

*

Model-based Planning for Coordinated Air Vehicle Missions

by

Philip K. Kim

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 8, 2000

Copyright 2000 Philip K. Kim. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author _____
Department of Electrical Engineering and Computer Science
August 8, 2000

Certified by _____
Brian C. Williams
Thesis Supervisor

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

Acknowledgements

This thesis could not have been completed without the help of several important people. First, I would like to thank Professor Brian Williams for providing technical guidance and for sacrificing so much of his personal time, especially in the final hours, to help me complete this work. I would also like to express my gratitude to the members of the Decision Systems Group at Draper Laboratory for sharing with me their knowledge of unmanned combat air vehicle missions. In particular, I greatly appreciate the help of Mark Abramson for not only helping me to understand the requirements of UCAV missions, but also for giving me valuable feedback and helping to keep me on track. Finally, I am grateful to my family and friends for their love and support throughout this past year.

This thesis also could not have been completed without the sponsorship of the Office of Naval Research under contract N00014-99-1080, MIT contract number 6890056.

Philip Kim
August 2000

Table of Content

ACKNOWLEDGEMENTS	2
TABLE OF CONTENT.....	3
CHAPTER 1.....	5
INTRODUCTION.....	5
MOTIVATION.....	5
APPLICATION	6
<i>Multi-UCAV Missions.....</i>	<i>6</i>
<i>Scenario Example</i>	<i>7</i>
<i>Mission Characteristics</i>	<i>7</i>
PROBLEM STATEMENT	8
<i>Activity Modeling Language.....</i>	<i>9</i>
<i>Temporal Planning Network.....</i>	<i>9</i>
<i>Model-based Temporal Planner</i>	<i>10</i>
THESIS LAYOUT	10
CHAPTER 2.....	11
BACKGROUND	11
REACTIVE MODEL-BASED PROGRAMMING LANGUAGE.....	12
<i>Hierarchical Constraint Automata</i>	<i>13</i>
SIMPLE TEMPORAL NETWORKS	15
<i>Representation</i>	<i>16</i>
<i>Distance Graph Analog</i>	<i>17</i>
PLANNING OVERVIEW.....	19
<i>STRIPS Activity Models</i>	<i>20</i>
<i>Partial Order Planning.....</i>	<i>21</i>
<i>Hierarchical Planning</i>	<i>23</i>
<i>State-space and Plan-space Planning</i>	<i>23</i>
<i>Temporal Planning</i>	<i>24</i>
CHAPTER 3.....	26
ACTIVITY MODELS	26
OVERVIEW	26
EXAMPLE	27
ACTIVITY MODELING LANGUAGE.....	29
<i>Basic Combinators.....</i>	<i>30</i>
<i>Derived combinators.....</i>	<i>33</i>
TEMPORAL PLANNING NETWORK	34
<i>Symbolic Constraints</i>	<i>36</i>
<i>Decision Nodes</i>	<i>38</i>
<i>Composition</i>	<i>38</i>
AML TO TPN MAPPING	40
UCAV ACTIVITY MODELS	42

<i>AML Scoping</i>	43
<i>Relative Duration Bounds</i>	43
<i>Vehicle Activities</i>	43
<i>Group Activities</i>	45
SUMMARY	46
CHAPTER 4	47
PLANNING ALGORITHM	47
OVERVIEW	47
PHASE ONE: SELECT PLAN EXECUTION	51
<i>Network Search</i>	51
<i>Temporal Constraint Consistency</i>	55
<i>Negative Cycle Detection</i>	56
PHASE TWO: REFINE PLAN	58
<i>Symbolic Constraint Consistency</i>	58
<i>Conflict Detection</i>	58
<i>Conflict Resolution</i>	61
PHASE THREE: HIERARCHICAL DECOMPOSITION	64
CHAPTER 5	67
CONCLUSIONS	67
RESULTS	67
<i>Planner Implementation</i>	67
<i>Performance</i>	69
FUTURE WORK	70
<i>Handling Contingencies</i>	70
<i>Improving the Activity Models</i>	71
<i>Optimizing the Planner</i>	72
SUMMARY	73
REFERENCES	74
APPENDIX A	77
AML ACTIVITY DESCRIPTIONS	77
APPENDIX B	81
<i>TPN specification format</i>	81
APPENDIX C	84

Chapter 1

Introduction

Motivation

The demand for vehicles to carry out complex tasks with little or no supervision has motivated a great deal of past and current research in the area of intelligent autonomy. The ability for a vehicle to act autonomously would be advantageous in contexts where human supervision and control is not possible. Onboard human control may be undesirable, for example, in a situation where the environment is dangerous or unknown, or it may be infeasible because the vehicle is physically unable to support a human. In addition, remote control might not be an option if there is no means to support communication, or if the communication latency or unreliability simply renders it ineffective.

Embedded autonomy has already proven both feasible and useful through successes such as the Remote Agent Experiment, which was conducted onboard NASA's Deep Space One spacecraft in May 1999. During this experiment, the Remote Agent control software was allowed to assume full command of the in-flight spacecraft, and demonstrated the robustness of its onboard planning, execution, and mode-identification capabilities [14].

There is also an emerging interest in multiple vehicle autonomy for applications that require capabilities that are impossible, inefficient, or not cost-effective using a single vehicle. One such example is military combat missions, which are typically carried out by groups of vehicles because it increases the probability of successful completion while decreasing the likelihood of vehicle damage or destruction. In the domain of space exploration, there has been recent interest in developing a tightly coordinated group of spacecraft to be used for long-range space interferometry, which would have a much greater range than current single-spacecraft interferometers.

In order for a vehicle to autonomously perform the set of activities necessary to complete a mission, it needs to make decisions. Autonomous systems achieve robustness by having at their disposal a range of alternative methods of performing activities. Whenever the vehicle encounters an activity that allows for several alternative methods, the vehicle must decide which one to employ. Furthermore, if the vehicle has control over the time and duration of activities, then it must also decide when each activity should be performed and for how long of a period. The problem is that there may be particular sets of decisions that lead to a state from which there is no possible way of successfully completing the mission. Some reasons for failure include the exhaustion of some un-renewable resource, conflicting activities being scheduled to perform at the same time, or simply too little time left to complete all the necessary activities. Planning, therefore, is an essential capability because it makes the decisions ahead of time to avoid failure situations.

The problem of planning for multiple-vehicle missions presents some major challenges. First, each mission plan must be expressive enough to fully describe complex coordinated behaviors. For example, the plan must be able to express that two vehicles should meet at some location and proceed together, and that multiple vehicles should not be transmitting messages on a single communication channel at the same time. Second, the plan must be flexible enough to handle variations in plan execution due to exogenous factors. A plan could be represented as a set of time-stamped commands that indicate exactly when each activity must commence and complete, but this type of plan is too brittle because it may fail if even a single activity takes even slightly more or less time than expected. The plan must also be able to express contingencies in order to be flexible with respect to uncontrollable factors. Third, in order to support automated planning, there must be some way of capturing knowledge about the domain and about vehicle capabilities using a representation that is compact enough to be manageable and modular to support reusability. Finally, the planner must be efficient enough to support reactive on-board re-planning over execution horizons ranging from as long as hours to as short as seconds.

Application

The target application of the research described in this thesis is the planning of Unmanned Combat Air Vehicle (UCAV) missions. For the purposes of this research, a UCAV was broadly defined as an autonomous aircraft with the ability to deliver munitions to attack air or ground targets. The actual form of the vehicle, its speed and maneuverability, and the specific resources available to it were abstracted out to a large degree. Instead, the focus was on the types of missions in which they might be deployed and the types of coordination necessary to carry out those missions. The information presented in this section is heavily based on UCAV mission requirements analysis presented in [27].

Multi-UCAV Missions

The missions of interest include those missions currently conducted with multiple manned air vehicles. These include Suppression of Enemy Air Defense, Close Air Support, Air-to-Air Combat, and Logistics Re-Supply. In the first of these missions, a group of aircraft attacks an enemy air defense structure in order to make an area safer for other aircraft and ground troops. The second mission mentioned entails aircraft attacking enemy targets that are in close range to friendly forces. Air-to-Air Combat missions involve engagement with enemy aircraft. Finally, Logistics Re-Supply missions require the delivery of supplies such as ammunition or food into a hostile and unpredictable environment. The focus was not on the particular characteristics of each mission but on their common elements. The models developed as part of the research attempted to capture these common elements as a library of reusable activity models. For the development of these activity models, and also scenarios for testing the planner, the Suppression of Enemy Air Defense (SEAD) mission was used as the prototypical mission.

Scenario Example

Consider a scenario in which friendly ground troops are about to be deployed into hostile territory. Satellite surveillance of the area has revealed two objects that may or may not pose a threat to the friendly forces that are about to be sent out. Rather than risk human lives, it is decided that a group of unmanned air vehicles should be sent to quickly identify whether the threats truly exist, and if so, to destroy them. Two unmanned combat air vehicles, referred to as ONE and TWO, which are located on nearby aircraft carriers, are chosen for this mission.

