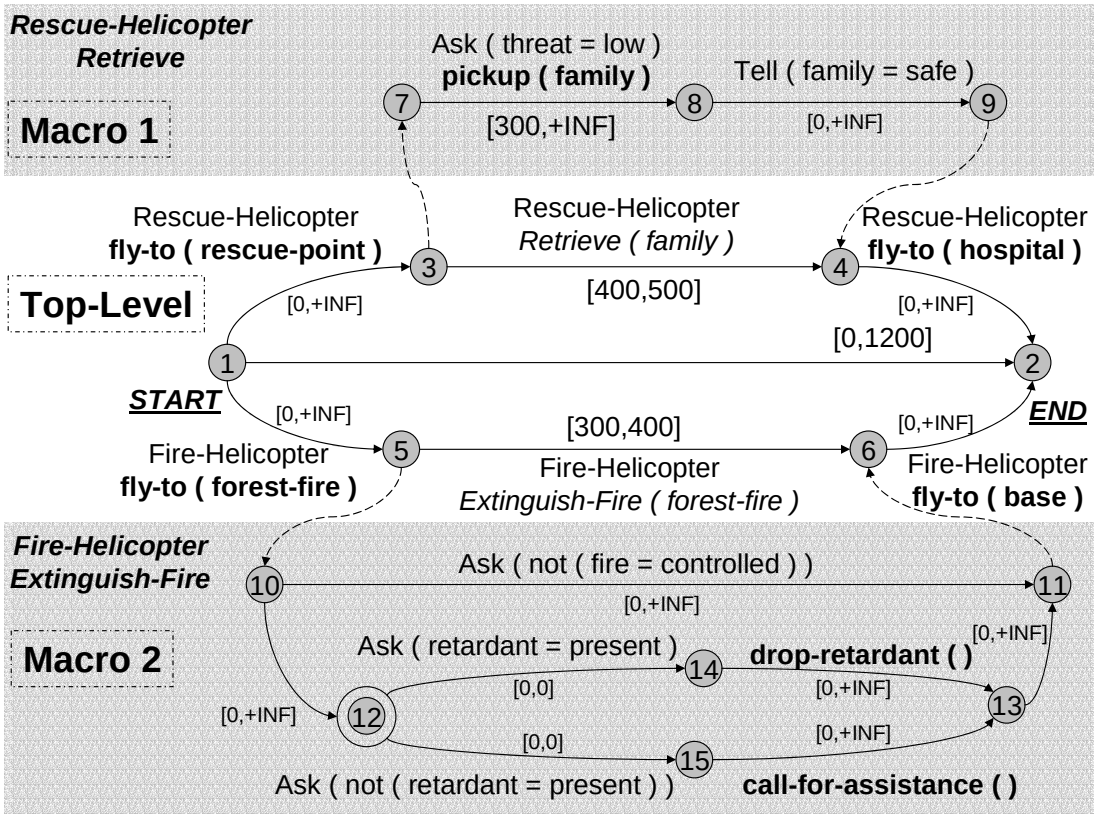


Figure 4-2 An Example Temporal Plan Network

An example TPN is shown in Figure 4-2 corresponding to the example RMPL code shown in Figure 3-1 of Chapter 3. Just like the original RMPL code, this graph has three distinct parts: the top-level program, and two macro activities that are expanded into the control program.

In this TPN, the top-level program sub-section contains two parallel threads of execution, (1-3-4-2 and 1-5-6-2). There is also a total mission time-bound of 1200 seconds.

The top-level program also demonstrates both primitive activities (the four `fly-to` activities) and macro activities (the `Rescue-Helicopter.Retrieve` and `Fire-Helicopter.Extinguish-Fire` activities). While primitive activities are simply included in the solution plan, macro activities need to be expanded into the TPN.

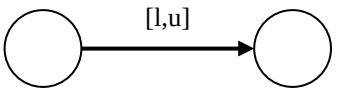
The TPN within sub-network Macro 1 corresponds to the expansion of the `Rescue-Helicopter.Retrieve` activity. In this sub-network, the episode between events 7 and 8 shows the expansion of the RMPL do-maintaining combinator. In this example, the command is `pickup`, while the state to maintain is `(threat = low)`. Thus the do-maintaining RMPL code is expanded into a TPN sub-network that asks that the mission threat remain low for the duration of the embedded rescue activity. Finally, when the pickup command (which is constrained to take at least 300 seconds) is finished, the state `family = safe` is asserted.

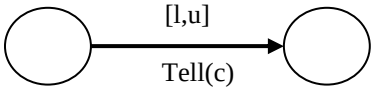
Macro 2 corresponds to the expansion of the `Fire-Helicopter.Extinguish-Fire` activity. The bulk of this activity is nested within a do-watching activity, which is similar to a do-maintaining. The difference between the two is that do-maintaining commands ask for a particular state to hold, while do-watching commands execute as long as a particular state does *not* hold. Moreover, a do-watching statement is specified to halt its execution when the embedded condition becomes true. Thus Macro 2 executes as long as `fire = controlled` remains false.

The code embedded in Macro 2's do-watching statement instructs the system with an if-then-else statement about how to fight the fire. As the if-then-else statement requires a decision to be made, the corresponding TPN sub-graph contains a decision node (denoted with a double-circle). The choice at the decision node is based on the state of the `retardant` variable due to the Ask constraints attached to both out-arcs. The (12-14-13) thread requires that `retardant = present` is true, in which case the drop-retardant primitive is executed, while the (12-15-13) thread requires that `retardant = present` is not true, in which case the call-for-assistance primitive is executed.

4.3 RMPL to TPN Mapping

This section summarizes the mapping from RMPL combinators to TPN constructs. By using the translations in this section, any RMPL program can be compiled in a TPN that is suitable for planning and execution tasks.

`Interval:`
 	
`[l,u]`
	

`Interval + Assertion:`
 	
`c[l,u]`
	

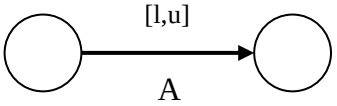
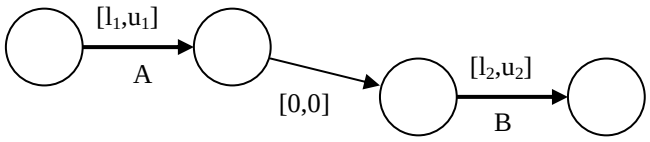
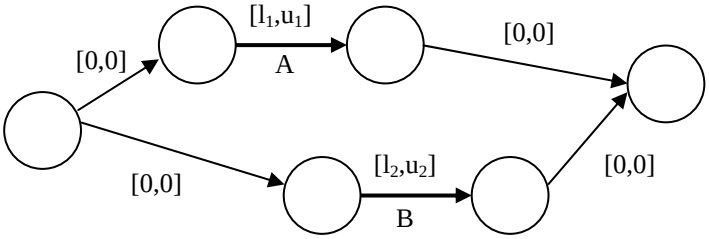
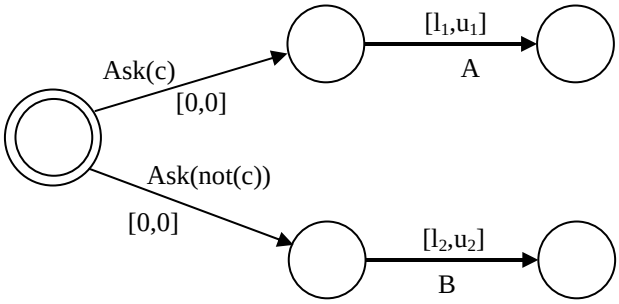
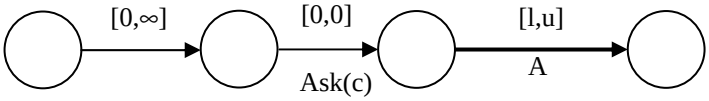
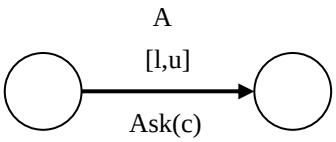
`Interval + Activity:`
 	
`A[l,u]`
	

Table 4-1 RMPL Primitives to TPN Sub-networks

Table 4-1 shows the mapping from RMPL to TPN primitives. Using the three shown primitive statements, mission programmers can express delays, timed assertions, and timed primitive actions in RMPL programs. Each of these primitive statements has a corresponding primitive TPN construction that represents the same information in graph form.

Sequential Composition: A[l_1,u_1]; B[l_2,u_2]	
Parallel Composition: A[l_1,u_1], B[l_2,u_2]	
Conditional Execution: if c then A[l_1,u_1] else B[l_1,u_1]	
Reactive Execution: when c then A[l,u]	
Condition Maintenance: do A[l,u] maintaining c	

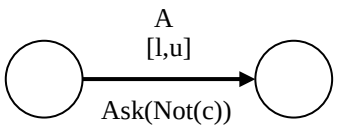
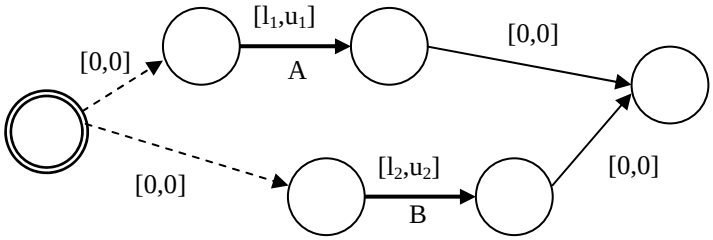
Preemption: do A[l,u] watching c	
Choice: choose{ A[l_1,u_1], B[l_2,u_2] }	

Table 4-2 RMPL Combinators to TPN Sub-networks

Table 4-2 shows the mapping from RMPL combinators to TPN sub-networks. Using the shown combinators, mission programmers can combine RMPL primitives to represent complex processes. As the graph-based equivalent of RMPL, TPNs can represent all of the process combinators using various graph constructions.

4.4 Executability of Temporal Plan Networks

Not all TPNs are executable on mission hardware. This is either because some open condition (Ask) within the TPN is not satisfied, or some combination of TPN constraints is conflicting. For example, a TPN control program, which encodes a mission designer's planning goals, is not executable, as it has open conditions that need to be satisfied.

A TPN planner takes a TPN control program and combines it with an environment model and activities from the activity library in order to satisfy the control program's open conditions. The resulting solution TPN is said to be executable if it is both complete and consistent. The following sections explain TPN consistency and completeness in detail.

4.5 TPN Consistency

Finding consistent plans is important, as only consistent plans can be executed on real-world systems. TPN consistency has two components: temporal consistency and Tell

consistency. Temporal consistency requires that a valid temporal assignment to each event exist such that no temporal constraints are violated, while Tell consistency requires that each state variable have at most a single assignment at any point in time.

4.5.1 TPN Temporal Consistency

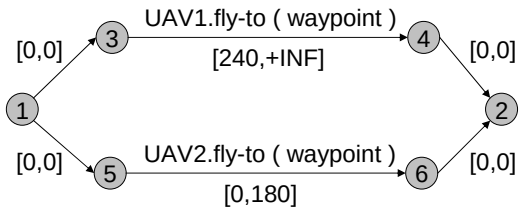


Figure 4-3 A Temporally Inconsistent TPN

It is possible for a Temporal Plan Network to represent a temporally infeasible mission plan that is therefore not executable. For example, in Figure 4-3, two aerial vehicles are commanded to rendezvous at a waypoint. One of the vehicles is far away and will take at least 4 hours to reach the waypoint, while the other vehicle is low on fuel and must complete the rendezvous in at most 3 hours. These two constraints conflict, meaning that there is no possible time for the rendezvous to occur without violating one of the vehicles' temporal requirements. Thus we say that the plan is temporally inconsistent.

Because TPNs have temporal constraints similar to Simple Temporal Networks [8], graph algorithms for determining STN consistency can also be applied in order to determine TPN temporal consistency. As shown by Dechter and Meiri, the temporal constraints from both STNs and TPNs can be reformulated into an equivalent graph, called a distance graph. A distance graph is a graphical encoding of each upper and lower bound in a graph with simple temporal constraints. Consistency checking for a graph with simple temporal constraints corresponds to negative cycle detection within the associated distance graph [2].

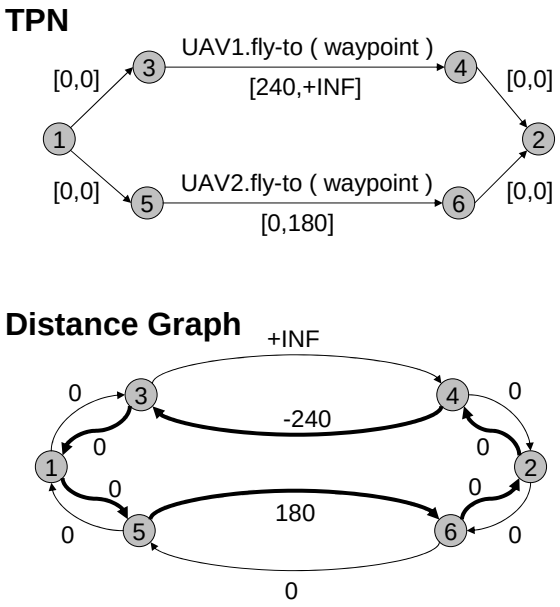


Figure 4-4 Inconsistent TPN with Corresponding Distance Graph

A graph with simple temporal constraints can easily be converted into a distance graph. First, all the nodes from the input graph are copied into the distance graph. Then, each upper bound in the input graph is converted into a directed arc with the same value and direction as the simple temporal constraint. Finally, each lower bound in the input graph is converted into a directed arc with the negative value and opposite direction as the simple temporal constraint.

```
make-distance-graph (TPN input)
   returns distance graph
1. Let d = distance graph
2. For each event, i, in input
3.     add node i to d
4. For each episode from i to j in input
5.     add arc (i,j) to d with episode upperbound as weight
6.     add arc (j,i) to d with negative episode lowerbound as weight
7. return d
```

Figure 4-5 TPN to Distance Graph Pseudo Code

As mentioned above, temporal consistency in a TPN or STN corresponds to negative cycle detection in the associated distance graph [8][2]. Once the distance graph for a

given TPN has been constructed, one can easily determine temporal consistency by using a negative cycle detection algorithm, such as the Floyd-Warshall all-pairs shortest path algorithm [7] or the FIFO label-correcting algorithm [1].

4.5.2 TPN Tell Consistency

To be consistent, a TPN must not only be temporally consistent, but also ensure that its assignments to state variables do not conflict. This is to ensure that each state variable is assigned a unique value at each point in time. When two Tells contain inconsistent state assignments, we say that they threaten each other. Thus, the process of ensuring that no inconsistent Tells co-occur in time is referred to as threat resolution.