The vehicles takeoff from their respective positions, and they both fly towards a pre-designated rendezvous location. Vehicle ONE arrives, begins to listen for messages from the second vehicle while it broadcasts a beacon message to indicate that it has arrived, and waits in a holding pattern. A short while later, vehicle TWO arrives, the two vehicles identify one another, and they proceed together to the area where the threats are thought to be located.

There are three corridors available to fly from the rendezvous location to the target area of interest, each with access restricted to a particular time window because other air vehicles in the area are scheduled to use or cross through these corridors. At the time of the rendezvous, all of the corridors are available, but only one of these will remain available until the vehicles can pass safely through, so this one is selected and the vehicles fly together through this corridor.

Finally, they arrive on the border of enemy territory and indicate their arrival to a third friendly party, the forward air controller, who is in charge of dispatching vehicles into the area. The vehicles are authorized to proceed immediately to the target location and attack all targets found within a specified area. The vehicles proceed together to this area, assuming that only one target will be found, but as they approach, they sense the second target. They immediately diverge so that vehicle ONE is continuing towards the first target following a pre-computed attack vector, while vehicle TWO continues towards the second target. They independently bomb their respective targets, check that they have been destroyed, and then exit hostile territory. The vehicles meet again and fly together back towards the original rendezvous point. When they arrive, they separate to return to their respective home ships, and land.

Mission Characteristics

The SEAD mission scenario described in the previous section demonstrates some of the characteristic features of multiple vehicle missions. First, these missions involve coordinated activities, including both activities that are performed together by a group of vehicles, such as flying together towards the target area, and activities that are performed separately but need to be synchronized, such as performing separate bombing runs and then rendezvousing. Second, in these missions, the same activities are performed many times by individual vehicles as well as by the group. For example, a simple activity repeatedly performed by an individual vehicle is flying to a waypoint, and the rendezvous activity, which is performed after takeoff and after the completion of the attack, is an

example of a repeated group activity. Third, the mission activities are hierarchical in that complex activities can be described in terms of simpler ones composed in various ways. For example, flying along a corridor or path is composed of a series of fly to waypoint activities, and performing a bombing run is composed of flying along a path while concurrently targeting and releasing a bomb.

The hierarchical structure of mission activities suggests that it is possible to describe multiple vehicle missions in terms of a set of modular activity models. In other words, since a multiple vehicle mission can be reduced to a set of activities, which can be recursively reduced to a set of primitive activities, this implies that given a set of primitive activity models, one can develop a set of hierarchical activity models, which can ultimately be composed into a multiple vehicle mission model. The fact that the missions involve many recurring activities also implies that this type of representation can be very compact, since in the best case, a mission can be composed of many instances of only a handful of primitive activities.

While using hierarchical activity models addresses the issue of compactness and reusability, the non-trivial problem of developing the activity models remains. It can be a tedious and time-consuming process to construct an activity model because although the corresponding activity may only consist of a few different sub-activities, they may be composed in very complex ways. This makes it very desirable to develop a method for describing these compositions of activities that is intuitive and easy to understand.

The fact that the missions consist of coordinated activities means the planner must not only be able to reason about which activities to perform given several choices, but it must also be able to reason about the implications of these choices on the timing requirements of the mission. For example, the group of vehicles in the previous scenario had to decide which corridor to use when flying to the target area, but these corridors were only available for specified windows of time, and in the scenario above, only one of these was available over the entire time it would take to travel through it. Furthermore, this temporal reasoning must be efficient enough so that onboard re-planning, which is necessary for reacting to unexpected conditions, is not debilitating. This is important because it is impractical to develop a mission plan that can account for all, or even many, of the possible ways in which a mission may unfold.

Problem Statement

The research presented in this thesis concentrates on developing a model-based planning system for coordinated multiple unmanned combat air vehicle (UCAV) missions, to address the challenges of concisely representing domain and vehicle knowledge and efficiently developing plans that are both expressive and flexible. The first contribution of this research is a novel method for developing activity models by extending reactive programming languages to express contingencies and metric time constraints, the second is a compact encoding of the activity models that facilitates efficient planning, and the third is an algorithm that uses these models to generate mission plans.

Activity Modeling Language

There are well-researched, formal languages for describing complex reactive systems [9,25], such as spacecraft, telephone switching networks, and commercial plane avionics systems. These languages describe the system in terms of its states, behaviors, and the effects of the behaviors on the state, and they are currently being explored for applications such as model-based mode-identification, diagnosis, execution, and reactive planning. They also offer a clean underlying semantics in terms of a process algebra. The planning system described in this thesis leverages this past research in formal modeling languages to help address one of the primary challenges of planning, the problem of capturing and encoding knowledge about the complex behaviors and interactions of cooperative agents and their environment.

A modeling language called the Activity Modeling Language (AML) was developed by extracting the useful expressive features of a constraint-based modeling language known as the Reactive Model-based Programming Language (RMPL) [25]. The Activity Modeling Language can be used to describe complex system behaviors using a set of intuitive combinators similar to those of procedural programming languages. In order to support the modeling of coordinated, temporally extended behaviors, this language was extended to support the representation of continuous, metric time.

Temporal Planning Network

Once system models are described using RMPL, they are compiled to a compact representation as a set of hierarchical, concurrent, probabilistic automata (HCA) [25] that encode system behavior. It was necessary to develop an analogous encoding for models described in AML.

The HCA encoding could not be used because it relies on the assumption of synchronous, unit-delay transitions between states, which is insufficient for representing time-critical activities.

Therefore, a different type of model encoding, called the Temporal Planning Network (TPN), was developed by merging some of the features of the HCA models with temporal constraint models known as Simple Temporal Networks (STN) [7]. The STN temporal constraint representation was adopted as the base representation of the encoding because it supports efficient temporal reasoning techniques, which have been well applied by other planners such as HSTS [12] and ASPEN [16]. The STN representation was then augmented with symbolic constraints to support the expression of non-temporal constraints, for example, constraints to represent usage of a shared resource for an interval of time, and decision nodes to model multiple alternatives for performing an activity.

The resulting Temporal Planning Network activity models are able to represent plans involving concurrent, unconditional plans, as generated by discrete event planners like STRIPS [8] and temporal planners like ASPEN [16] and HSTS [12], and can additionally be used to express temporal duration of activities and maintenance conditions. The

Temporal Planning Network activity models extend this representation to encode conditionality and protections.

Model-based Temporal Planner

Our planning algorithm employs a mixture of classical planning techniques and temporal reasoning to efficiently generate mission plans. It takes as input an activity model, for example the SEAD mission model, and identifies a plan using a combination of network search, incremental temporal consistency checking, symbolic conflict discovery and repair, and hierarchical decomposition.

The planning algorithm achieves efficiency through its use of activity models. Because the activity models are hierarchical, this planner benefits by being able to plan hierarchically. For example, when a high-level plan is found to be temporally inconsistent, the planner does not have to consider any plans that could result from the expansion of this plan. Detecting inconsistencies at the higher levels of planning rules out large portions of the space of possible plans from examination, and therefore can significantly reduce the time necessary for planning.

Furthermore, the planner presented in this thesis uses activity models, each of which encodes the possible behaviors of its corresponding activity by describing the set of valid executions. Therefore, while classical partial order planners spend most of their time trying to compose activities to construct a valid execution, this planner simply searches over the pre-generated structures of the activity models to simply identify a valid execution. This technique is similar to Graphplan [2] and SAT-plan [18] in that they also rely on pre-generated structures to simplify and speed up the run-time planning.

Thesis Layout

The next chapter provides background material in the areas of constraint-based modeling, temporal reasoning, and planning and scheduling from which the planner draws conceptually. Chapter 3 presents the Activity Modeling Language used to describe the activity models and the mapping from the activity model descriptions to the Temporal Planning Network representation used by the planner, along with examples of models developed for the UCAV missions. Chapter 4 explains the planning algorithm in detail, and includes examples to illustrate the process. The final chapter presents results that demonstrate the planner's capabilities, a discussion of planner performance, conclusions, and future work.

Chapter 2

Background

The planner presented in this thesis addresses the problem of planning for coordinated air vehicle missions. In order to do this, the planner draws from three distinct research areas. The first of these is modeling reactive systems using constraint-based languages [25]. The second area of research is temporal constraint modeling and reasoning using Simple Temporal Networks [7].

These two areas form the foundation of the Activity Modeling Language and the Temporal Planning Networks introduced in Chapter 3. The third research area is Artificial Intelligence Planning, which is drawn from extensively by the planning algorithm described in Chapter 4 [17,21].

The Reactive Model-based Programming Language [25] is a language originally developed to model the behavior of complex, mixed software and hardware systems consisting of many components. The language is a form of process algebra [4] that provides a set of combinators for facilitating the description of the behaviors of these complex systems. The reason this is relevant to the problem of multiple vehicle mission planning is that the challenge of modeling the behavior of complex reactive systems is the very similar to the challenge of modeling the coordinated activities of vehicles in a multiple vehicle mission. Since the challenge of modeling reactive systems has been addressed by the development of RMPL and similar constraint-based languages, it is logical that a similar language can be used to facilitate the modeling of multiple vehicle mission activities.

RMPL describes complex behaviors as the composition of less complex behaviors, which are implicitly coordinated through system constraints. For example, two activities that assert conflicting constraints are implicitly coordinated in that they are never concurrently executed. One limitation of RMPL is that it assumes the system can be modeled using discrete time steps. This is a necessary assumption for tractability of problems addressed by RMPL models, including mode identification and diagnosis. However, RMPL's notion of time makes it difficult to describe the explicit coordination of activities. For example, RMPL cannot be used to express that one activity should follow another by a certain number of time units.

The representation of temporal constraints is the strength of Simple Temporal Networks (STN) [7]. These networks provide a means of explicitly coordinating activities in terms of their times of execution and the time between different activities. For example, this network representation can be used to express that an activity should execute for exactly 30 seconds, and that the start of a second activity should precede the completion of the first by at least 5 seconds. Furthermore, there are efficient techniques for reasoning about temporal constraints in Simple Temporal Network form. For example, given a set of activities and temporal constraints as an STN, it is possible to determine whether it is possible to perform the activities such that no temporal constraints are violated, or compute the feasible times at which an activity may start.

Reactive Model-based Programming Language

The Reactive Model-based Programming Language is a high-level language used to describe models of reactive systems. The models specify the behaviors of a system in terms of its default behavior and also its possible actions and their effects on the system. For example, consider the RMPL code fragments in Figure 2.1, which are models from the *Deep Space 1* spacecraft. Figure 2.1a specifies the default behavior of the onboard MICAS camera, which functions normally most of the time, but fails with 1% probability at each time step. Figure 2.1b describes the Auto Navigation activity, in which the spacecraft uses the MICAS camera to take several pictures and then uses these to correct its course.

<pre> MICAS :: always { choose { if MICASon then { if TurnMicasOff thennext MICASoff elsenext MICASon, if MICASoff then ..., if MICASfail then ..., } with 0.99, next MICASfail with 0.01 } } </pre>	<pre> AutoNav() :: { TurnMicasOn, if IPSon thennext SwitchIPSSStandBy, do { when IPSstandby^MICASon donext { TakePicture(1); . . . { TurnMicasOff, OpticalNavigation() } } } watching MICASfail^OpticalNavError, when MICASfail donext { fMICASReset, AutoNav() }, when OpticalNavError donext { AutoNavFailed } } </pre>
(a)	(b)

Figure 2.1 RMPL examples. (a) Default behavior of a component. (b) Auto Navigation activity.

These models can be used by the system for execution, as well as to infer current state, diagnose problems, and develop plans to reconfigure itself into a desired state. The benefit of RMPL is that it offers a simple and natural way of expressing complex behaviors.

RMPL supports a set of combinators that provide an intuitive way to describe behaviors of reactive systems. The types of behaviors that it is possible to express using these combinators include conditional execution, iteration, serial and parallel execution, preemption, probabilistic choice, and utility-based choice. These combinators can be combined recursively to describe arbitrarily complex behaviors that can be factored down to the primitive set of expressible behaviors. The list of combinators is provided in Figure 2.2.

```

A := c |
    if c thennext A |
    unless c thennext A |
    A, A' |
    A; A' |
    do A watching c |
    always A |
    choose-probability { A with p, A' with p',
... } |
    choose-reward { A with r, A' with r', ... }
c := constraint
p := probability
r := reward

```

Figure 2.2 RMPL combinators

A modeler can use RMPL to define the behaviors of activities such as TakePicture() and OpticalNavigation(), and then define the behavior of higher-level activities such as AutoNav(), in terms of these other behaviors. Models of simple components, such as valves and tanks, can also be composed to form more complex component models, such as a spacecraft propulsion system.

RMPL describes system behaviors in terms their assertions of constraints. For example, in the example of Figure 2.1a, MICASon, MICASoff, and MICASfail are constraints corresponding to the conditions that the MICAS camera is on, off, or in a failure state, respectively, and TurnMicasOff is a constraint representing the assertion that the MICAS camera should be turned off. The Auto Navigation activity, for example, uses the MICAS camera and to take pictures and then asserts TurnMicasOff to turn it off.

Hierarchical Constraint Automata

Each model described in RMPL is compiled into the Hierarchical Constraint Automata representation consisting of a collection of states and transitions. One slight difference

between HCA and traditional automata is that multiple states may be enabled, or marked, at any time. More significantly, each state of HCA may be labeled with constraints that are asserted whenever the state is marked. Transitions may also be labeled with constraints to indicate a guarded transition, or in other words, a transition that is conditioned on the presence or absence of a constraint assertion. These constraints represent the interactions of the state variables of the system, and therefore provide a basis for system diagnosis and control.

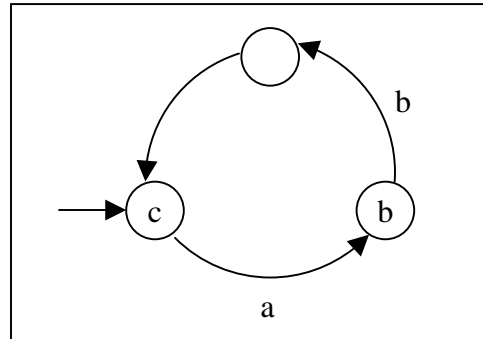


Figure 2.3 Simple Hierarchical Constraint Automaton

Figure 2.3 illustrates an example of a simple HCA consisting of three states and three transitions. The labels in the states represent constraints that are asserted when that state of the automaton is marked. The transition that starts at no state and leads into the left-most state signifies the left-most node to be a start state of this automaton. This state is marked in the time step that the automaton execution is initiated, at which point it asserts the constraint *c*. After this time step, this state is exited and the automaton transitions into the state containing *b*, only if constraint *a* is asserted externally by another automaton.

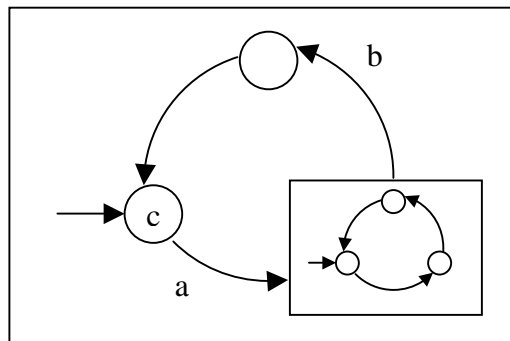


Figure 2.4 Example of an automaton serving as a state

The automata are hierarchical in that a state of an automaton may itself be an automaton, similar to State Charts [10]. This is illustrated in Figure 2.4. The HCA representation can also be viewed as an encoding of a partially observable Markov decision problem

since transitions have an associated probabilities and reward. This allows HCA to model stochastic behaviors and the utility of alternative choices.

The RMPL combinators each have an encoding as an HCA. The mapping from RMPL to HCA, which is provided in [25], defines how RMPL descriptions are compiled into a set of HCA.

Simple Temporal Networks

Temporal constraints are used to specify requirements concerning the times of different events, where an event is defined as something that occurs at a single point in time. For example, *walking to the store* is not an event because it occurs over an interval of time, but *starting the walk* and *completing the walk* are both events because they correspond to instants of time.

A unary temporal constraint restricts the time of an event to be within a specified absolute time range, while a binary temporal constraint restricts the duration between two events to be within a relative time range. For example, in order to express that the walk to the store needs to be completed some time between 8:00am and 8:15am, one would introduce a unary temporal constraint on the *completing the walk* event with that absolute time range, [8:00am, 8:15am]. To express that the walk takes between 30 to 40 minutes, one would introduce a binary temporal constraint between the *starting the walk* and *completing the walk* events with the range [30,40] to indicate that the time between these events should be between 30 and 40 time units, which are minutes in this case.

Temporal Constraint Networks [7] provide a formal framework for representing and reasoning about systems of temporal constraints. There are two classes of problems addressed by this representation, Simple Temporal Problems (STP) and the more general Temporal Constraint Satisfaction Problems (TCSP). Note that these are classes of problems that include specific problems such as checking whether a system of temporal constraints is consistent and computing the feasible time bounds for an event. The difference is that TCSPs allow temporal constraints that specify multiple disjoint ranges whereas STPs represent only a single range per temporal constraint. Although the difference may seem minor, Temporal Constraint Satisfaction Problems have been proven to be NP-hard [7] while Simple Temporal Problems can be solved using a variety of polynomial-time algorithms.

Temporal Constraint Networks that only address Simple Temporal Problems are also known as Simple Temporal Networks (STN). The activity models used by the planner use the STN representation to encode temporal information, and the planner uses STP techniques as part of the planning algorithm. The following sub-sections will discuss relevant information about the Simple Temporal Network representation and solution techniques.