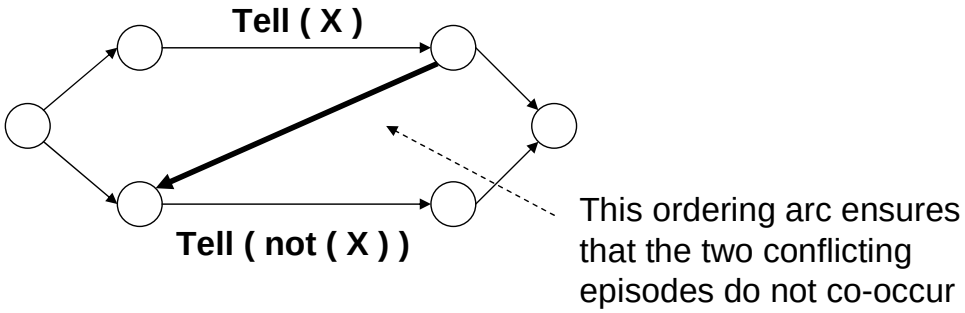


Figure 4-6 TPN Threat Resolution

An example of a TPN with threatening Tells is shown in Figure 4-6. In this TPN, the threat is resolved by introducing an ordering arc that prohibits the two Tells from co-occurring. Finally, we say that a TPN is consistent when its temporal constraints are consistent and its Tell constraints are consistent (or not threatening).

4.6 TPN Completeness

For a Temporal Plan Network to be executable, it must be complete. A TPN is complete when all of its embedded open conditions (Asks) are satisfied. Specifically, TPN completeness corresponds to a control program TPN being successfully combined with a

TPN environment model and a set of activity TPNs from the activity library in order to achieve the mission designer's planning goals.

In a Temporal Plan Network, Ask constraints represent open conditions that the system must satisfy. Therefore, the planning goals within a scenario's control program and activities always take the form of Ask constraints.

Recall that whereas Ask constraints request state assignments, Tell constraints assert state assignments. Thus for the open condition in an Ask constraint to be closed, a TPN must guarantee that the Ask's state assignment is entailed by some Tell constraint in the network. Also, as Ask and Tell constraints are assigned temporal episodes, a Tell can only close an Ask if its time-bounds subsume (or *contain*) the time-bounds of the Ask constraint.

When all of the Ask constraints in a TPN are closed by Tell constraints and any conflicting Tell constraints are ordered so as to not co-occur, we say that the TPN is complete.

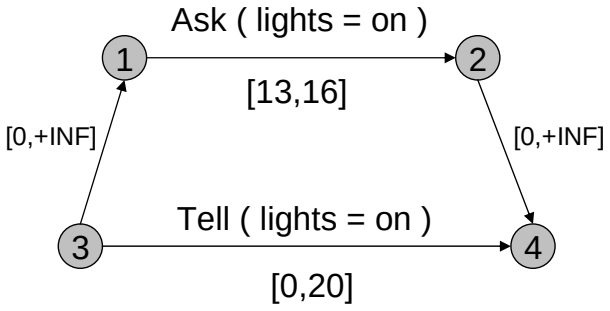


Figure 4-7 Example of Complete TPN

Consider the example in Figure 4-7. In this example, the TPN has an Ask constraint that requires the lights to be on for 13 to 16 seconds. The Tell constraint asserts that the lights are on for up to 20 seconds. By connecting the Ask and Tell constraints with causal links (episodes with [0,+INF] time-bounds), the TPN ensures that the Tell constraint *contains* the Ask constraint, thus closing the open condition.

4.7 TPN Subsumption of PDDL+ Operators

As described in Chapter 3, the planning community has established the Planning Domain Description Language (PDDL+) as a standardized format for encoding planning problems [12]. PDDL+ supports durative actions, start pre-conditions and effects, invariant conditions and effects, and end pre-conditions and effects.

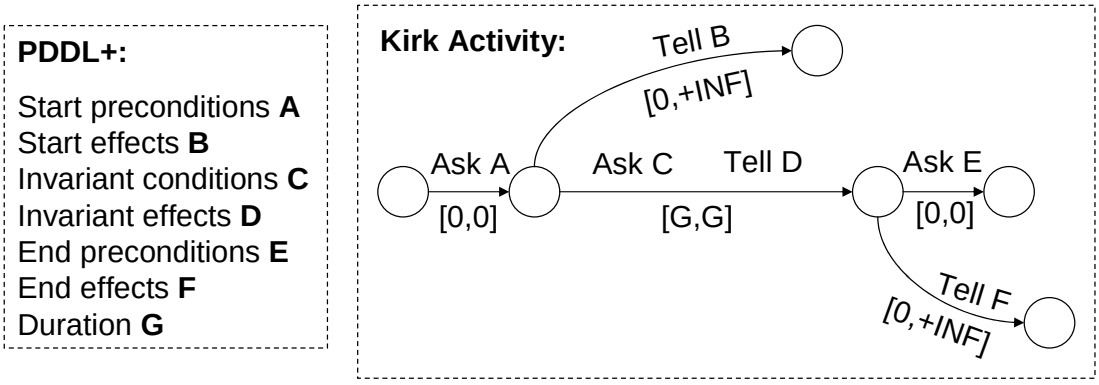


Figure 4-8 Mapping from PDDL+ Operators to TPN Activities

An important claim for this thesis is that Spock's planning operators, Temporal Plan Networks, allow the specification of rich activity models. This claim is supported by the fact that TPN activities subsume PDDL+ operators. Figure 4-8 demonstrates how any arbitrary PDDL+ operator can be expressed as a TPN activity.

4.8 Summary

Temporal Plan Networks are a compact graph encoding of the constraints expressed in an RMPL program. Representing complex processes in network form, TPNs can be quickly processed via graph search algorithms to determine temporal consistency and perform scheduling. Finally, there is a direct mapping between the combinators in RMPL and the constructs in a TPN, allowing the easy translation from human-generated code to a machine-understandable graph format.

5 The Spock Generative TPN Planning Algorithm

Spock is a generative optimal forward progression TPN planner, designed to support strategic-level control of autonomous mobile systems as part of the Kirk model-based executive. This chapter describes the Spock planning algorithm in detail. An overview is first presented, followed by a discussion of Spock's internal plan representation. Next, Spock's child expansion function is given, followed by Spock's consistency checking algorithm. Finally, the chapter concludes with a description of Spock's candidate cost update function. Throughout the chapter, illustrative examples are used to help convey the relevant concepts.

5.1 Overview

Spock is designed to integrate forward progression heuristic search, temporal flexibility, and the composition of complex processes. While HTN planners such as HSTS [17] have been developed for real-world systems in the past, forward progression heuristic search has recently been demonstrated by such planners as FF [16], HSP [6], and LPG [14] to be a novel way to achieve even faster planning speeds. Spock applies this fast search algorithm to the Temporal Plan Network structure, which provides temporal flexibility like the constraint-based interval planners HSTS and Europa [17]. Finally, Spock's inputs are expressed in the Reactive Model-based Programming Language, which allows mission designers to specify the evolution of state variables within complex processes by using a process algebra with a rich set of activity combinators. While each of these components has been demonstrated individually, Spock is novel in that it combines these three capabilities in one framework to support model-based programming for mobile autonomous systems.

5.1.1 Spock Algorithm

Spock requires three inputs: a control program that describes a system's intended state evolutions, an environment model, and an activity library that Spock uses to assemble a solution plan. The solution plan output by Spock is a complete and consistent Temporal

Plan Network that achieves the behavior specified in the control program by piecing together activities from the activity library, while maintaining consistency.

Spock uses Temporal Plan Networks as a uniform representation for representing control programs, activities, and plans. As described previously, TPNs are collections of events and episodes between those events, representing processes that may have their own sub-goals in the form of open conditions represented by Ask constraints. Spock generates a complete plan by walking over a control program from its start to its end, along the way satisfying any open conditions using activities from the activity library. When Spock has a choice as to how to proceed, it branches, adding each possible expansion to its queue of plan candidates.

When Spock inserts an activity from the activity library, it is committed to inserting the entire activity TPN. Because Spock inserts events and episodes of an activity into a plan candidate TPN one at a time, each plan candidate needs to keep track of the events and episodes that it must insert in the future. These events and episodes are called *pending* events and episodes. Thus Spock's internal plan candidate representation contains both a candidate TPN, and a set of pending events and episodes (see Figure 5-7). When a consistent candidate is found with no remaining pending events or episodes, the plan candidate is complete and is returned as a solution plan.

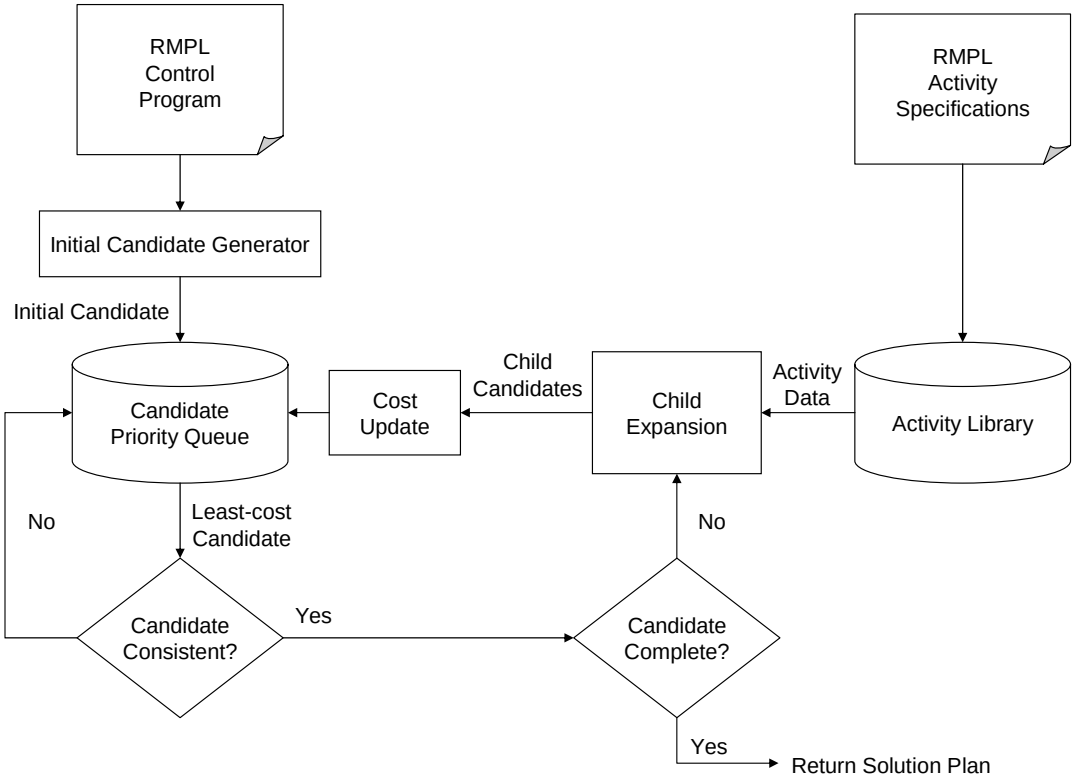


Figure 5-1 Spock Block Diagram

Spock's planning loop is shown pictorially in Figure 5-1 and in Figure 5-2 as pseudo code. When Spock solves a planning problem, it begins by removing a least-cost plan candidate from the queue (pseudo code line 4). This candidate is tested for consistency, and if it fails, the candidate is discarded (pseudo code line 5).

Next, Spock checks to see if the candidate is complete (pseudo code line 6). If a candidate has no remaining pending events or episodes, it is complete and is returned as a solution plan. If the candidate is not complete, planning continues with the child expansion function.

Spock's child expansion function generates child candidates based on the parent candidate (pseudo code line 7). As it expands a candidate, Spock's child expansion function can either insert a pending event or episode, or instantiate an additional activity from the activity library.

Finally, after each child candidate is constructed by the child expansion function, its cost is updated (pseudo code line 9) and it is reinserted into the candidate queue in the manner of uniform cost search and A* search (pseudo code line 10) [26].

```
Spock ( control program, activity library )
returns complete, consistent plan that achieves the control program

1.     let C = initial candidate based on control program
2.     let priority queue = {C}
3.     while priority queue is non-empty
4.        let C = least cost candidate from priority queue
5.        if C is consistent,
6.           if C is complete, return C
7.           let children = child-expansion ( C, activity library )
8.           for each D in children
9.              update-cost(D)
10.             insert D into priority queue
11.          endfor
12.       endif
13.    endwhile
```

Figure 5-2 Spock Top-level Pseudo-code

5.1.2 Example Generative TPN Planning Problem

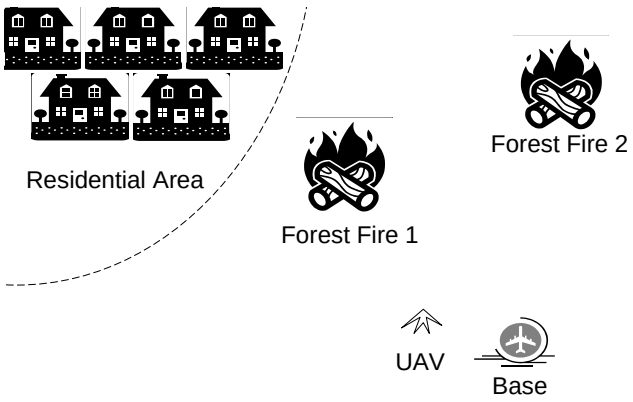


Figure 5-3 Forest Fire Suppression Scenario

Consider the following example scenario. Two forest fires are burning, and a fire marshal wants to send an unmanned aerial vehicle to suppress the flames (see Figure 5-3). One of the fires is threatening a residential area, so the fire marshal writes a control

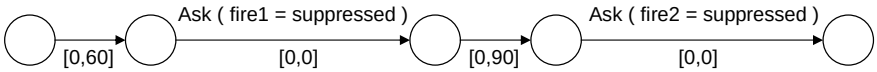
program requiring that the fire near the residential area be suppressed first (within 60 time units), followed by the second forest fire (within 90 subsequent time units) (see top left of Figure 5-4).

```
Fire-Suppression ( )
{
   [0,60];
   fire1 = suppressed [0,0];
   [0,90];
   fire2 = suppressed [0,0]
}
```

Control Program RMPL Code

Note:

- Deadline for fire1 suppression is 60 time units after mission start
- Deadline for fire2 suppression is 90 time units after fire1 is suppressed



Control Program TPN

Figure 5-4 Control Program RMPL Code and TPN for Fire Suppression Mission

Along with the control program, the fire marshal gives Spock an activity library with the activity models for the fire-fighting UAV.

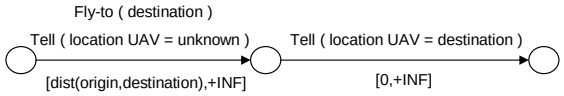
In this scenario, the activity library includes two activities: move and drop-water (see Figure 5-5). The move activity simply moves the UAV from one location to another. Note that one of the time-bounds in the Move activity (`dist(origin,destination)`) is parameterized based on the location of the origin and destination waypoints. This allows the time-bound for the Move activity to vary depending on the distance that the vehicle must travel. The drop-water activity is more straightforward, as it simply drops water on a location, requiring that the UAV remain at the location throughout the duration of the drop.