Representation

A Simple Temporal Network consists of nodes and directed arcs with interval labels. Each node i represents an event, and each arc (i, j) between the nodes i and j represents a binary temporal constraint over their corresponding events. The interval label on each arc indicates the single range specified for that temporal constraint. For example, node 1 and node 2 in Figure 2.5 represent the events *start walk to store* and *complete walk to store*, respectively. The directed arc $(1, 2)$ with interval label $[30, 40]$ represents the temporal constraint that the difference between the times of these two events should be between 30 and 40 minutes.

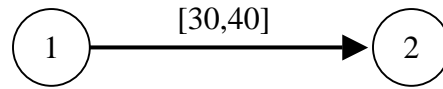


Figure 2.5 STN model of the *walk to store* activity

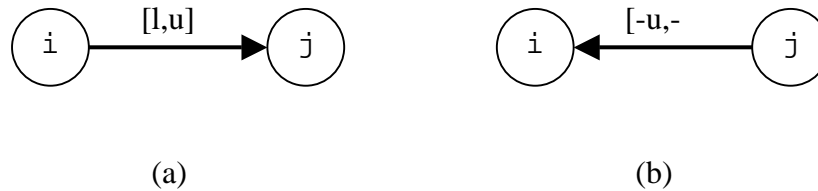


Figure 2.6 (a) Generic temporal constraint between event i and event j , (b) An alternative representation of the same temporal constraint

More generally, a temporal constraint represented by the arc (i, j) with label $[l, u]$ says that the time of event i must precede the time of event j by at least l time units and at most u time units. Note, this temporal constraint could be represented alternatively by the arc (j, i) with the interval label $[-u, -l]$, as in Figure 2.6.

Unary temporal constraints can be represented in one of two ways. The first way is for the node corresponding to the event to be labeled with the absolute time range as in Figure 2.7a. Absolute time ranges are enclosed in angled brackets instead of square brackets to clarify the difference between them. The second way for a unary temporal constraint to be represented is as a binary temporal constraint between the node corresponding to the event and an artificial node representing a fixed time point. This is illustrated in Figure 2.7b, where an artificial node is added to represent the absolute time 8:00am. For any STN, there only needs to be one of these temporally anchored nodes against which any other event's time may be referenced. This anchored node is referred to as the origin node.

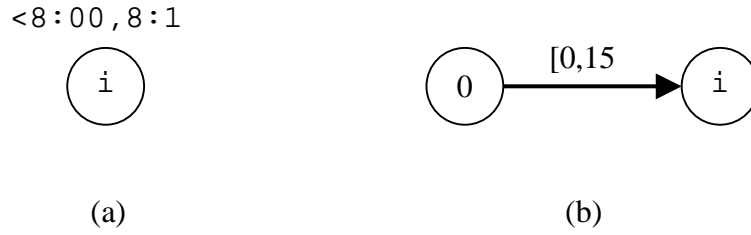


Figure 2.7 Event i can be constrained to occur between 8:00 and 8:15 using a unary constraint (a), or a binary constraint (b) where event 0 is anchored at 8:00

Finally, just as the *walk to the store* activity was represented in Figure 2.5, any activity occurring over an extended period can be represented by its start event, end event, and the duration constraint.

Distance Graph Analog

This section provides an overview of technical material much more thoroughly explained in “Temporal Constraint Networks” by Dechter, Meiri, and Pearl [7]. The reader is referred to this paper for a more formal treatment of the concepts presented here.

Another way of representing a system of temporal constraints modeled as an STN is by using an equivalent distance graph. In the distance graph, the nodes still correspond to temporal events, but the arcs are used slightly differently. Instead of having interval labels, each directed arc holds a distance label. If there is an arc (i, j) with a distance label d , this can be interpreted as restricting the time of event j to be at most d time units greater than the time of event i . The distance labels are not restricted to be non-negative.

Applying this interpretation of the distance graph, the binary temporal constraint represented by an STN arc (i, j) with label $[l, u]$ could be rewritten using a pair of arcs in distance graph form. In the distance graph, there would be an arc (i, j) with distance label u , representing the upper-bound constraint, and there would be an arc (j, i) with distance label l , representing the lower-bound constraint. This is illustrated in Figure 4.8. This is also consistent with the alternative STN representation of the same temporal constraint using the opposite arc (j, i) . In that case, the interval label on (j, i) would be $[-u, -l]$, which would translate to the distance graph form as an arc (j, i) with distance $-l$ and arc (i, j) with distance u .

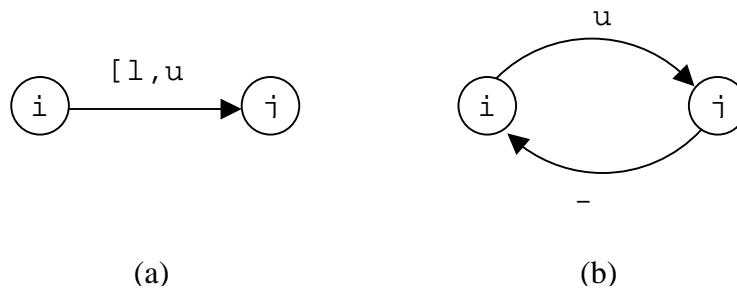


Figure 2.8 (a) 2-node STN, and (b) its corresponding distance graph

The distance graph representation leads to efficient techniques for solving a variety of questions or problems that one might have given a set of events and a system of temporal constraints over these events. For example, one problem is determining whether a system of temporal constraints is consistent. Another way to phrase this is, does there exist times that can be assigned to each event such that all temporal constraints are satisfied? A second problem is to find the possible times at which an event can occur while not violating any constraints. Solving both of these problems are critical to the planning algorithm described in Chapter 4. The planner repeatedly solves this first problem to quickly detect whether the plan is invalid, and it solves the second problem in order to identify and resolve symbolic constraint inconsistencies.

Both of these problems can be solved using common network-based algorithms.

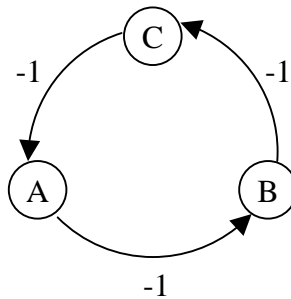


Figure 2.9 Example of a negative cycle

Determining whether a system of temporal constraints is consistent can be done by checking for negative cycles in the distance graph representation. If a negative cycle exists, then the system of constraints is inconsistent [7]. To illustrate this, consider the impossible situation that event A is exactly one time unit before event B, event B is exactly one time unit before event C, and event C is exactly one time unit before event A. The distance graph representation of these events and temporal constraints contains a negative cycle as shown in Figure 2.9.

Consider the problem of determining the upper-bound time difference from node i to node j . Even if there is a temporal constraint between them, this may not be the tightest constraint on the upper-bound time difference. For example, consider the situation in Figure 2.10, in which the time of event j is constrained to be at most 8 time units after the time of event i . However, the constraints between node i and node k , and between node k and node j , imply a tighter constraint on the upper-bound time difference between events i and j . Now consider the problem of computing the lower-bound time difference from node i to node j . Again, though there may exist an explicit lower-bound constraint as there is in the example of Figure 2.10, this may not be the tightest lower-bound constraint. The actual lower-bound constraint is given by the implied constraint through event k . This example gives some intuition into why the lower- and upper-bound time differences between two events can be computed by solving two single-source shortest path problems, whose correctness has been proven formally by Dechter, Meiri, and Pearl [7].

The range of feasible times for a given event can be determined by computing the lower- and upper-bounds on the temporal distance between the origin node, which is anchored to

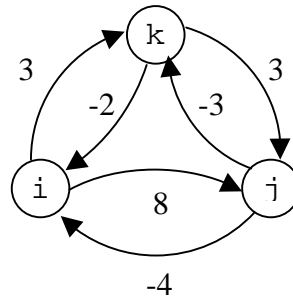


Figure 2.10 Example of how indirect distance bounds between two nodes may be tighter than direct bounds: $\text{distance}(i,k) + \text{distance}(k,j) < \text{distance}(i,j)$

an absolute time, and the node corresponding to the event. This means it is also possible to compute the feasible time bounds for all events by solving an all-pairs shortest path problem for this distance graph.

Planning Overview

The traditional planning problem consists of an agent that must decide which activities to perform to transition from an initial state to the goal state. The agent is the virtual or embodied entity that performs the actions, which effect the state of the agent's universe, including the agent itself. In order for the agent to make decisions about which activities to perform, it must know at least what activities are available and their restrictions and effects. This information is often distilled into simple activity models described in section 2.3.1. Given this knowledge, the agent can employ a variety of planning methods to either generate a plan from scratch or repair an incomplete plan, as described in sections 2.3.2 through 2.3.4. Section 2.3.5 shows some ways in which classical planning methods have been extended into newer temporal planning techniques.

STRIPS Activity Models

STRIPS operators [8] are commonly used to represent an agent's available actions or activities. Each STRIPS operator models an activity in terms of a set of pre-conditions and post-conditions. The pre-conditions define conditions that must be true in order for the activity to be used, and the post-conditions represent the activity's effects by defining the set of conditions that are true after the activity completes. These conditions are typically represented as a conjunction of literals, where each literal is simply a symbol corresponding to a condition that may be either true or false.

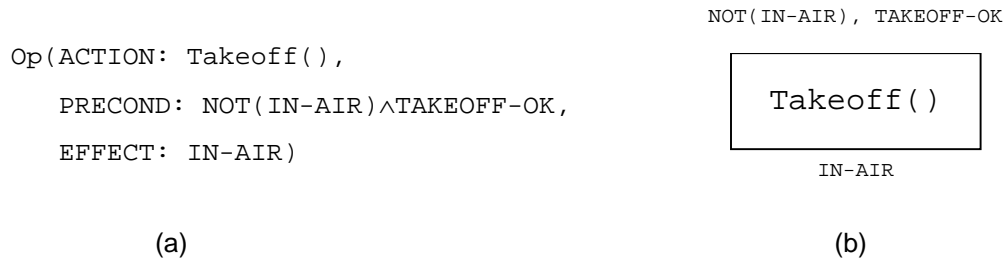


Figure 2.11 The STRIPS operator activity model for the Takeoff activity expressed (a) textually and (b) graphically, in the formats used by Russell and Norvig [17]

For example, assume the agent is an air vehicle. Then the condition that the vehicle is in the air might be represented by the literal IN-AIR, and the condition that the runway is clear for Takeoff might be represented by TAKEOFF-OK. The agent might have the ability to Takeoff, which could be represented by a STRIPS operator with the pre-conditions $\text{NOT}(\text{IN-AIR}) \wedge \text{TAKEOFF-OK}$ and the post-condition IN-AIR. This would mean that the Takeoff activity could only be used if the vehicle was not in the air and if the runway was clear, and after the Takeoff activity completed the vehicle would be in the air.



Figure 2.12 Example of Start and Goal states represented as activity models

The start and goal states can also be modeled as STRIPS operator activity models. The start state activity model has no pre-conditions, but has as post-conditions those conditions that are true initially. The goal state activity model has no post-conditions but represents the conditions of the goal state as pre-conditions. Figure 2.12 shows how the start and goal states can be represented by activity models.

Partial Order Planning

Given a set of STRIPS activity models, one way to plan is by constructing a path through the state-space connecting the start state with the goal state. From the initial state of the agent's universe, the planner can identify those activities that are enabled, meaning that their pre-conditions are satisfied by the conditions that are true in the initial state. Then the planner can add one or more of these activities to the plan, which has the effect of transitioning the agent's universe into a new state in which different conditions are true. From this new state, the planner can again check which activities are enabled from the current state, and select more activities to add to the plan, continuing until the conditions of the current state satisfy the conditions of the goal state.

When the planner adds an activity, it must also introduce dependency links, often called causal links, to represent that the activity depends on other activities or the initial state to satisfy its preconditions. For example, in Figure 2.13, the activity Takeoff has the pre-conditions $\text{NOT}(\text{IN-AIR}) \wedge \text{TAKEOFF-OK}$, which are both asserted by the initial state, so a causal link is added from the initial state to the Takeoff activity for each of these conditions. The Fly-to(X) activity has the pre-condition IN-AIR which is satisfied by the Takeoff activity, so a causal link is added from the Takeoff activity to the Fly-to(X) activity. This causal link indicates that Takeoff must precede Fly-to(X).

An alternative way to construct a plan is to start from the goal state work back towards the start state. The planner selects one or more activities whose post-conditions include the goal conditions and introducing a dependency from the goal state to each of these activities. However, in order for these activities to be used, their pre-conditions must be satisfied, so the pre-conditions of these added activities are added to the set of goal

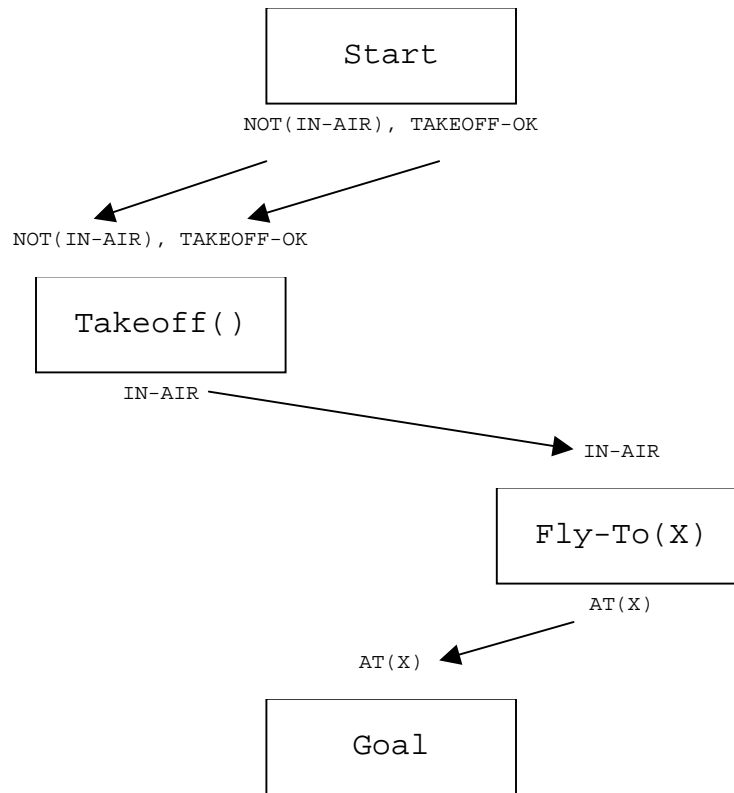


Figure 2.13 Example of a plan generated through Partial Order Planning

conditions that must be satisfied. The process is complete when all of the remaining goal conditions are satisfied by the start conditions. This type of planning is known as regression planning because it starts at the goal state and works back to the start state, whereas the first approach described is called progression planning, because it builds forward from the start state towards the goal state.

A plan produced by either progression or regression planning consists of an ordering of activities to be performed to get from the start state to the goal state. These forms of planning produce a non-linear plan, consisting of activities and dependencies from activities to other activities that form a partial ordering. Each activity has a dependency on zero or more other activities, and it cannot be performed until every activity upon which it has a dependency has been performed. If the activities were totally ordered then the plan would be linear, consisting simply of a sequence of activities to be performed one after another. However, in many cases a total ordering is not necessary or

undesirable, because the planner must make ordering decisions even between activities that have no dependence on one another.

Hierarchical Planning

One problem with the planning technique described in the previous section is that it is slow in practice, especially with a large number of available activities. A way to speed up planning is to compose the primitive activities into a set of higher-level activities or macro-activities, introducing a hierarchy to the activities. Each macro-activity is essentially a partially ordered plan consisting of primitive activities, which can be composed into higher-level macro-activities that describe behaviors that are even more complex. The planner can first construct a plan using the highest-level macro-activities, and then decompose each macro-activity until the plan includes only primitive activities.

However, there is some additional work to be performed beyond decomposition, because decomposition may reveal a potential conflict between lower-level activities. A conflict occurs if two activities prevent one another from performing correctly, due to conflicting post-conditions or pre-conditions. Conflicts are resolved by ordering activities such that their conditions no longer interfere with one another. A hierarchical planner first generates a plan using macro-activities, decomposes these macro-activities into primitive activities, and then repairs the plan to resolve any inconsistencies.

There are two desirable properties of hierarchical activity models. First, any valid high-level plan should be decomposable to a primitive-level plan that is still valid, and second, for any valid primitive-level plan there is a corresponding highest-level plan. The first is called the “downward solution” property and the second is called the “upward solution” property [17]. If one or both of these properties hold, then hierarchical planning provides a significant performance benefit over planning with the primitive activities because it prunes the search space.

State-space and Plan-space Planning

There are two general methods of planning, state-space planning and plan-space planning. The partial order planning techniques described in section 2.3.2 fall into the realm of state-space planning, while hierarchical planning described in the previous section employs both state-space and plan-space planning methods. The input to a typical state-space planning problem is a start state, a goal state, and a set of available activities, and the state-space planner incrementally adds activities to the plan until a path is found that can transition the agent from the start state to the goal state. The reason this is called state-space planning is that the planner is searching over the space of possible states, moving from state to state, as it is adding activities to the plan.

The input to a typical plan-space planning problem is an incomplete or inconsistent plan, and the plan-space planner iteratively adds activities or orderings and repairs the plan until it is both complete and consistent. For example, the generic hierarchical planner described in the previous section took a high-level incomplete plan and repaired it by decomposing macro-activities and adding orderings to resolve conflicts between activities. At each iteration, rather than deciding which activities to add to bridge the gap

between the start state and the goal state, the planner decides how to modify the current plan to move towards completeness and consistency. It makes decisions over the set of possible plans, while the state-space planner makes decisions over the set of possible states.