Finally, note that, as discussed in Chapter 3, the interpretation of RMPL state assertion is different for a control program and an activity model. In the control program, state

assertions become Ask constraints representing planning goals, while in an activity, state assertions become Tell constraints representing operator effects.

```
Move (origin, destination)
{
  fly-to(destination),
  location UAV = unknown,
  [dist(origin,destination),+INF];
  location UAV = destination [0,+INF]
}
```

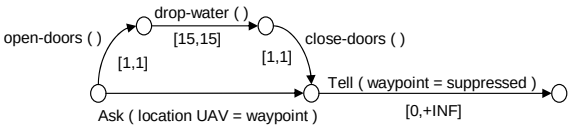
Move Activity RMPL Code



Move Activity TPN

```
Drop-water (waypoint)
{
  do {
    open-doors() [1,1];
    drop-water() [15,15];
    close-doors() [1,1]
  } maintaining location UAV = waypoint;
  waypoint = suppressed [0,+INF]
}
```

Drop Water Activity RMPL Code



Drop Water Activity TPN

Figure 5-5 Activity Library RMPL Code and TPN for Fire Suppression Mission

Given the scenario control program and activity library as inputs, Spock generates and returns a complete and consistent solution plan that achieves the control program with minimal cost using activities from the activity library, if it exists (see Figure 5-6).

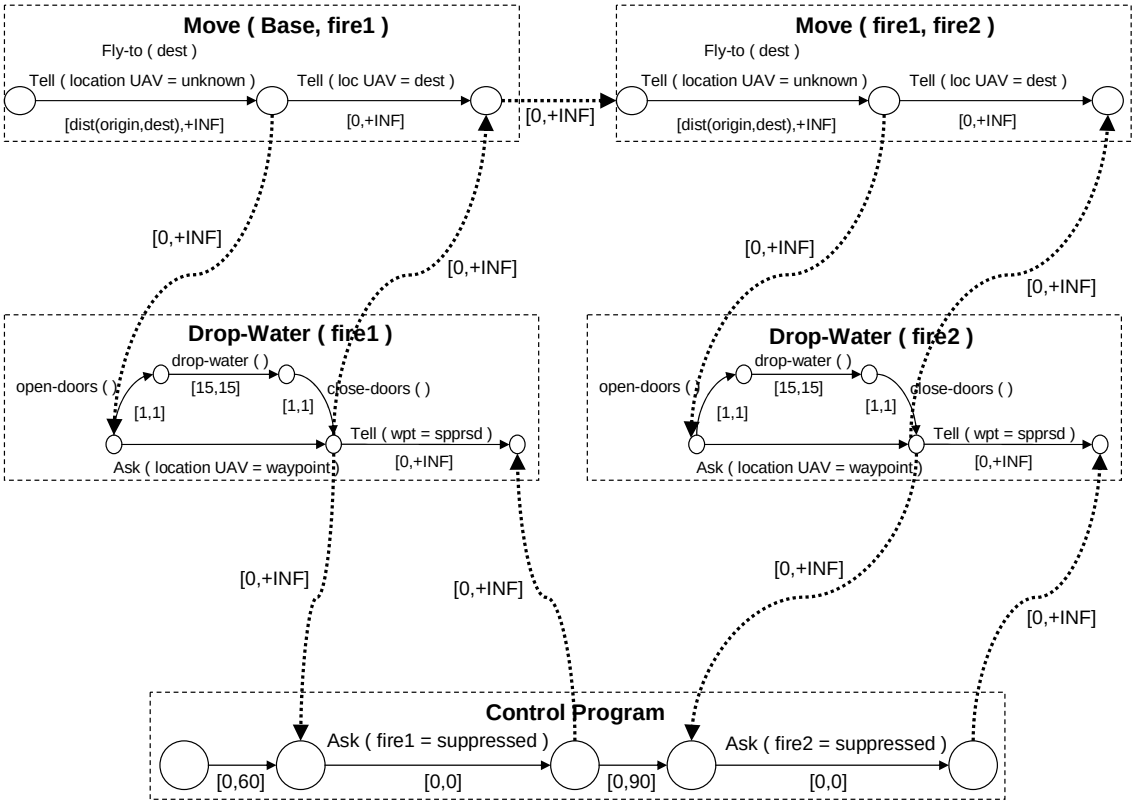


Figure 5-6 Solution TPN for Fire Suppression Mission

For this scenario, the solution plan achieves the control program by commanding the UAV to fly to forest-fire 1, drop water on the fire, fly to forest-fire 2, and finally drop water on the second fire (see Figure 5-6).

5.2 Internal Plan Candidate Representation

As Spock plans, it builds a solution plan by inserting the control program's events and episodes into a plan candidate one by one. When it encounters an Ask constraint in the control program, it inserts an activity from the activity library in order to close the open condition. Because Spock's walk over the control program is monotonic and temporally guided, there is an intuition that the part of the control program that has already been

considered represents the past, while the part of the control program that has not been considered represents the future. Spock's child expansion function makes frequent use of this distinction. Therefore, it makes sense to develop a relevant vocabulary to aide in describing the Spock planner.

There are four types of events and episodes in a plan candidate (see Figure 5-7). The first distinction within a plan candidate is between inserted and pending events and episodes. Inserted events and episodes correspond to the events and episodes that Spock has already considered (the past), while pending events and episodes correspond to the events and episodes that Spock will consider in the future. When an episode is pending, its associated Ask constraints are open and its associated Tells may be threatened, while an episode that is inserted contains closed Ask constraints and its Tells are all consistent.

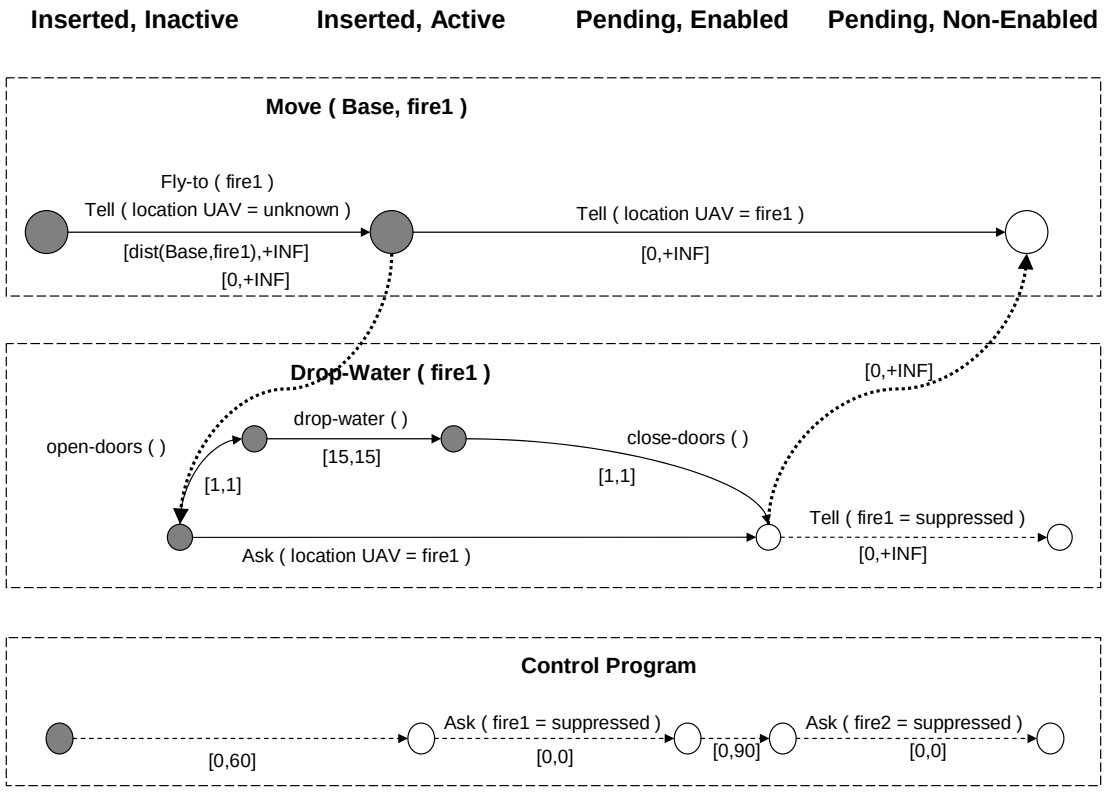


Figure 5-7 Plan Candidate Structure

Within the set of inserted events and episodes, Spock differentiates Tell constraints into active and inactive Tells. Active Tells represent the present state within the solution plan, and thus correspond to the part of the solution graph that affects the insertion of new events and episodes. Inactive Tells represent the solution plan's past, and have no effect on the insertion of new events and episodes. Specifically, active Tells are defined as inserted Tells with pending end events. Thus an active Tell is deactivated when the event at which it ends is inserted.

Spock differentiates the set of pending events and episodes into enabled and un-enabled events and episodes. An enabled event or episode is an event or episode that may be inserted into the solution plan while maintaining Spock's monotonic flow of time, as well as TPN consistency and completeness. Therefore, child expansion only inserts enabled events and episodes into a child candidate. It follows that un-enabled events and episodes correspond to the pending events and episodes for which insertion would yield an incomplete or inconsistent child candidate.

5.2.1 Enablement of Activities and Pending Events and Episodes

During child expansion, Spock must only insert activities, events, and episodes into a candidate TPN if the insertion results in a complete and consistent TPN. As described above, we refer to the activities, events, and episodes for which insertion is valid as enabled activities, events, and episodes. The conditions for enablement are as follows.

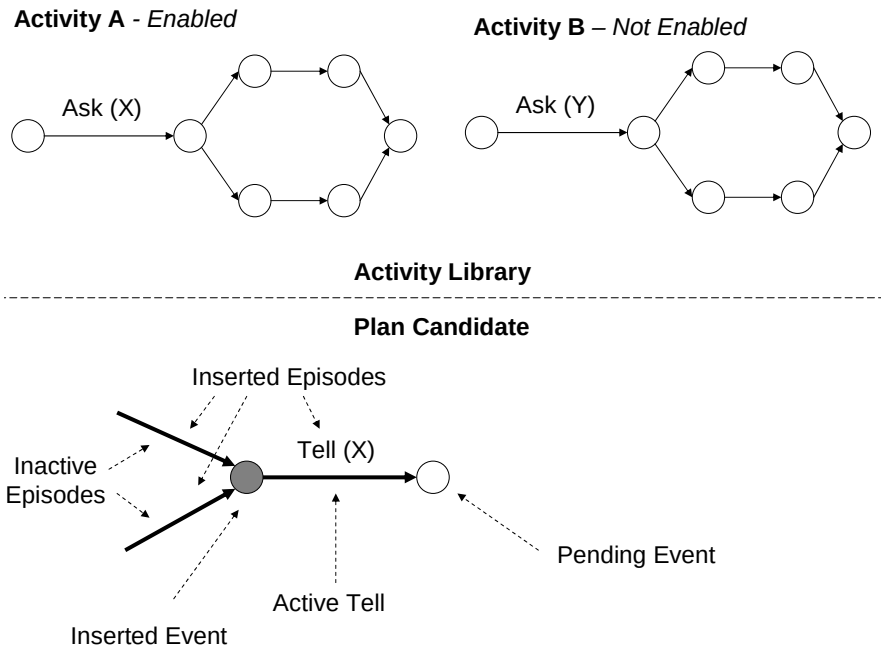


Figure 5-8 Activity Enablement

An activity in the activity library is enabled if the Ask constraints following its start event are closed by the candidate TPN's active Tells (see Figure 5-8). The intuition behind this is that, for an activity to be inserted, the candidate TPN should satisfy the activity's preconditions, which are represented by its initial Ask constraints.

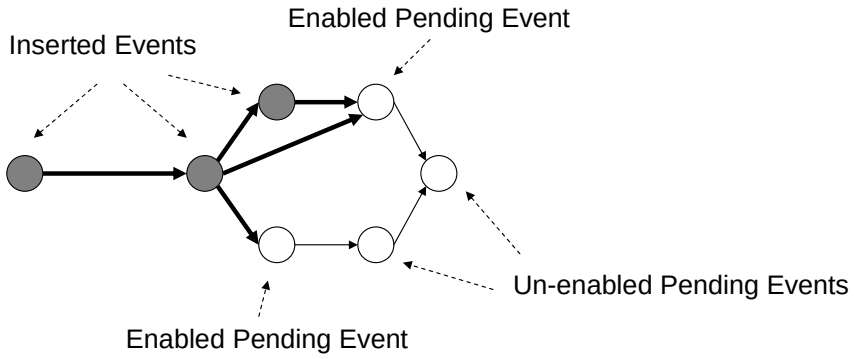


Figure 5-9 Event Enablement

An event is enabled if its preceding episodes are inserted (see Figure 5-9).

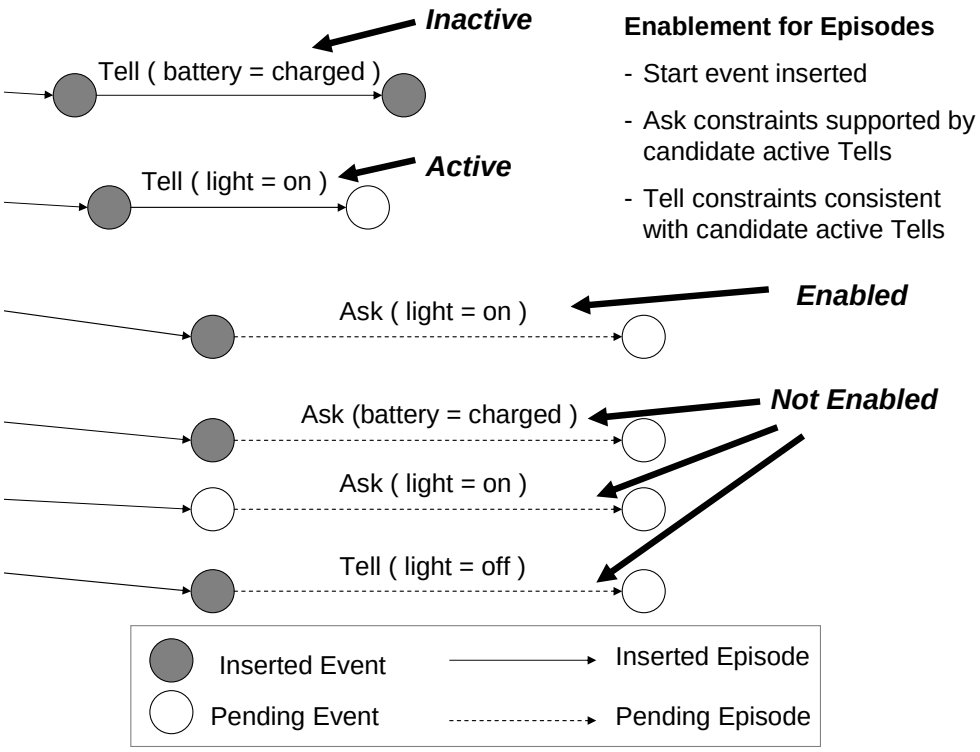


Figure 5-10 Enablement for Episodes

An episode is enabled if (1) its start event is inserted, (2) any Asks it contains are closed by the candidate TPN's active Tells, and (3) any Tells it contains are consistent with the candidate TPN's active Tells (see Figure 5-10).