Temporal Planning

The technique of partial order planning produces a plan that is an ordering of activities, but does not care about the duration of activities or the time between activities. Many planners similarly rely on an abstract notion of time. The justification for these methods is the assumption that timing issues can be addressed as a separate scheduling problem; a plan is first constructed without worrying about how long activities take to perform, and then the activities are scheduled, or assigned times, such that the activity dependencies are satisfied. The problem with this is that even though a plan might be constructed relatively quickly, if the planner does not reason about temporal constraints, there may be no feasible way to schedule the activities to fulfill the timing requirements of the mission. Recent work in temporal planning has attempted to tackle exactly this problem by blurring this separation of planning and scheduling, and incorporating temporal reasoning into the planning process. A good overview of research in this area is provided by [18].

One general technique for bringing together the problems of planning and scheduling is by casting them as resource-constrained project scheduling problems [18]. This type of problem assumes that there are resources that are consumed in some quantity by each of the activities, and that activities have a fixed, pre-determined duration. It takes as input an ordering of activities, as would be produced by a partial order planner, and assigns start times to each activity to times such that all the ordering constraints are satisfied and no resource is over-consumed. Well-known constraint satisfaction techniques, such as backtracking search or forward checking [17], can be used to find feasible assignments. However, because of the number of possible times to assign to each activity depends on the resolution of time, this problem requires a tradeoff between tractability and precision; with very fine time resolution the space of possible solutions becomes enormous, while using a more coarse resolution reduces this space but exaggerates the discretization of time.

There has also been work with continuous planning, which addresses the issue of both continuous time and resource consumption. The ZENO planner [15] uses more complex activity models that represent the conditions and effects of each activity using a set of metric constraints. For example, using the ZENO representation, one could model that the consumption rate of fuel was equal to a specified value for the duration of the Fly-to(X) activity. These models also allow inequality constraints, for example, that the quantity of fuel must be greater than zero during flight. The activity's temporal duration can also be represented as a metric constraint between the start time and end time of the activity. ZENO generates plans using the regression planning technique, starting with a set of goal conditions and introducing new activities and dependencies until there are no goals left to be satisfied. Although very powerful, ZENO is admittedly practical for only toy problems because it is relatively slow.

The HSTS planning and scheduling system [13] takes a novel approach to the planning problem. Instead of modeling the state of the planning subject and its environment as a set of conditions, it maintains an explicit set of variables each represented by a timeline over a finite time horizon. Each timeline holds tokens, covering some portion of the timeline, which indicate the state of the variable or the invocation of an activity acting on the variable state over the duration corresponding to the timeline segment covered by the token. For each type of token, there are associated constraints called compatibilities, which must be satisfied in order for the plan to be complete. A compatibility might require the addition of tokens or it might impose a temporal constraints between itself and another token. A planning goal state is represented by an incomplete plan consisting of a partially populated set of timelines representing the desired state of the variables at particular times. The planner is then responsible for resolving compatibility conflicts by adding tokens and temporal constraints between tokens, or shifting tokens to enforce temporal constraints. The planner also uses a Temporal Constraint Network data structure, also known as a Simple Temporal Network [12,7], to represent and perform temporal reasoning over a system of temporal constraints.

While planners such as ZENO and HSTS address the issue of temporal planning, they do not adequately address the problem of planning for coordinated air vehicle missions for two reasons. The first is that they do not provide a natural way to develop activity models used by the planner, which makes it very difficult to model activities involving complex coordinated behaviors of multiple vehicles. The second problem is that these planners require much more time to generate plans than is allowable in the context of an unmanned combat air vehicle mission. It is necessary for the planner to generate a plan in a few seconds rather than a few hours [3]. These issues are the primary focus of the remainder of this thesis.

Chapter 3

Activity Models

Overview

In order for any planner to function, it needs knowledge about the abilities of the agents for which it is planning. This information is conveyed to the planner through activity models. As mentioned in chapter two, STRIPS operators provide a simple and compact means for a modeler to describe an activity in terms of what conditions must be true of the environment and the subject in order for the activity to be used, and also the effects of the activity. However, in using STRIPS operator models, the modeler abstracts away a lot of information about activities.

For the reasons stated in chapter one, the planning of multiple vehicle missions requires activity models that are rich enough to represent the requirements of coordinated behaviors, making STRIPS operators insufficient. For example, the activity model of a rendezvous activity must be able to represent the requirement that all vehicles must meet somewhere at the same time. An example of another type of coordination requirement is that multiple vehicles of a group should not transmit messages on a single communication channel at the same time. Therefore, the activity models used for multiple vehicle missions must be able to express both timing requirements and resource constraint requirements. Given these requirements on the expressive power of the requisite activity models, it is also important to be able to keep the activity models as simple, compact, and easily encodable as possible.

The RMPL models of reactive systems described in chapter two offers two things to this end. First, RMPL provides an expressive but simple process algebra that makes it easy to describe activities and the composition of activities. Second, the HCA representation, into which the RMPL activity descriptions can be compiled, provides a compact encoding of concurrent behaviors in terms of the constraints they impose on the system, which can be used to model the resource constraint requirements of coordinated vehicle activities. However, these models are deficient in their ability to express the timing requirements of coordinated activities.

The STN representation provides a way of representing complex systems of temporal constraints, so this representation can be used to model the timing requirements of coordinated mission activities. However, the STN representation was not designed to represent the resource constraint requirements necessary for modeling a coordinated activity. In addition, it is tedious to construct an STN model of the temporal constraints of an activity, especially if the activity involves complex composition of many primitive activities.

This chapter presents the result of blending these two representations to form the Activity Modeling Language, which extends RMPL to allow for the description of temporal

constraints, and the Temporal Planning Network, which unifies the expressive power of HCA and STN for representing activity models. Because the Activity Modeling Language and the Temporal Planning Network activity models are based on the behavior models described by programming languages such as CC [9] and derivatives, and RMPL [25], it is possible to leverage off their past work and experience in the formal modeling of reactive systems.

These and related languages have been used in the past to develop models used for simulation, system mode identification, diagnosis, and execution in real-world situations such as onboard the Deep Space One spacecraft [14]. As described in the previous chapter, RMPL provides a set of combinators that allow for the description of system behaviors modeled as concurrent hierarchical constraint automata. These combinators can be used to express the assertion of constraints, parallel and serial composition, conditional execution, iteration, preemption, probabilistic transition, and utility-based transition. The Activity Modeling Language borrows from RMPL the combinators useful for describing activity models for planning, and augments this subset with a means for representing temporal relations, including the duration of activities and the time between activities.

Just as RMPL is used to describe the state and behavior of complex systems in terms of the composition of the states and behaviors of their components, AML can be used to describe complex coordinated activities in terms of the composition of component activities. Hierarchical modeling not only helps to minimize the size and complexity of the models, but also aids in making the planner more efficient. These models express alternate ways of performing activities explicitly as part of the activity models, rather than leaving these decisions to be inferred by the planner as in the case of classical partial order planners. As mentioned in chapter one, this is necessary for planning UCAV missions because these types of choices are typically tightly controlled. One resulting benefit of this type of representation is that the modeler can easily control the complexity of the planning problem; introducing many decisions allows for more variation and correspondingly more complexity, while incorporating few decisions means the mission will always be performed basically in the same way, but missions can be developed more quickly. Note that even if there are no explicit decisions in the activity models, there are still many decisions regarding activity start times and durations. While this kind of mission model is rigid in the sense that it is always performed using the same activities, there may remain a great deal of temporal flexibility in how the mission is performed given different requirements.

Example

Consider the SEAD mission scenario described in chapter one. One of the activities performed during the mission was the *Enroute* activity, in which the group of vehicles flew together from the rendezvous point to the target area. In this activity, the group selects one of two paths for traveling to the target area of interest, flies together along the path through a series of waypoints to the target position, and then transmits a message to the forward air controller to indicate their arrival, while waiting until the group receives authorization to engage the target.

The two paths available for travel to the target area are each only available for a predetermined window of time, which is important for the planner to consider when selecting one of these paths. In addition, the Enroute activity will be bound in time by externally imposed requirements, for example, the mission must complete in 25-30 minutes, while at least 20% of this time and at most 30% of this time is allotted to the Enroute activity. Therefore, it is also useful for the activity to pass these time constraints down to its constituent sub-activities. For example, 90% of the time available for the Enroute activity should be allotted to the sub-activity of flying along the selected path since that will probably require the most time of all sub-activities. The desired behavior of this activity is captured in the AML code below.

```

Group-Enroute()[1,u] = {
  choose {
    do {
      Group-Fly-Path(PATH1_1,PATH1_2,PATH1_3,TAI_POS)[1*90%,u*90%];
    } maintaining PATH1_OK,
    do {
      Group-Fly-Path(PATH2_1,PATH2_2,PATH2_3,TAI_POS)[1*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 3.1 AML description of the Enroute activity of a multiple vehicle SEAD mission

Once the behavior of an activity is described in AML, this description can be compiled into a Temporal Planning Network model. The Temporal Planning Network corresponding to this Enroute activity is graphically depicted in Figure 3.2. Activity name labels are omitted to keep the figure clear, but the node pairs 4,5 and 6,7 represent the two Group-Fly-Path activity, and node pairs 9,10 and 11,12 correspond to the Group-Wait and Group-Transmit activities, respectively. Node 3 is a decision node that represents that there is a choice between the two methods for flying to the target area.

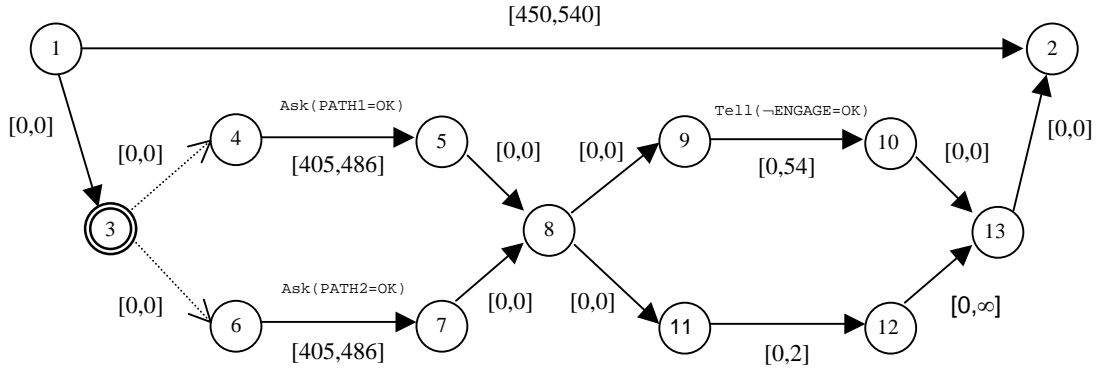


Figure 3.2 A possible instantiation of the Temporal Planning Network activity model for the Enroute activity

This is the Temporal Planning Network that would result if the mission were allowed to take between 25 and 30 minutes, and the Enroute activity were allowed 30% of this time. This network models the temporal constraints on the overall activity, as well as its sub-activities. The activity model constrains the time for the Enroute activity to be at least 450 seconds and at most 540 seconds, it constrains the time for flying to the target area to at least 405 seconds and at most 486 time units (90% of the allotted time for the Enroute activity), and so on. It also models the decision between the two paths to the target area, and it models the restrictions that each of the paths can only be used if they are available.

Section 3.3 will describe the combinators of the Activity Modeling Language, and justify why they are necessary and useful. The following section will describe in detail the Temporal Planning Network representation and the meaning of its constructs in terms of how it describes execution, and how they can encode activity models described in AML. The Enroute activity description and model in Figures 3.1 and 3.2 will be used as a running example through this chapter.

Activity Modeling Language

One reason for using any modeling language, including AML, is to add a layer of abstraction between the description of the behavior and its actual encoding, especially if the encoding can become incomprehensible. This is certainly the case with the TPN encoding of activity models, which can be both tedious to directly encode and difficult to understand with complex activities. There is no benefit of using a language if it is as difficult to describe the activities using the language as it is to encode the models directly, but at the same time, the language must be sufficiently expressive to be able to describe some minimal set of desired behaviors. For AML, the sufficiency of its expressiveness was based on its ability to describe the necessary coordinated behaviors of multiple vehicle missions. The basic combinators of the AML language are listed in Figure 3.3.

```

A ::= A[l,u] |
      activityinstance |
      c |

```

```

[1,u] |
if c then A |
do A watching c |
A; A' |
A, A' |
{ A } |
choose { A, A', ... }

activityinstance := activityname ( argumentlist )
c := proposition | Not(proposition)

```

Figure 3.3 Basic AML combinators

The Activity Modeling Language was modeled after the Reactive Model-based Programming Language [25] presented in section 2.1. The main differences are that AML excludes the utility-based choice combinators of RMPL, replaces the probabilistic choice combinator (choose-reward) with non-probabilistic, non-deterministic choice (choose) used to express alternate ways for performing an activity, and AML augments this set with the interval construct used for the expression of temporal constraint.

Basic Combinators

In order to express timing requirements of coordinated activities, it is necessary to express temporal duration, for example, that the Enroute activity takes 10 minutes. To accommodate the goal of flexibility it is also useful to simply bound the feasible execution time of an activity and let the planner determine the appropriate amount of time in which the activity should be performed. Therefore, instead of saying that the Enroute activity must take 10 minutes, it might be better to say the Enroute activity must take between 9 minutes and 11 minutes. In AML, this would be expressed as `Enroute() [540, 660]`, with the name of the activity instance followed by the allowed duration range specified by the lower and upper bound pair enclosed in square brackets. By default, if an activity is not labeled with a duration bound, it is assumed that the activity may have any non-negative duration.

In order to support the modeling of coordinated behavior, it is also necessary to describe the requirement and assertions of conditions. This is necessary to make sure that the activities of the vehicles are consistent with one another and consistent with the conditions of the environment. For example, the portion of the Enroute activity in which the group is flying along one of the paths to the target area requires that the path on which they are traveling is available. This requirement needs to be described so that the execution behavior of the vehicles is consistent with the availability of the paths. In order to assert the condition that the first path is available for 5 minutes, one can use the AML expression `path1=ok[300, 300]`. The constraint `path1=ok` represents the condition that the first path is available, and it is qualified by the duration range `[300, 300]` with the time units being seconds.

A condition may be required as a pre-condition or as a maintenance condition of an activity. A precondition asks that a condition be true for the instant before an activity begins. For example, the single vehicle $\text{Bomb}(x, y)$ activity has the precondition that the vehicle has a bomb to use. This precondition is represented in AML using the expression `if c then A[l,u]`, where c represents the condition that is required, and A represents the activity whose execution is conditioned on constraint c . The $\text{Bomb}(x, y)[l, u]$ activity can then be described in AML as `if bomb=ok then Drop-Bomb(x, y)[l, u]`, where bomb=ok represents the condition that the vehicle has a bomb, $\text{Drop-Bomb}(x, y)$ is the activity of dropping the bomb, and $[l, u]$ describes the duration bounds of the bombing activity.

```
Bomb(x,y)[l,u] := { if bomb=ok then Drop-Bomb(x,y)[l,u] }
```

Figure 3.4 AML definition of the Bomb activity

A maintenance condition asks that a condition be true over the duration of an activity. For example, consider again the portion of the Enroute activity in which the group flies along either path one or path two to the target area. If the group flies along path one, then this activity requires that over the duration of this flight, the condition that path one is available is maintained. These maintenance requirements are expressed in AML as `do A[l,u] maintaining c`. This says that the condition represented by c must be true over the duration of the activity represented by A . Figure 3.5 shows the portion of the Engage activity description from Figure 3.1 that corresponds to the activity of flying along path one.

```
do {
    Group-Fly-Path(PATH1_1,PATH1_2,PATH1_3,TAI_POS)[l*90%,u*90%];
} maintaining PATH1=OK
```

Figure 3.5 An example of the AML combinator for expressing maintenance conditions

When describing complex activities, it is useful to describe them in terms of the composition of simpler activities. For example, the activity of flying along a path can be described as the composition of several fly to waypoint activities. AML describes sequential and parallel composition in the same way as RMPL, using two types of delimiters. Semicolon are used to delimit activities to be executed in series, so the AML expression in Figure 3.6a describes the behavior that the vehicle should fly to a series of waypoints, one immediately after another.

- (a) `Fly-To(wpt1)[10,12]; Fly-To(wpt2)[12,12]; Fly-To(wpt3)[8,10]`
- (b) `Fly-To(wpt1)[5,20], Bomb(x,y)[3,8]`

Figure 3.6 An example of the AML (a) sequential composition combinator, and (b) the parallel composition combinator

If activities are separated by commas, then this describes the behavior that they are performed in parallel. For example, `Fly-To(wpt1)[5,20], Bomb(x,y)[3,8]` describes the behavior that the vehicle should fly to a waypoint while bombing a location. To be more precise, this describes the behavior that these activities start together and end together.

```
Fly-To(wpt1)[8,10]; { Fly-To(wpt2)[5,10], Bomb(x,y)[3,6]
}
```

Figure 3.7 An example of activity grouping in AML

Since neither delimiter is given precedence over the other, this might lead to ambiguous compositions. For example, consider the AML fragment `A[1,2], B[5,6]; C[2,3], D[3,3]`. This could be describing that activity A and B should be in parallel, followed by C and D in parallel, or it could be saying that B and C are in series, and that they are execute parallel to both A and D. In order to address this problem of ambiguity, brackets can be used to clearly group sections of an AML activity description, and each grouped section is treated as a sub-activity. Figure 3.7 shows an example of how this grouping may be used. This code fragment describes the behavior of a vehicle flying to a waypoint, then bombing a location while flying to a second waypoint.