Now that we have a vocabulary to describe the various events and episodes within a Spock plan candidate and we understand the conditions that give rise to enablement, we can discuss the algorithms that Spock uses to actually generate a solution plan.

5.3 Child Expansion

When a candidate is removed from the queue, it first is checked for consistency (see Section 5.4). When Spock determines that the candidate is consistent, it proceeds to check if it is complete. A candidate is complete when it has no pending episodes or events. If the candidate is complete, it is returned as a solution plan. Otherwise, child expansion is called.

Child expansion grows the plan candidate by inserting an enabled episode or event, or by instantiating a new activity from the activity library as shown in Figure 5-11. The expansion that is applied is selected arbitrarily. However, all possible expansions are considered and applied in order to create distinct candidates that ensure search completeness. This section describes the child expansion process for each of these cases.

```
Child-Expansion (candidate, activity-library)
returns child candidate
1.   let c = copy of candidate
2.   choose between lines 3, 7, 11:

3.     find enabled activities
4.     let a = choose an enabled activity
5.     instantiate(c,a)
6.     return c

7.     find enabled episodes
8.     let e = choose an enabled episode
9.     insert(c,e)
10.    return c

11.    find enabled events
12.    let e = choose an enabled event
13.    insert(c,e)
14.    return c
```

Figure 5-11 Child Expansion Pseudo-Code

5.3.1 Instantiating Activities

To expand a plan candidate, child expansion may add an activity from the activity library to the candidate's set of pending events and episodes. This is called activity instantiation, because an activity from the activity library is instantiated, its parameters are bound, and its components are added to the plan candidate's set of pending events and episodes.

Activity instantiation is a key planning component, as it represents the part of Spock that makes it a true generative planner. Without activity instantiation, Spock's child

expansion procedure would behave like Kirk's strategy selection algorithm, ensuring consistency and completeness of a pre-defined control program.

For an activity to be instantiated, it must be enabled, meaning that the Asks following its start event must be closed by the candidate's active Tells. This is because the Asks following an activity's start event represent the activity's preconditions, and it is wasteful to instantiate an activity whose preconditions are not satisfied.

Spock determines which activities are enabled by evaluating each activity in the activity library and checking to see if the Asks following its start event are closed by the candidate's active Tells (see Figure 5-12). An alternative method for performing this task is described in future work Section 6.5.

```
Find-Enabled-Activities (candidate C, activity library L)
1.   let S = empty set of activities
2.   for each activity, A, in L
3.     let OK = true
4.     for each Ask, K,  following the start event of A
5.       if K is not closed by the active Tells in C
6.         OK = false
7.     end-for
8.     if OK = true
9.       add A to S
10.    end-if
11.  end-for
12.  return S
```

Figure 5-12 Find Enabled Activities Pseudo Code

When an enabled activity is selected for instantiation, the newly instantiated activity is copied into the candidate, with its events becoming pending events and its episodes become pending episodes (see Figure 5-13). In effect, the instantiated activity becomes part of the control program, in that all of its constructs must be integrated into the candidate TPN in order for the candidate to be a solution plan.

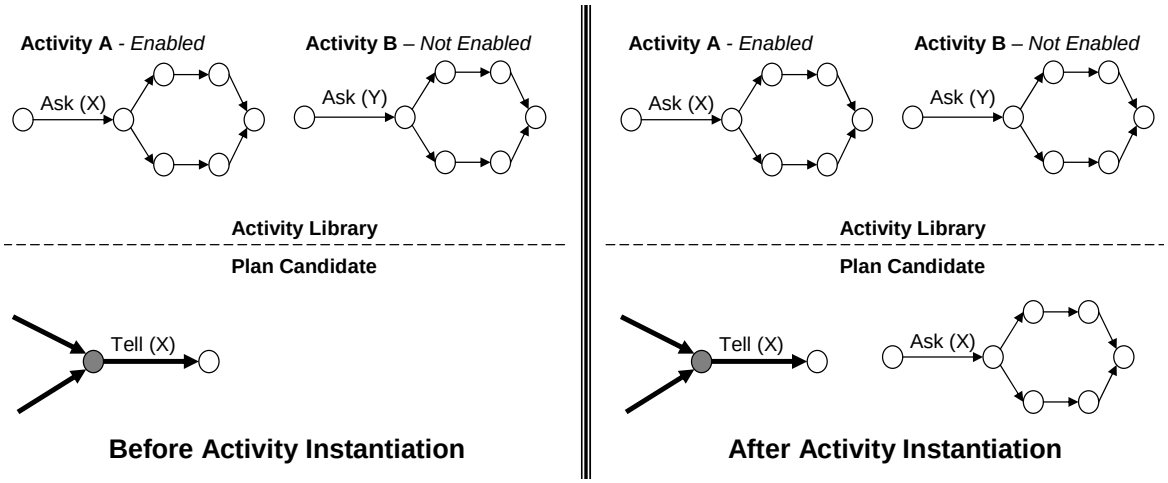


Figure 5-13 Example of Activity Instantiation

After an activity is instantiated, the candidate is returned to the queue. As the activity's events and episodes have been added to the candidate's set of pending constructs, the actual insertion of the activity's constructs is handled in the same way as the rest of the candidate's pending constructs (see the following two sections).

Note that Spock maintains search completeness by branching at each iteration and creating distinct candidates that instantiate each enabled activity. We acknowledge that a better approach would use a goal-directed search that only instantiates activities that close an existing open condition.

5.3.2 Inserting Enabled Episodes

Recall from Section 5.2.1 that an episode is enabled when its start event is inserted, any attached Ask constraints are closed by the candidate's active Tells, and any attached Tell constraints are consistent with the candidate's active Tells. Start events must be inserted in order to maintain a contiguous TPN graph, while Ask constraints must be closed in order to maintain TPN completeness. Finally, Tell constraints must be consistent to ensure that the candidate TPN maintains Tell consistency.

When Spock's child expansion function elects to insert enabled episodes, it starts by determining which episodes within the candidate are enabled for insertion. To determine which episodes are enabled, Spock searches the set of active Tells to see if an episode's

Asks are closed and if its Tells are consistent (see Figure 5-14). While this search process is not the most efficient way to detect closing Tell constraints, the number of Tells that need to be examined is small because the active Tells are only a small subset of the candidate TPN. Moreover, an improved algorithm is discussed in future work Section 6.5.

```
Find-Enabled-Episodes (candidate C)
1.   let S = empty set of Episodes
2.   for each pending episode, T, in C
3.     let OK = true
4.     for each Ask, A, in T
5.       if A is not closed by the active Tells in C
6.         OK = false
7.       end if
8.     end-for
9.     for each Tell, L, in T
10.      if L is inconsistent with the active Tells in C
11.        OK = false
12.      end-if
13.    end-for
14.    if OK = true
15.      add T to S
16.    end-if
17.  end-for
18.  return S
```

Figure 5-14 Find Enabled Episodes Pseudo Code

When an enabled episode is inserted, its Ask and Tell constraints are processed to ensure TPN completeness and consistency, respectively. Specifically, each Ask constraint is bound to its closing active Tell within the candidate TPN. In addition, Tell constraints are ordered so as to avoid conflicts with mutually exclusive Tells elsewhere in the candidate TPN.

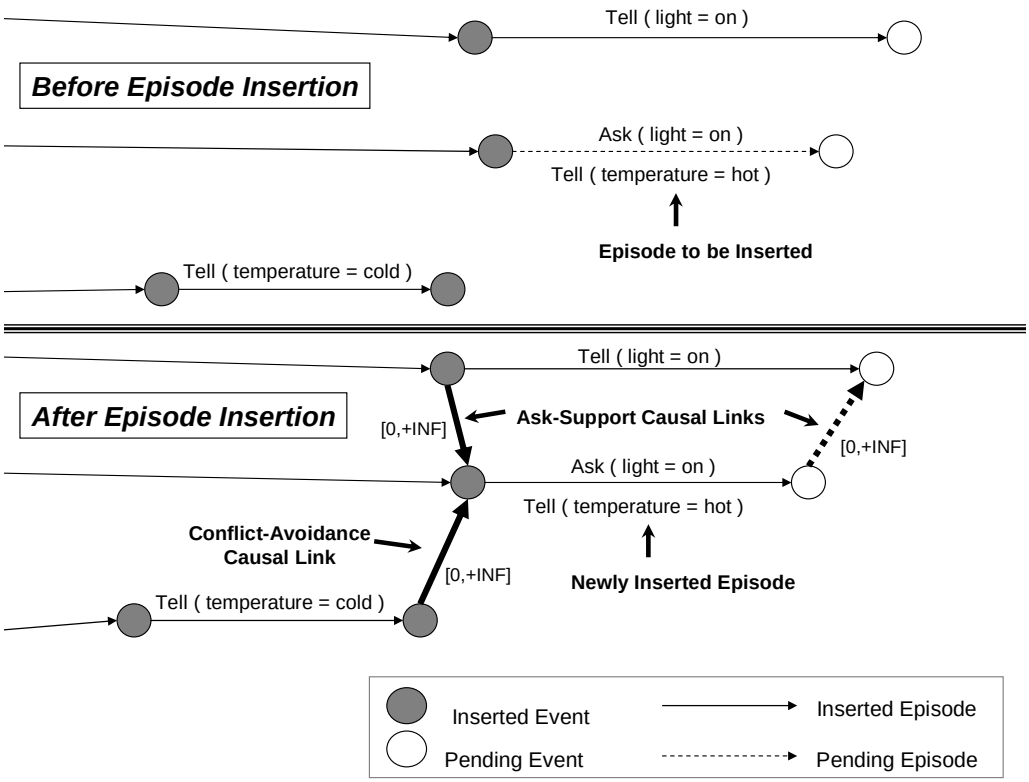


Figure 5-15 Episode Insertion

When Spock's episode insertion algorithm processes an episode Ask constraint, it binds each Ask to its closing Tell in order to ensure plan completeness. Note that Spock does not have to do any work to determine which Tell should close an Ask, as the binding is determined during the enablement checking procedure.

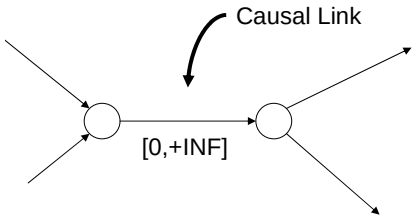


Figure 5-16 A Causal Link

At this point it is necessary to introduce the term *causal link*. A causal link is an episode with [0,+INF] time-bounds, and no attached Asks, Tells, or primitive activities (see Figure 5-16). While causal links are a type of episode, we distinguish them as they are

used merely to order plan activities, and never contain state assignments or constrained time-bounds of their own.

In the Ask / Tell binding process, Spock adds two causal links to ensure that the closing Tell constraint contains the Ask constraint (see Figure 5-18). One of the inserted causal links orders the Tell's start event to occur before the Ask's start event, and the other causal link orders the Ask's end event to occur before the Tell's end event. These causal links ensure that the Ask will be closed, as they require the Tell to be in effect for the entire duration of the Ask constraint.

```
Find-Conflicting-Tells (candidate C, Tell A)
1.   let S = empty set of Tells
2.   for each inactive Tell, T, in C
3.     if T conflicts with A, add T to S
4.   end-for
5.   return S
```

Figure 5-17 Find Conflicting Tells Pseudo Code

When Spock's episode insertion algorithm processes an episode Tell constraint, it adds causal links to ensure that the Tell will not co-occur with any pre-existing conflicting Tells. As Spock is designed to insert events and episodes in chronological order, Spock always orders new Tell constraints to occur *after* any conflicting pre-existing Tell constraints. Note that this does not violate search completeness, as all possible expansions are considered. That is, for each candidate where activity A is ordered to succeed activity B, an alternate candidate will have been generated where activity A is considered first, and activity B will be ordered to succeed activity A. This claim is argued further in Section 5.6 and Section 5.7.

Before inserting an episode Tell, Spock must find all conflicting inactive Tells. Spock finds the set of conflicting inactive Tell constraints by simply searching the set of inactive Tells (see Figure 5-17). This is admittedly not the most efficient solution, however an improvement is described in future work Section 6.5.

Once Spock identifies all of the inactive Tells that conflict with the new Tell, it de-conflicts the Tells by inserting a causal link from the end event of each conflicting inactive Tell constraint to the start event of the new Tell constraint (see Figure 5-18).

```
Insert (candidate, episode)
1.   for each Ask, A, in episode
2.     let S = choose supporting active Tell in candidate
3.     add causal link to candidate from S.start to A.start
4.     add causal link to candidate from A.end to S.end
5.   end-for
6.   for each Tell, T, in episode
7.     for each Tell, C, in candidate that conflicts with T
8.       add causal link to candidate from C.end to T.start
9.     end-for
10.  end-for
```

Figure 5-18 Insert Episode Pseudo-Code

5.3.3 Inserting Enabled Events

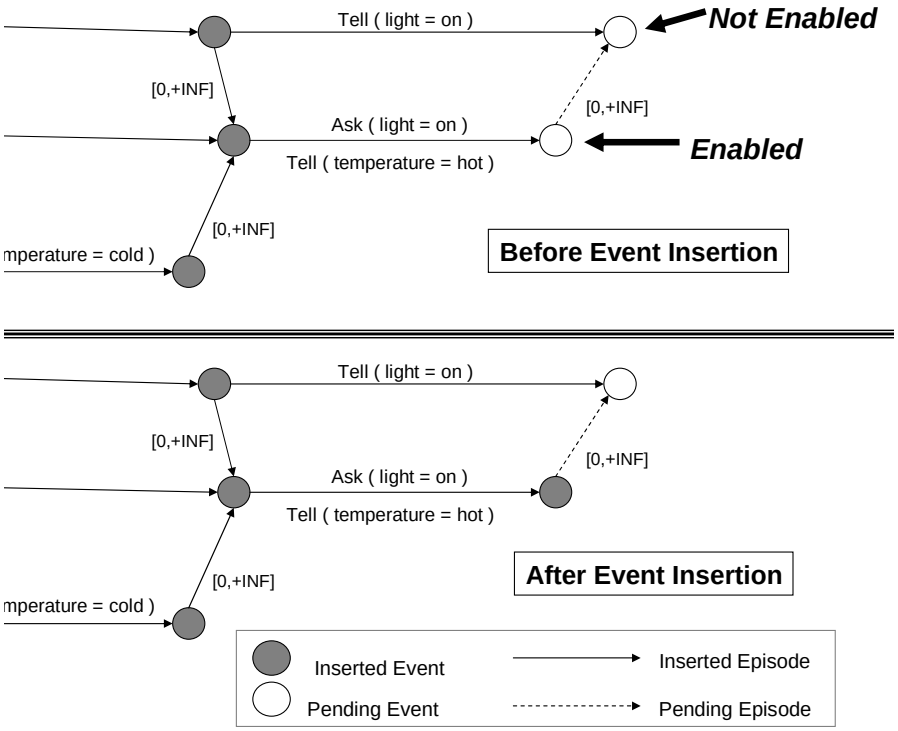


Figure 5-19 Event Insertion

Recall that an event is enabled if its preceding episodes are inserted. When child expansion elects to insert enabled events, it begins by determining which events are enabled (see Figure 5-20). It then branches by creating a distinct candidate that inserts each enabled event. When an enabled event is selected for insertion, it is inserted simply by moving the event from the candidate's set of pending nodes into the set of inserted nodes (see Figure 5-19 and Figure 5-21).

```
Find-Enabled-Events (candidate C)
1.   let S = empty set of Events
2.   for each pending event, E, in C
3.     let OK = true
4.     for each preceding episode, P, of E
5.       if P is not inserted
6.         OK = false
7.       end-if
8.     end-for
9.     if OK = true
10.      add E to S
11.    end-if
12.  end-for
13.  return S
```

Figure 5-20 Find Enabled Events Pseudo Code

```
Insert (candidate, event)
1.   mark event inserted (not pending)
```

Figure 5-21 Insert Event Pseudo-Code

5.4 Checking Candidate Consistency

Finding consistent solution plans is important, as only consistent plans can be executed on real-world systems. Spock ensures consistency by detecting and pruning inconsistent candidates. A plan candidate becomes inconsistent when a combination of the time-bounds on the episodes of the TPN conflict. Since episode time-bounds constrain the

time at which a TPN's events can occur, conflicting episode time-bounds mean that some event in the TPN cannot occur without violating at least one of the temporal constraints.

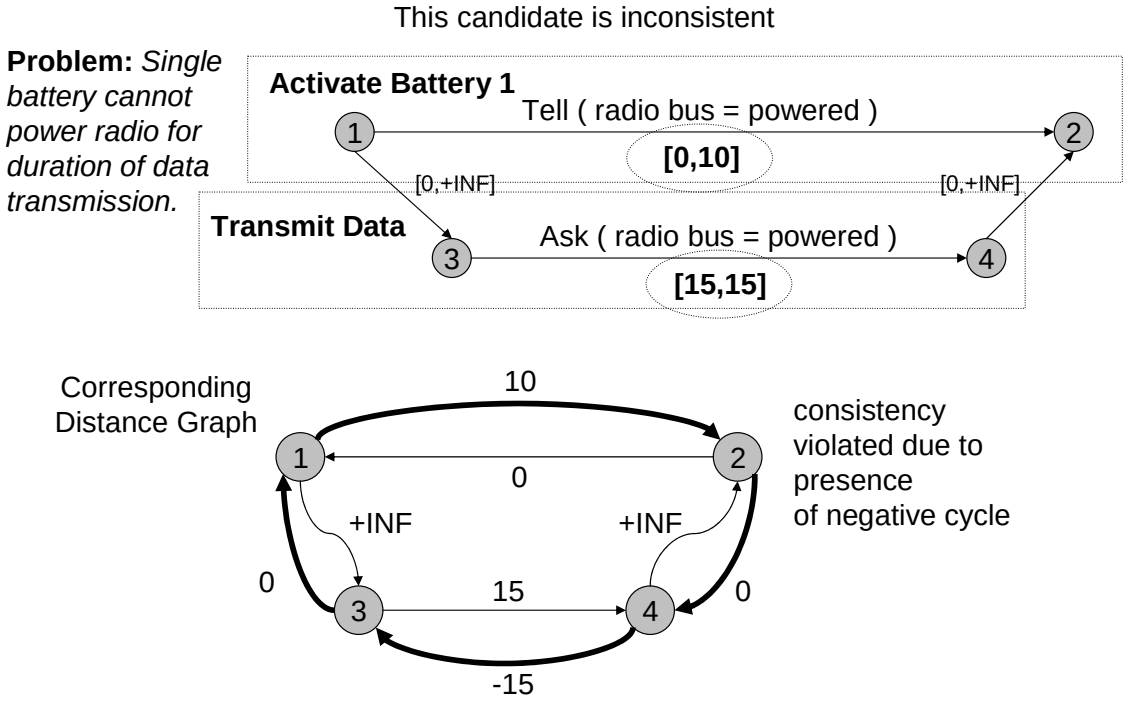


Figure 5-22 Example of Inconsistent Candidate

Episodes are never removed from a candidate, so an inconsistent candidate can never be made consistent. Therefore, Spock improves efficiency by verifying temporal consistency after each candidate is de-queued and pruning inconsistent candidates as soon as they are detected.

As described in Chapter 4, temporal consistency is verified by mapping a TPN to an equivalent distance graph and then checking the distance graph for the existence of negative cycles [8][2]. Spock checks for negative cycles in a TPN's associated distance graph with the FIFO label-correcting algorithm [2]. This algorithm was selected for its simplicity and relatively small O(nm) time-complexity. The pseudo code for the FIFO label-correcting algorithm is shown in figure Figure 5-23.

```
FIFO-label-correcting (N,s)
1.   For i = 1 to |nodes of N|
2.     d(i) = +INF
3.     examined_count(i) = 0;
4.   End-For
5.   d(source) = 0;
6.   list = {source};
7.   while (list is non-empty)
8.     i = pop head of list;
9.     examined_count(i)++;
10.    if examined_count(i) > n
11.      return "Negative Cycle"
12.    End-If
13.    For each arc (i,j) in N
14.      If d(j) > d(i) + c(i,j)
15.        d(j) = d(i) + c(i,j);
16.        If j is not in list
17.          push j to end of list;
18.        End-If
19.      End-If
20.    End-For
21.  End-While
22.  return "No Negative Cycles"
```

Figure 5-23 FIFO Label Correcting Algorithm for Detecting Negative Cycles in a Distance Graph

As the changes to a candidate TPN are small from one iteration to the next, it makes sense to check temporal consistency using an incremental algorithm that reuses work from past iterations. An algorithm for performing incremental temporal consistency checking, ITC, has recently been introduced by I-hsiang Shu [28]. While not integrated in the current implementation of Spock, using ITC to perform temporal consistency checking is discussed in future work Section 6.3.1.

5.5 Continuation: Combining Equivalent Tell Constraints

In the domain of temporal planning, it is important to consider cases where multiple temporally short activities are needed to close a single temporally long activity (see Figure 5-24). In Spock, this occurs when a single Tell constraint's upper time-bound is not long enough to close some Ask constraint. When this happens, multiple Tell constraints may be chained together to close the Ask constraint. This is accomplished by adding an episode from the start of each Tell to the end of the opposite Tell, requiring that the first Tell not end until the second (or *continuing*) Tell has begun.

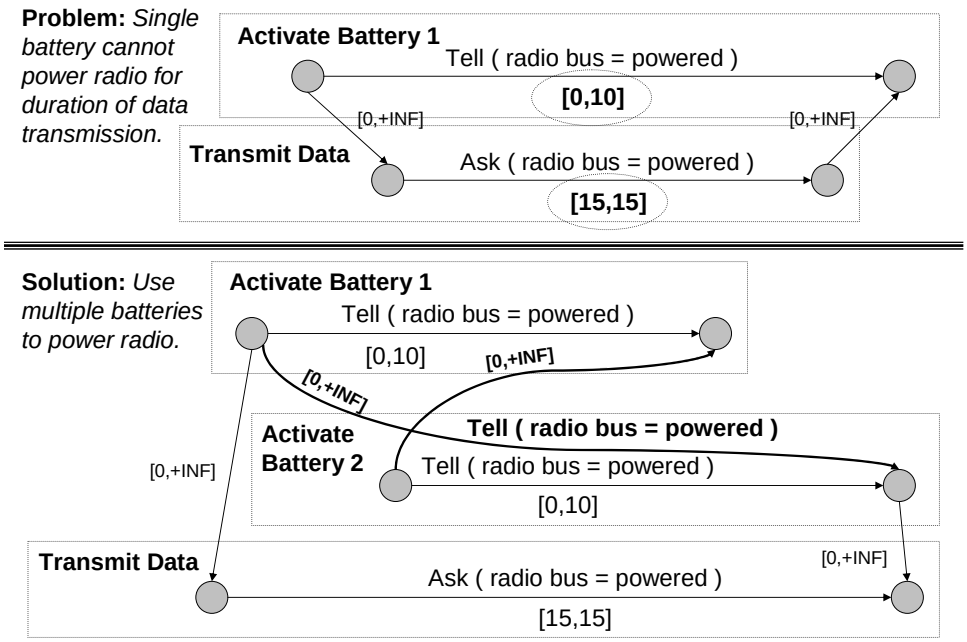


Figure 5-24 Continuation: Problem and Solution

An example of continuation is shown in Figure 5-24. In this example, a "Transmit Data" activity requires that a radio bus be powered for the duration of the transmission (15 time units). However, a single battery can only power the bus for at most 10 time units. Thus the solution is to sequence two batteries such that the radio bus is powered for the necessary 15 minutes. The continuation arcs that Spock adds constrain the two "Activate Battery" activities to overlap, ensuring that the radio bus will be powered without interruption for the required 15 minutes.

Continuation is implemented within Kirk's episode insertion sub-routine (see Figure 5-25). When the episode insertion procedure handles an episode's Tells, it checks to see if those Tells match other active Tells within the plan candidate TPN. When it finds a match, Spock branches. One branch invokes the continuation, while the other branch omits the continuation.

```
Insert (candidate, episode)
1.   for each Ask, A, in episode
2.     let S = choose supporting active Tell in candidate
3.     add causal link to candidate from S.start to A.start
4.     add causal link to candidate from A.end to S.end
5.   end-for
6.   for each Tell, T, in episode
7.     for each Tell, C, in candidate that conflicts with T
8.       add causal link to candidate from C.end to T.start
9.     end-for
10.    if candidate contains matching active Tell, C
11.      optionally:
12.        add episode to candidate from C.start to T.end
13.        add episode to candidate from T.start to C.end
14.        add Tell to candidate from C.start to T.end
15.        add Tell to candidate from T.start to C.end
16.      end-option
17.    end-if
18.  end-for
```

Figure 5-25 Insert Episode Pseudo-Code with Support for Continuation

When a continuation is invoked, episodes are inserted from the start node of each Tell to the end node of the opposite Tell, requiring that the two Tells overlap in time (see Figure 5-26). Each new episode is labeled with a Tell constraint that represents the combined Tell constraints for the entire duration of the continuation. These new Tell constraints can be used in future iterations to close Asks that require support from the chain of Tell constraints.

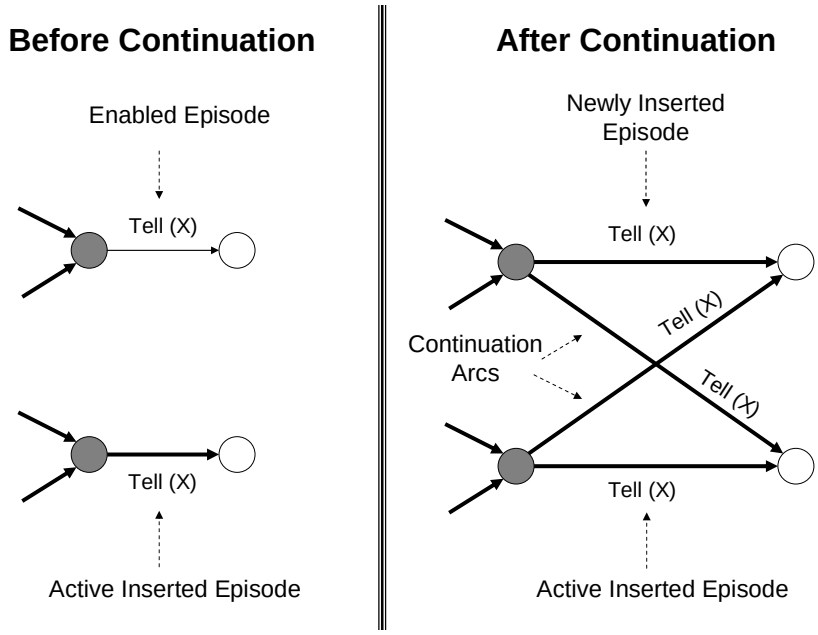


Figure 5-26 Continuation Example

5.6 Ensuring Systematicity

Spock would be much less efficient if it were to revisit previously considered planning states. This is addressed in Spock by memoizing each insertion as it is considered, and disallowing any previously considered candidate expansion. Figure 5-27 motivates the need for an efficient systematic search process. Suppose that a 0 represents the absence of a candidate plan expansion, while a 1 represents the inclusion of a candidate plan expansion. Spock's system of memoizing and prohibiting previously considered plan expansions is analogous to the search tree shown on the right side of the figure.

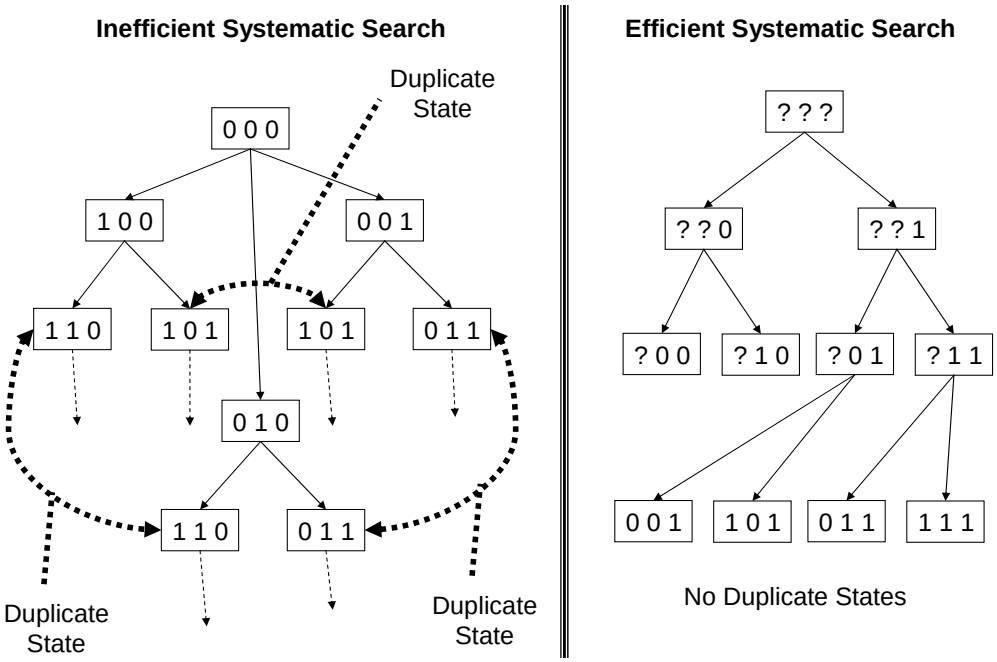


Figure 5-27 Motivation for Systematicity

To achieve systematicity, Spock's child expansion function generates two child candidates as it considers each expansion. One candidate applies the expansion that Spock selected, while the other does not apply the selected expansion. In both children, the selected expansion is memoized and prohibited from future consideration (we call this *blocking* an expansion). The child for whom the expansion was not applied is then considered for alternate enabled expansions. This process repeats until no remaining expansions are possible, ensuring search completeness. The child-expansion function with blocking is shown in Figure 5-28.

```
Child-Expansion (candidate, activity-library)
returns two child candidates
1.    let c1, c2 = copy of candidate
2.    choose between lines 3, 9, 15:

3.      find enabled activities
4.      let a = choose unblocked enabled activity
5.      instantiate(c1,a)
6.      block(c1,a)
7.      block(c2,a)
8.      return c1, c2

9.      find enabled episodes
10.     let e = choose unblocked enabled episode
11.     insert(c1,e)
12.     block(c1,e)
13.     block(c2,e)
14.     return c1, c2

15.     find enabled events
16.     let e = choose unblocked enabled event
17.     insert(c1,e)
18.     block(c1,a)
19.     block(c2,a)
20.     return c1, c2
```

Figure 5-28 Spock Child Expansion Pseudo-Code with Blocking

5.7 Preserving Search Completeness

With support for systematicity, candidate expansions are blocked after they are considered. Because blocked expansions can never be considered again, we need to show that all possible complete plans will be visited regardless of the order in which expansions are considered. That is, we want to be sure that we won't inadvertently skip a valid child candidate by applying expansions in some particular order.

To analyze this problem, there are two cases that must be considered. The first case is when the expansions available to Spock are all *independent*, and inserting one enabled event or episode will not un-enable the other enabled events or episodes. The other case is when the expansions available to Spock are *dependent*, and inserting one enabled event or episode may cause other enabled events or episodes to become un-enabled.

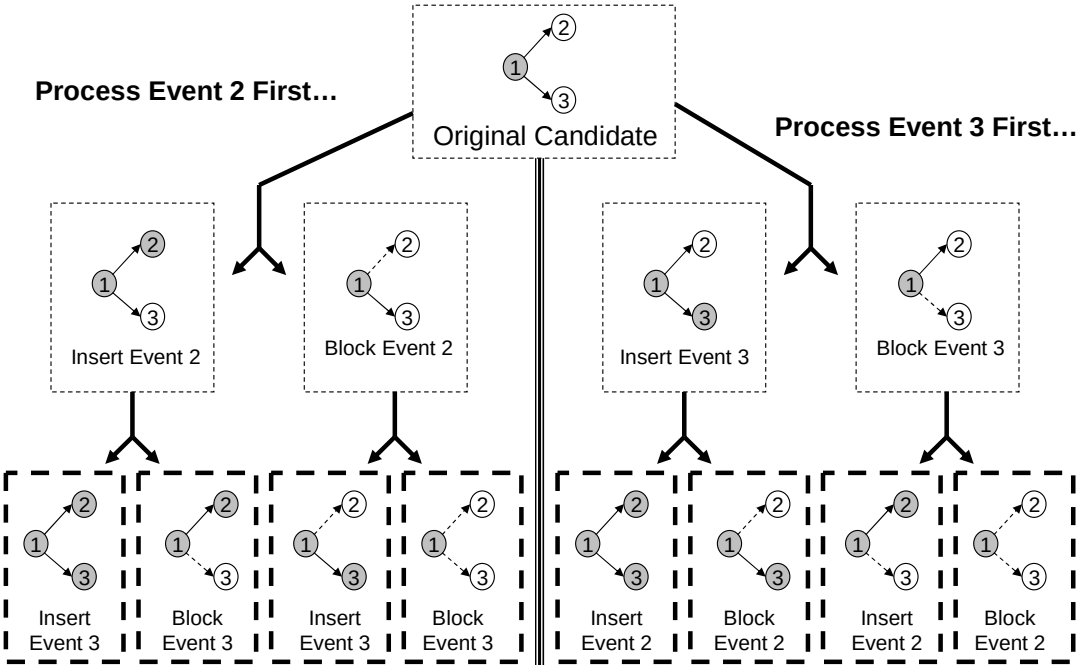


Figure 5-29 Search Tree for Independent Candidate Expansions

In the independent case, search completeness is easy to demonstrate. We know that each enabled event or episode will remain enabled regardless of whether we insert or block another event or episode, thus Spock simply explores all combinations of candidate structures by inserting and blocking each expansion as it is considered (see Figure 5-29). Again, because the insertion or blocking of an expansion will not affect the enablement of other events and episodes, the order in which each expansion is considered is irrelevant.

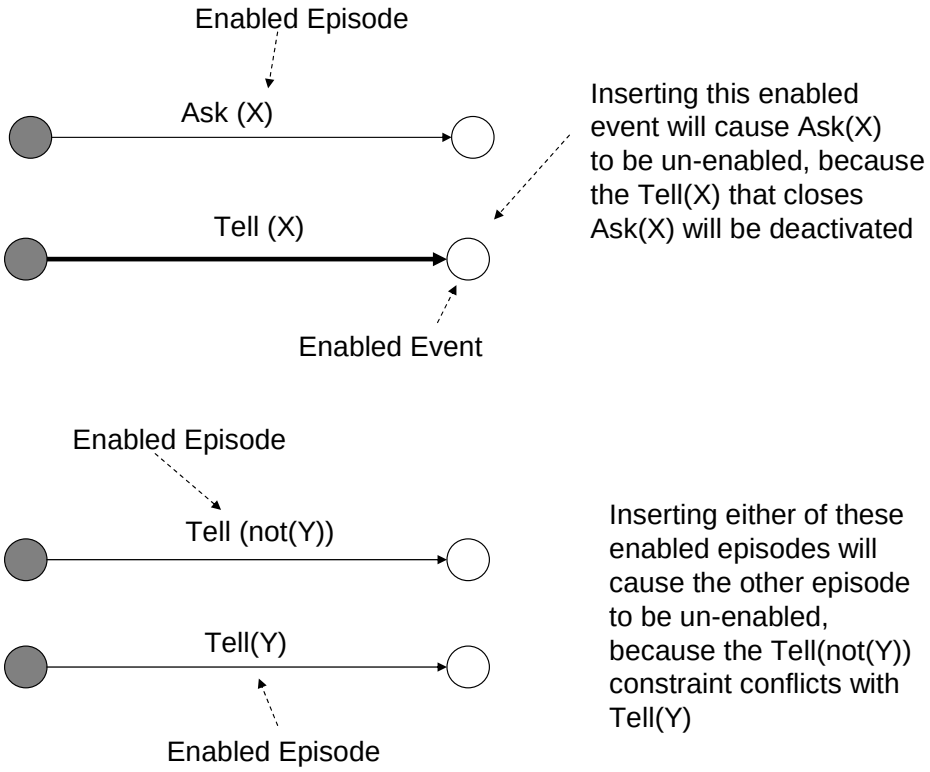


Figure 5-30 Two Cases of Un-enablement

Search completeness is much harder to demonstrate in the case of dependent candidate expansions. Recall that dependent expansions are expansions where inserting one enabled expansion may cause another enabled expansion to become un-enabled. There are two ways in which this may occur (see Figure 5-30). In the first case, one expansion may un-enable another expansion if the first expansion deactivates an active Tell that is needed to close an Ask constraint in the second expansion. The second case is when the first expansion inserts an active Tell that conflicts with a Tell constraint that is part of the second expansion.

We argue that Spock's systematic search with blocking does not remove completeness by analyzing these two cases. For each case, we show that all possible child candidates are generated regardless of the order in which expansions are considered.

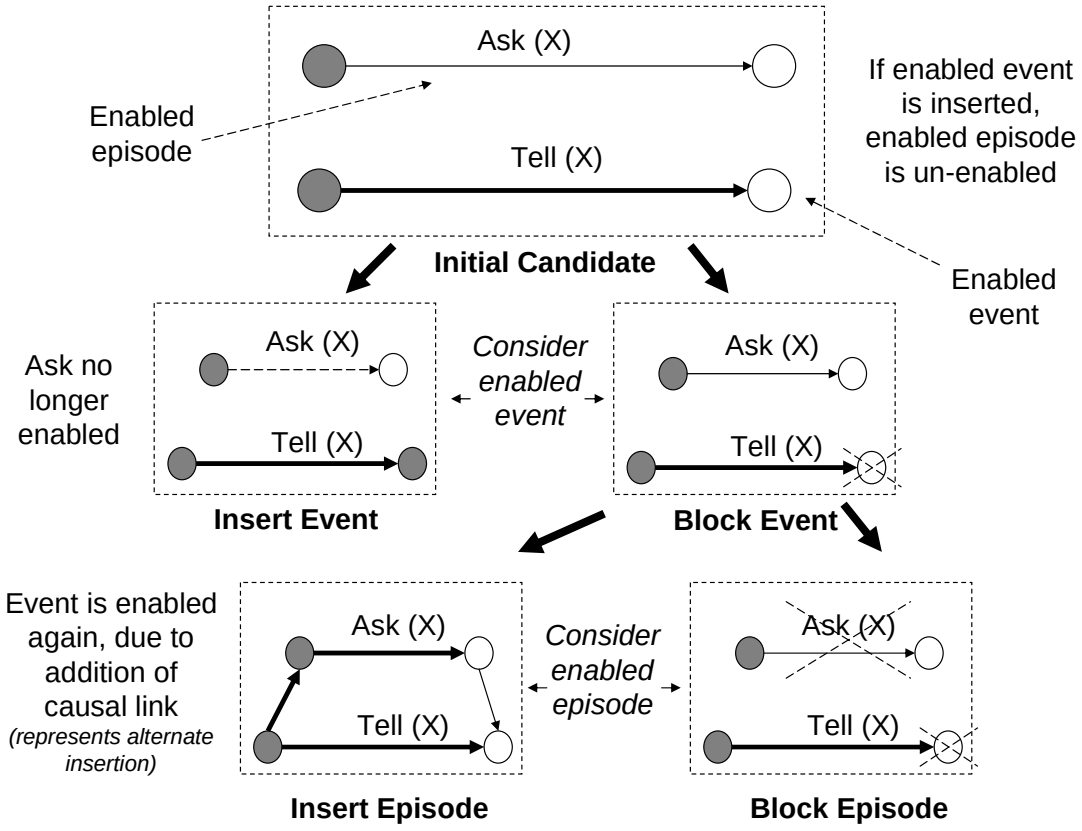


Figure 5-31 Expansion Tree for Enabled Ask Episode and Event

In the case of an enabled Ask episode and an enabled event, we trace the expansion tree as shown in Figure 5-31. Recall that we are interested only in the case where inserting an enabled event or episode will result in another enabled event or episode becoming un-enabled. In the case shown in Figure 5-31, this occurs when the enabled event is considered first. Thus, to show that Spock's child expansion with blocking is complete, we need to show that the expansion tree where the enabled event is considered first will contain all possible child candidates.

There are two distinct child candidates that Spock needs to find. The first is the candidate including the Tell episode, but not the Ask episode. The other is the candidate including both the Tell episode and the Ask episode. We can see that Spock discovers the first required child candidate by simply inserting the enabled event. This de-activates the active Tell and un-enables the Ask episode.

The other required child candidate is reachable via a less direct path. First, Spock, blocks the insertion of the enabled event. Next, it continues by considering the enabled episode and inserting it into the candidate. Recall that when an Ask is inserted into a candidate, causal links are added to ensure that the Ask is contained by the closing Tell. When these causal links are added, the topology of the TPN with respect to the blocked event changes. Because the event was blocked when it had only a single predecessor episode, the addition of the new causal link (which is an episode) means that the blocked insertion no longer applies. This re-enables the event, which can then be inserted at the next iteration, completing the construction of the desired plan candidate.

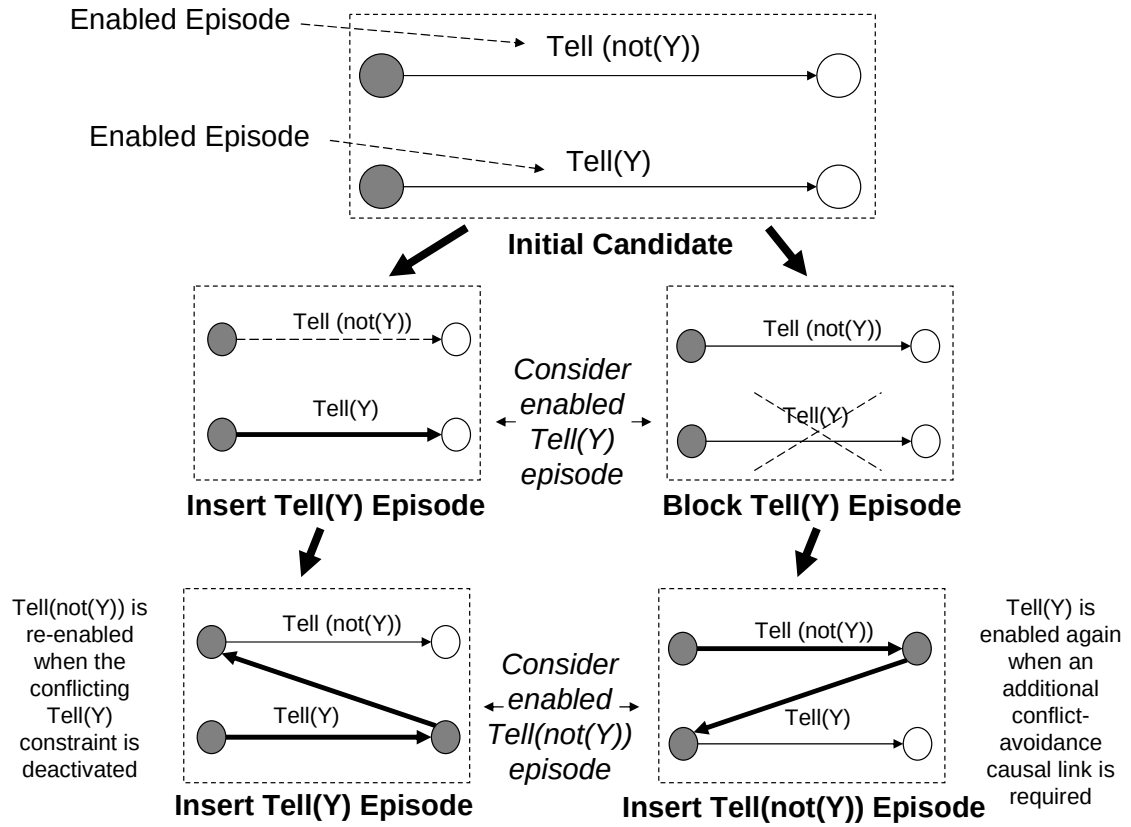


Figure 5-32 Expansion Tree for Conflicting Enabled Episodes

The second case where an insertion may un-enable an enabled event or episode is the case of conflicting Tells (see Figure 5-32). In this case, inserting either enabled episode will un-enable the other episode, as they are mutually exclusive. Due to the symmetry in this example, we need only consider one of the two consideration orderings. Note that, in

either case, we need to show that Spock demonstrates four distinct child candidates. The first candidate contains only the Tell(not(Y)) episode, the second candidate contains only the Tell(Y) episode, the third candidate contains both episodes, with the Tell(not(Y)) being ordered to occur first, and the last candidate also contains both episodes, but with the Tell(Y) being ordered to occur first.

Spock can generate two of the desired candidates as follows. First, the Tell(Y) episode is inserted. This un-enables the Tell(not(Y)) episode, because Tell(Y) is now active and it conflicts with Tell(not(Y)). However, eventually the Tell(Y) episode will be deactivated. At this point, the Tell(not(Y)) episode can be inserted, achieving the case with both episodes where the Tell(Y) episode is ordered to occur first, or the Tell(not(Y)) episode can be blocked, achieving the case with only the Tell(Y) episode.

Spock can generate the other two needed candidates via the following process. First, the Tell(Y) episode is blocked. Next, the Tell(not(Y)) episode is inserted. Once the Tell(not(Y)) episode is completely inserted (and deactivated), the Tell(Y) episode will become re-enabled. Recall that conflict-avoidance causal links are added to a TPN when a Tell is inserted that is inconsistent with some inactive Tell. This means that the inactive Tell(not(Y)) episode will cause a new conflict-avoidance causal link to be added, should the Tell(Y) episode be inserted. Thus, the graph topology respective of the Tell(Y) episode is changed, and the Tell(Y) episode can be re-enabled, in spite of its previous blocking. At this point, Spock can either insert the Tell(Y) episode, achieving the case with both episodes where the Tell(not(Y)) episode occurs first, or it can block the Tell(Y) episode, achieving the case with only the Tell(not(Y)) episode.

As Spock can successfully generate all possible child candidates within its systematic framework regardless of the order in which expansions are considered, we conclude that Spock's planning algorithm is complete.

5.8 Candidate Cost Update

Before a child candidate can be inserted into the priority queue, its cost must be updated.

An important contribution of this thesis is the fact that Spock is an optimal TPN planner. This means that Spock can optimize for minimum possible mission time as well as the number of inserted activities or episodes. In the manner of A* search [26], Spock is designed to eventually support the evaluation of each plan candidate according to a utility function, $f = g + h$. The g component represents the cost of a candidate solution thus far, while the h component is an *admissible heuristic estimate* of the remaining cost to the goal. While g causes Spock to search the plan-space in best-first order, h focuses the search towards likely solutions without sacrificing optimality, thus improving efficiency.

Spock's heuristic cost estimate is not yet implemented (i.e. $h = 0$), however a possible design is included in future work Section 6.3.2. Spock does currently perform an optimal uniform cost search using the cost of a candidate's embedded TPN graph. The remainder of this section discusses the procedure by which Spock calculates a candidate's f value (the cost thus far).

As it is currently implemented, Spock optimizes over total plan execution time. Thus a plan candidate's cost is the minimum time in which the entire plan can be executed. To calculate this value, each event's earliest execution time is determined using Equation 1. Finally, the maximum earliest execution time over all events in the TPN is returned as the candidate's cost.

$$\min time(N) = \max_{i \in predecessors(N)} \left(\min time(i) + lowerbound\,(arc_{i,N}) \right)$$

Equation 1 Candidate Cost Update Equation

The solution to Equation 1 is equivalent to solving a single-source shortest path algorithm over the TPN using an episode's lower time-bound as its cost. Thus Spock could calculate a plan candidate's cost by using any standard single-source shortest path algorithm, such as Dijkstra's algorithm [7] or the FIFO label-correcting algorithm [2].

While a single-source shortest path algorithm would work for performing candidate cost updates, Spock improves the speed at which it performs this task by utilizing the incremental nature of plan candidate expansion. That is, events and episodes are always

inserted into a candidate TPN in chronological order, so Spock can perform cost updates incrementally by simply applying the above equation as each event is inserted into the candidate. This reduces the cost of Spock's cost update procedure from O(mn) to constant time.

5.9 Summary

The Spock generative TPN planning algorithm finds optimal solution plans when given an input control program and activity library. The key contributions of this work are that Spock supports rich activity operators and goal specifications, flexible time-bounds, and optimal planning with arbitrary cost functions.

Spock enforces systematicity by blocking the repeated consideration of candidate expansions, and allows combined Tell constraints to close Ask constraints with long durations. Furthermore, while not currently implemented, Spock's cost update procedure is designed to support a heuristic cost estimate to further focus the search and improve planning speed.

6 Results and Conclusions

This thesis presented the Spock planning algorithm, which enables generative planning with complex processes. Spock provides three key contributions. First, Spock represents operators using the RMPL language that describes behaviors as a parallel and sequential composition of state and activity episodes. Second, Spock uses a uniform operator and plan-space representation of processes in terms of Temporal Plan Networks. Third, Spock uses a forward progression algorithm that walks monotonically forward through plan processes, closing any open conditions and resolving any conflicts.

This chapter concludes by discussing Spock's implementation and performance, and presents possible directions for future research.

6.1 Implementation

The Spock generative TPN planner described in this thesis was implemented in C++ and tested on a Pentium III 700 MHz processor with 256 MB of RAM running RedHat Linux 8.0.

As described in Chapter 1, Spock is part of the Kirk model-based executive for mobile autonomous systems. The primary components of this system include the RMPL compiler [32], the TPN Sequencer [19][28], the Spock generative TPN planner, and the plan runner [23]. The RMPL compiler takes input RMPL files and converts them into Temporal Plan Networks suitable for mission planning. The TPN Sequencer identifies a consistent goal / strategy plan that establishes the guidelines for a particular mission. Spock takes this goal plan (control program) and forms a solution plan by combining the goal plan with a set of activities from the activity library. Finally, the solution plan is passed to the plan runner, which schedules activities and executes primitive commands on the vehicle hardware.

While Spock's planning algorithm is complete, it still needs to be integrated with the rest of the Kirk model-based executive. Currently Spock's inputs are given as hand-coded

TPN structures, and solution plan TPNs are dumped to a text file. However, integration with the rest of the Kirk model-based executive will be completed in the near future.

The Spock implementation described in this thesis contains five primary C++ classes: Spock, Candidate, Activity, Event, and Episode. The Event and Episode classes are self-describing. The Activity class represents an activity in the activity library, and thus contains a collection of events and episodes. The Candidate class corresponds to a plan candidate, and thus contains the candidate's solution TPN, as well as a set of pending events and episodes. The Candidate class also contains the methods that perform enablement checking, consistency verification, completeness checking, and cost updating. Finally, the Spock class contains the top-level search algorithm that passes around the plan candidate objects and calls the appropriate methods when necessary.

6.2 Performance

Spock was run on a series of seven test problems to chart its effectiveness. The smaller problems (1-4) were used to validate Spock's correctness, while the larger problems (5-7) demonstrated Spock's applicability to actual autonomous vehicle control scenarios.

Problem	Events in Solution	Episodes in Solution	Number of Candidates Generated	Time to Solve
1	6	5	8	0.04s
2	10	13	11	0.14s
3	9	11	14	0.14s
4	11	13	17	0.11s
5	16	32	68	0.67s
6	20	44	299	2.35s
7	16	30	892	15.21s

Table 6-1 Performance of Spock Generative TPN Planner

The larger test problems (5-7) are all instances of a forest-fire rescue scenario (see Figure 6-1). In these scenarios, an unmanned fire-fighting aerial vehicle (UFFAV) is stationed at a base waypoint. A forest-fire is located at a waypoint within the forest, and the

UFFAV must fly to the forest-fire, put it out, and depending on the scenario, also return to base.

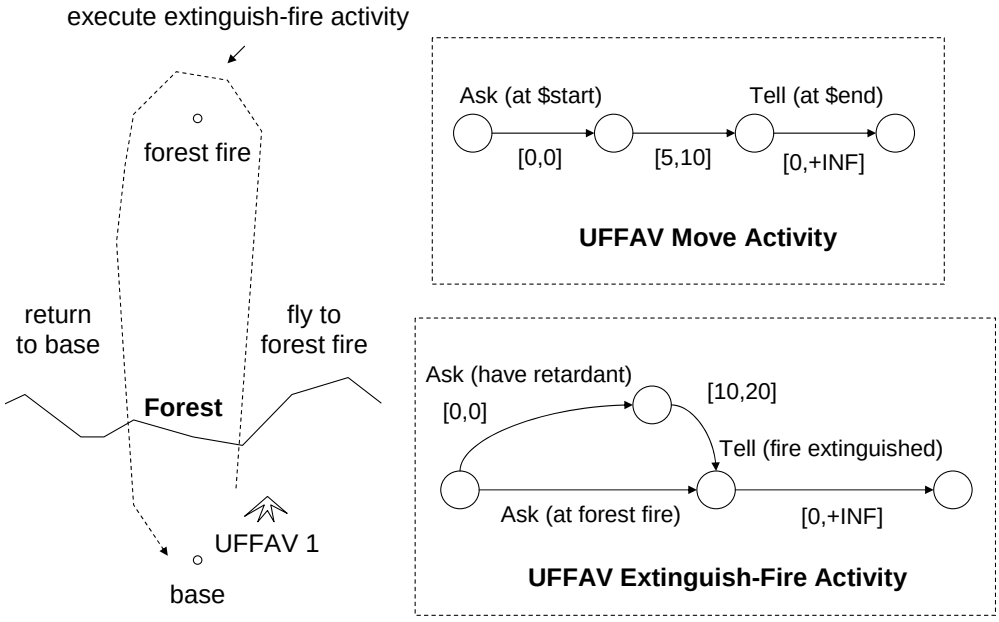


Figure 6-1 Test Scenario with Activity Library

In the demonstration problems, the UFFAV uses two activity operators to complete its mission: Move, and Extinguish-Fire (see Figure 6-1). Move moves the UFFAV from one waypoint to another, while Extinguish-Fire commands the UFFAV to put out the fire, requiring that the vehicle be at the forest-fire waypoint.

The current implementation of Spock validates Spock's planning representation and its ability to plan with complex processes. Fast planning has not yet been demonstrated, as Spock does not yet include a relaxed graph heuristic cost estimate. Correspondingly, Spock was able to successfully find solution plans when tested on small example problems.

Besides its lack of a relaxed graph heuristic cost function, Spock is also currently slowed by inefficient helper functions. One example of this is Spock's child expansion function, which copies candidates in their entirety each time it branches. This process is very

inefficient and consumes unnecessary time and memory. Additionally, Spock detects enabled events and episodes using a simple search process that is not efficient within an iterative context. These searches consume a large amount of time per iteration, and circumventing them should yield a significant performance improvement. Solutions to these performance bottlenecks are discussed in Section 6.5.

6.3 Future Work

While Spock's implementation is complete, there are areas in which performance or functionality could be improved. This section describes three significant improvements: incremental temporal consistency checking, the incorporation of a relaxed plan-graph heuristic cost estimate, and extending RMPL to handle more types of timing constraints. Finally, a selection of implementation efficiency improvements is described.

6.3.1 Incremental Temporal Consistency Check

TPN temporal consistency checking is not very computationally expensive. For example, the FIFO label-correcting algorithm runs in $O(nm)$ time, where n is the number of nodes in the distance graph, and m is the number of arcs. However, as a planner like Spock is always extending TPNs, the frequency with which a temporal consistency checker is called warrants trying to improve its runtime. One way to improve Spock's runtime is to use an incremental algorithm to check temporal consistency.

An incremental algorithm updates its answer by reasoning about problem changes and changes in their consequences. When the problem changes and resulting changed consequences are small relative to that of the overall problem, a significant performance gain is achieved. Examples of incremental algorithms include a truth maintenance systems [9][22] and Incremental A* [20].

Candidate TPNs are modified only slightly during each child expansion. Thus, it makes sense to check temporal consistency with an incremental temporal consistency checking algorithm. Recently, I-hsiang Shu has developed an incremental consistency checking algorithm, ITC [28]. ITC verifies temporal consistency in minimal time by using an

efficient verification algorithm that reuses existing work from previous iterations. It uses the idea of a set of support from truth maintenance systems to identify edges that require updating. Finally, it uses an update rule from a modified FIFO label-correcting algorithm to perform this update incrementally.

ITC has been shown to decrease validation time by an order of magnitude over non-incremental methods when applied to incremental problems. In the future, Spock's candidate consistency checking sub-routine could use this type of consistency checking algorithm to help maximize planning speed.

6.3.2 Relaxed Plan Graph Heuristic Cost Estimate

The uniform cost search currently implemented within Spock can inefficiently explore regions of plan-space that are unlikely to yield a desirable solution. The reason for this is that uniform cost search does not include any estimate of how close the candidate is from achieving the goal. To focus the search, Spock should be extended to use an admissible heuristic estimate of the remaining cost from a plan candidate to the goal (the *g-value*). The admissible heuristic estimate transforms the uniform-cost search to an informed search process, increasing efficiency [26].

For a cost estimate to be *admissible*, it must be less than or equal to the true cost of achieving the goals. If a search algorithm uses an inadmissible heuristic, it may overestimate the cost of a particular plan candidate. As search algorithms examine least-cost candidates first, overestimating the optimal solution may cause the search algorithm to discover a sub-optimal solution first. Thus heuristic estimates that underestimate (that is, admissible heuristics) are essential for ensuring an optimal search process.

Within the space of admissible heuristics, we still want an estimate that is as close to the true cost as possible. The farther an estimate is from the true cost to the goal, the less information it contains and the less focused the search. Thus Spock needs an admissible, yet close estimate of the remaining cost of a plan candidate to the goal.

Recently, advances in planner efficiency have been demonstrated by the FF [16] and HSP [6] planners through the use of an admissible heuristic cost estimate called a *relaxed plan graph*. Relaxed plan graphs are similar in structure to standard plan graphs, however they do not prohibit mutually exclusive facts, and facts persist indefinitely after they are created. This allows a relaxed plan-graph to insert all enabled actions at each action layer, eliminating the need for branching. We can apply the ideas behind the relaxed plan-graph to Spock's plan candidates in order to generate an admissible heuristic estimate of the cost remaining for a particular plan candidate. Constructing a relaxed plan-graph in Spock would be performed as follows.

First, note that the input to the heuristic estimate function is a current state in plan-space (a candidate), and the output is an admissible, yet close estimate of the cost to the goal. In Spock, a plan candidate is complete when all of its events and episodes are inserted. Thus, the relaxed plan-graph's goal state is a state where all initially-pending events and episodes are inserted. Any events or episodes that are added to the pending sets during the heuristic estimate calculation do not need to be inserted for the relaxed plan graph to be a goal state. This is to protect admissibility, as most relaxed plan-graphs will attempt to include superfluous plan actions, and we only want necessary plan actions to contribute to a relaxed graph's cost estimate.

A key feature of relaxed graphs is that they can be constructed quickly to determine a cost estimate. Constructing a relaxed graph require polynomial time [14], which implies an algorithm that avoids decision making and expensive search procedures. While the elimination of decision making causes a relaxed graph to become inconsistent, this is not a problem as the relaxed graph is meant to be used as a heuristic, or imperfect, cost estimate.

Constructing a relaxed graph is the same as constructing a normal candidate TPN, with three rule modifications that are designed to eliminate decision making. First, active Tells never become deactivated. This allows relaxed graphs to monotonically add Tell constraints. Note that deactivating a Tell would require a decision point, as Tell deactivation may un-enable some enabled Ask episode. Second, conflicts between Tell

constraints are ignored. This is because de-conflicting active Tells also requires decision making. Finally, at each iteration, all possible expansions are applied. Recall that in a relaxed graph, active Tells never deactivate and conflicting Tells are ignored. It follows that all expansions therefore behave independently. Thus, we are required to apply all possible expansions at each iteration, as we do not want to make any decisions, including the decision of which expansion to apply.

The set of active Tells in a relaxed graph monotonically increases, because active Tells never deactivate. Thus, plan operators only need to be inserted into a relaxed plan-graph at most once. This is useful because there are a finite number of plan operators, and we can therefore guarantee that a point will be reached where either the goal is discovered, or the solution is determined to be infeasible.

Finally, consistency is not checked between iterations of relaxed graph construction. This is because resolving temporal or Tell inconsistencies would require a decision-making process to attempt the exclusion of some violating episode.

When all originally-pending events and episodes are inserted, the relaxed graph construction is complete. At this point, heuristic cost is estimated by returning the maximum of the minimum execution times assigned to each event.

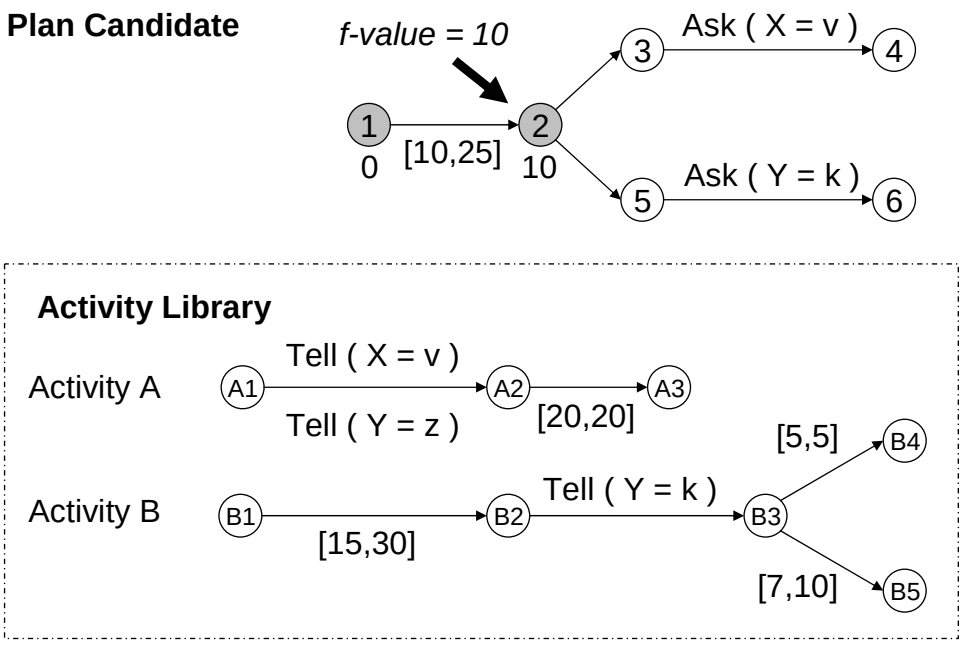
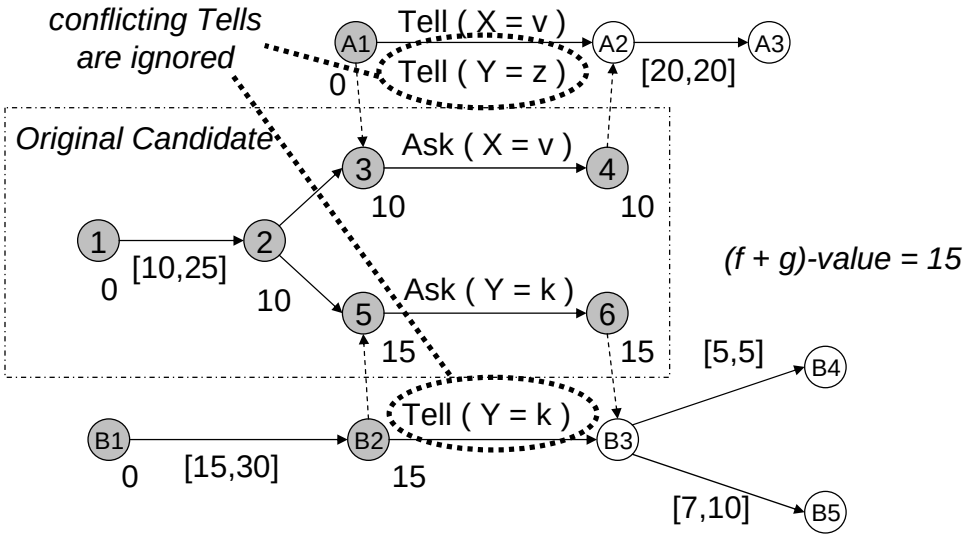


Figure 6-2 Example Candidate Graph before Heuristic Cost Estimation

Relaxed Candidate Graph



The Ax and Bx nodes do not need to be inserted, as they were only instantiated as part of the relaxed candidate graph

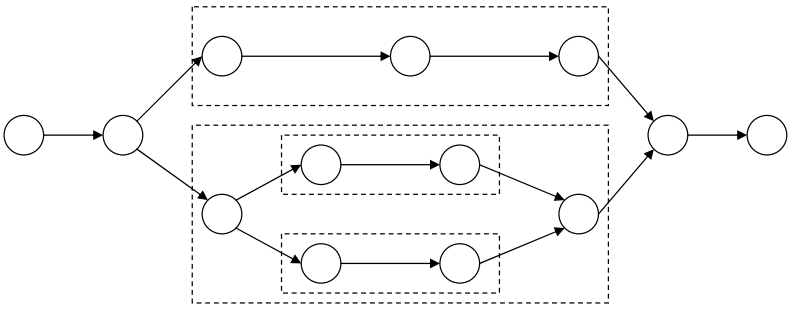
Figure 6-3 Relaxed Candidate Graph with Heuristic Cost Estimate

Figure 6-2 shown a plan candidate and associated activity library. An example complete relaxed graph based on this plan candidate is shown in Figure 6-3. First, note that several of the activity nodes in the relaxed graph are not inserted, even though the relaxed graph is complete. This ensures admissibility by making the relaxed graph an underestimate of the cost to the goal. Next, note that two of the inserted Tell constraints conflict. This is because relaxed graphs do not require Tell consistency. Finally, note that the cost values for the candidate events are propagated in the same manner as in normal plan candidate graphs.

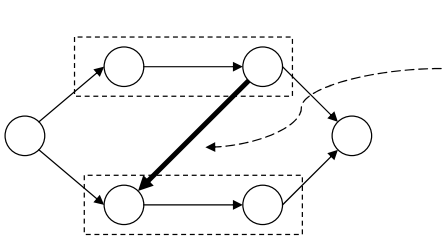
By modifying Spock's candidate expansion algorithm to avoid decision making, Spock can support an admissible relaxed-graph heuristic cost function. This cost function focuses Spock's search towards optimal solution plans, improving efficiency.

6.4 Extending RMPL to Support Additional Temporal Constraints

RMPL builds processes using the standard constructs of parallel and sequential composition. These combinators allow the natural expression of complex operators in terms of intended state evolutions. In some cases a richer vocabulary is required for describing the temporal relationships between episodes (see Figure 6-4). This vocabulary is provided by qualitative and metric temporal algebras [1].



This TPN structure can be generated with RMPL



This TPN structure motivates additional RMPL constructs

Figure 6-4 RMPL TPN Limitation

Extending RMPL to handle arbitrary temporal constraints can be accomplished by adding a new RMPL construct to the language, with a naming device allowing programmers to refer to sections of RMPL code in locations other than where they are initially defined. These constructs could take a form similar to Allen's temporal relations [1], or some other intuitive method for ordering sub-activities within an RMPL method.

Expanding RMPL's vocabulary presents two technical challenges. First, care must be taken to ensure that the simplicity of RMPL's process descriptions is not corrupted, as this is RMPL's key feature. This can be accomplished by simply adding one or two additional combinators while preserving the rest of RMPL's syntax. As long as the method in which RMPL activities and control programs are developed is not changed, RMPL's significant features will be preserved.

Second, the planning algorithms that process RMPL programs need to be adapted to handle the new construct. Spock was designed with this improvement in mind, and thus already supports the advanced TPN constructions. Kirk's strategy selection algorithm,

however, will need to be revised, as it assumes a TPN that is encapsulated as shown in the top portion of Figure 6-4.

6.5 Implementation Efficiency Improvements

Spock is a complicated planning algorithm, and as such, relies on several smaller algorithms in order to solve its planning problems. Some of these sub-algorithms are currently implemented in ways that motivate implementation improvements.

Currently, Spock's enablement checking algorithm searches a plan candidate's entire pending event and episode sets in order to determine which components are enabled for expansion. This process takes polynomial time, which is expensive within Spock's iterative planning process. A better approach would be to track active Tells, enabled events, and enabled episodes using support links in the style of truth maintenance systems [22][9], in order to efficiently determine which pending events and episodes become enabled or un-enabled during the expansion process. Such an approach could theoretically result in a linear time enablement checking algorithm, which would result in a significant performance improvement.

Another inefficiency in Spock is that it copies entire plan candidates when it branches. This uses a lot of space, and the time it takes to copy a plan candidate negatively impacts runtime. A better solution would have plan candidates share common components, requiring new child candidates to contain only the events and episodes that differentiate them from their parent candidate. This approach would require modifications to Spock's planning algorithm in the way it interacts with a candidate, and possibly a garbage-collection sub-routine that would be responsible for deleting parent candidates whose children have all been pruned. However, the result of this improvement would be a significant reduction in space and time consumption.

Finally, we note that Spock does not partition its event and episode sets based on time. Spock's enablement-checking function only needs to consider interactions between pairs of events when they can co-occur. Thus, a temporal indexing storage mechanism would be useful. A temporal indexing storage device would store Spock's events and episodes,

but organize them based on their possible execution times. This would allow Spock to compare episodes only if they might co-occur, reducing the search space for enablement checking.

6.6 Summary

This thesis provides a generative planning algorithm that supports temporally-flexible planning with complex processes. We achieve this through three key contributions. First, we describe operators and goal behaviors as the concurrent evolution of actions and states, comprised of behavioral episodes that are combined through sequential and parallel composition. Second, goal behaviors, operators, and plan-space are all represented uniformly during the planning process as Temporal Plan Networks. Finally, planning in Spock is a forward progression process that walks over the goal TPN, moving forward in time, while closing open conditions by inserting activity TPNs as needed.

References

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

[2] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[3] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. *New Directions in Planning, M. Ghallab and A. Milani (Eds.),* pages 141-153, 1996.

[4] G. Berry, S. Moisan, J. P. Rigault. ESTEREL: Towards a Synchronous and Semantically Sound High-Level Language for Real-Time Applications, *Proc. IEEE Real-Time Systems Symposium, IEEE Catalog 83CH1941-4*, pp. 30-40, 1983.

[5] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281-300, 1997.

[6] B. Bonet and H. Geffner. Heuristic search planner 2.0. *AI Magazine*, 22(3):77-80, Fall 2001.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61-95, May 1991.

[9] J. Doyle. A Truth Maintenance System. *Artificial Intelligence* 12:231-272, 1979.

[10] R. E. Fikes, N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189-208.

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

[12] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research: Special Issue on the 3rd International Planning Competition,* 2003.

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

[14] Alfonso Gerevini and Ivan Serina. LPG: A planner based on local search for planning graphs. *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS'02),* 2002.

[15] B. Hayes. The Naïve Physics Manifesto. *Expert Systems in the Micro-electronic Age, D. Michie (ed.),* Edinburgh University Press, May 1979.

[16] Jorg Hoffman and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research,* 14:253-302, 2001.

[17] Ari Jonsson, Paul Morris, Nicola Muscettola, Kanna Rajan, and Ben Smith. Planning in interplanetary space: Theory and practice. *Proceedings of the 5th AIPS, Breckenridge, CO,* 2000.

[18] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. *Proceedings of IJCAI-1999,* 1999.

[19] Philip K. Kim, Brian C. Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. *Proceedings of IJCAI-2001*, 2001.

[20] S. Koenig and M. Likhachev. Incremental A*. *Advances in Neural Information Processing Systems, 14*, 2001.

[21] Derek Long and Maria Fox. Exploiting a graphplan framework in temporal planning. *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, 2003.

[22] D. McAllester. Truth Maintenance. *Proceedings of AAAI-90*, pp. 1109-1116, 1990.

[23] N. Muscettola, P. Morris, B. Pell, B. Smith. Issues in Temporal Reasoning for Autonomous Control Systems. *Proc. 2nd International Conference on Autonomous Agents*, Minneapolis, MN, 1998.

[24] Dana Nau, et. al. Total-order planning with partially ordered subtasks. *Proceedings of IJCAI-2001*, 2001.

[25] 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.

[26] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[27] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26, 1996.

[28] I-hsiang Shu. Enabling fast flexible planning through incremental temporal reasoning. Master's thesis, Massachusetts Institute of Technology, 2003.

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

[30] David E. Smith, Jeremy Frank, and Ari K. Jónsson. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review*, 15(1), 2000.

[31] B. C. Williams. Doing Time: Putting Qualitative Reasoning on Firmer Ground. *Proc. Of the 5th National Conference on Artificial Intelligence,* pp. 105-112, 1986.

[32] B. C. 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.

[33] Brian C. Williams, Michel D. Ingham, Seung H. Chung, and Paul H. 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.