```
Transmit(ONE,ALL,STATUS)[1,1]; [58,62]; Transmit(ONE,ALL,STATUS)[1,1]
```

Figure 3.8 Example of a temporal spacer

In order to support the description of more complex coordination, it is useful to be able to express arbitrary temporal constraints between activities. One example of where this is necessary is in expressing a delay between the activities executed in sequence; for example, if a vehicle is supposed to periodically broadcast status messages, it is necessary to describe the time between these broadcasts. Figure 3.8 shows a description of this behavior with broadcasts approximately every minute. The end of the first transmission is constrained to be at least 58 seconds and at most 62 seconds before the start of the second transmission activity.

To express that there are several ways of performing an activity, AML incorporates a choice combinator. This is also the key combinator for expressing contingent executions of an activity. It can be used, for example, in the Enroute activity to represent that there

are two paths available for flying to the target area. It is also used in the Transmit activity to represent the choice between multiple communication channels, as shown below in Figure 3.9. The choices are not associated with reward or probability as in RMPL, but instead choices are considered by the planner in the order they are listed. Therefore, for the Transmit activity, `ch1` represents the default communication channel and `ch2` represents the other available channel that is used if the first is unavailable.

```
Transmit(from,to,msg)[l,u] := {
    choose {
        {ch1=from}[l,u],
        {ch2=from}[l,u]
    }
}
```

Figure 3.9 AML description of the Transmit activity

Derived combinators

The basic combinators are the minimal set needed to describe the coordinated behaviors for multiple vehicle missions. However, a few more combinators can be derived from these that are useful for keeping AML descriptions easy to understand. These derived combinators are listed in Figure 3.10.

```
A := do A[l,u] watching c |
    repeat A[l,u]
```

Figure 3.10 Derived AML combinators

The `do A[l,u] watching c` combinator is similar to the `do A[l,u] maintaining c` except that it describes the behavior that activity `A` is executed until condition `c` becomes true. While the analogous combinator in RMPL was used to express that an activity should be preempted when some condition became true, the AML version of this combinator simply expresses that the interval over which activity `A` executes must not overlap with any interval over which condition `c` is true. The planner is responsible for ensuring that this condition is satisfied. If `c` corresponds to an exogenous condition, there is no way for the planner to guarantee that the unconditional plan it generates will necessarily be consistent through execution. This requires an instance of contingent planning that should be addressed in future work.

The `repeat A[l,u]` combinator is used to model the repeated execution of activity `A` without specifying the number of executions. During the compilation from AML to TPN, the compiler is responsible for determining the minimum and maximum number of times activity `A` may be performed, given `A`'s duration bounds and the duration bounds on the `repeat A[l,u]`. The compiler encodes this combinator with a decision node and the

appropriate number of choices. For example, if the AML expression {repeat A[10,12]}[20,40] implies that activity A must be performed between one and four times, so this would be encoded into a TPN model by a decision node with four choices, one corresponding to a possible number of executions of A. This defers the decision of how many times to perform A to the planner. This is useful in combination with the preemption combinator to express the repeated execution of an activity until some condition becomes true.

For example, the AML fragment `do { repeat Wait(wpt1,wpt2,wpt3)[l,u] } watching group_arrived` describes an activity in which a vehicle flies in a loop through three waypoints repeatedly until the other vehicles in the group have arrived. One limitation of this combinator is that it will not accept a zero duration activity, $A[0,0]$, as the activity to be repeated. The reason, of course, is that an infinite number of these activities may be performed in any instant, which cannot be modeled using the Temporal Planning Network representation. This should still be sufficient, however, because in reality most activities that need to be modeled have positive duration.

Temporal Planning Network

The Temporal Planning Network (TPN) serves as the representation of the activity models used by the planner described in this thesis. A TPN activity model encodes the behavior of an activity by defining the set of feasible executions. TPN models don't enumerate these executions, but instead constrain the set of valid executions by specifying both the temporal and symbolic constraints of the activity. A temporal constraint restricts the behavior of an activity by bounding the duration of an activity, time between activities, or more generally the temporal distance between two events. A symbolic constraint restricts the behavior of an activity by expressing the assertion or requirement of certain conditions by activities. Both types of constraints must be satisfied in any valid execution of an activity.

For example, consider some of the possible executions of the Enroute activity whose TPN model is shown in Figure 3.2. One possible execution is that the group flies along path one to the target area in 420 time units (seconds in this case), transmits an arrival message to the forward air controller in one second, then waits for another 40 seconds to receive authorization to proceed. Another possible execution is that the group selects the second path, flies to the target area in 500 seconds, takes 2 seconds to transmit the arrival message, and is authorized to proceed immediately. If it were the case that path one was available from the time at which the Enroute activity started to at least the time that the group arrived at the target area, then the first execution is valid. This is because it satisfies both the temporal constraints on the Enroute activity, and the requirement that path one is available for the duration of the flight along it. The planning algorithm described in chapter four performs the identification of consistent activity executions.

Each execution corresponds to a set of trajectories or paths through the TPN encoding of an activity. In this sense, the TPN encoding can be understood to be an expansion of the set of possible executions. This encoding is one of the reasons why the planner described in this thesis performs so quickly, but there is a slight downside to this. The TPN

representation is not as compact as the HCA, because it cannot encode iterative behavior using a loop as illustrated in Figure 3.11a. The TPN representation would model this behavior with a non-deterministic choice on the number of iterations to perform, as in

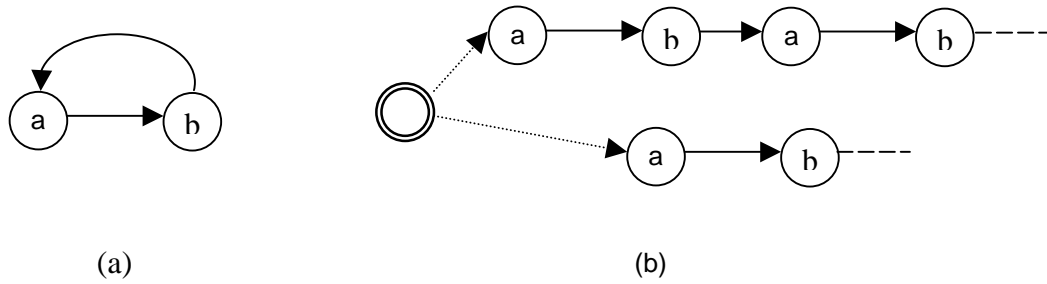


Figure 3.11 Iterative behavior encoded in (a) HCA form, and in (b) TPN form

Figure 3.11b. In effect, the TPN encoding expands out the loop in time, which consumes more space, but saves the planner from performing this expansion at the time of planning. The reason that the activity encoding needs to be expanded in this way is that otherwise the planner cannot directly apply STN methods for temporal reasoning.

A Temporal Planning Network is essentially a Simple Temporal Network that incorporates some features of Hierarchical Constraint Automata, in particular simple symbolic constraints and decision nodes, that make the representation sufficiently expressive for modeling coordinated activities. Just as in Simple Temporal Networks, the nodes represent temporal events, and the arcs represent temporal relations that constrain the temporal distance between events. Figure 3.12 gives an example of a Simple Temporal Network and a Temporal Planning Network.

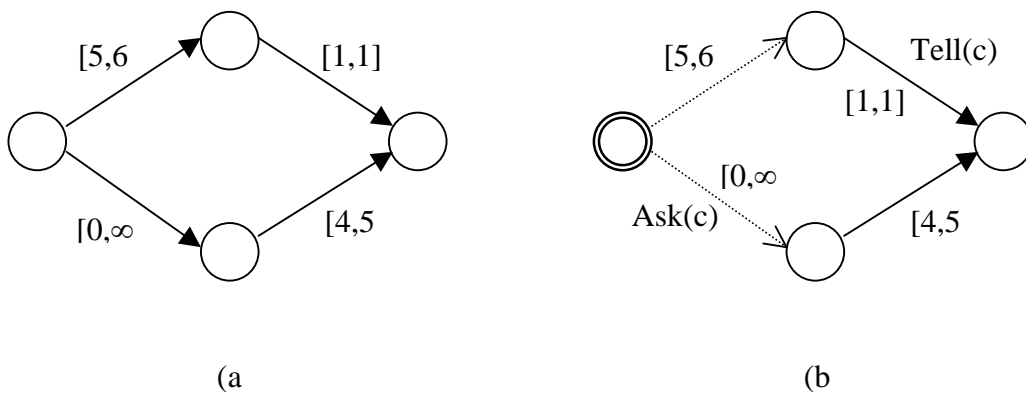


Figure 3.12 An example of a (a) Simple Temporal Network, and (b) a Temporal Planning Network with a decision node and symbolic constraints

The first difference between the Simple Temporal Network and the Temporal Planning Network is while arcs in the STN are labeled with duration ranges, the arcs of the TPN are labeled with both duration ranges and symbolic constraints such as $Tell(c)$ and $Ask(c)$. The symbolic constraints, which represent conditions regarding the state of the vehicles or their environment, have similar semantics as the constraints of Hierarchical

Constraint Automata. Just as HCA constraints were used to either assert or check for the assertion of a condition, the $\text{Tell}(c)$ symbolic constraint in Figure 3.12b represents the assertion that the condition corresponding to c is true, and the $\text{Ask}(c)$ symbolic constraint represents the requirement that the condition corresponding to c is true. For example, the proposition PATH1=OK in the Enroute activity model corresponds to the condition that the first path is available, so $\text{Ask}(\text{PATH1=OK})$ represents the requirement that the path is available. The usage and interpretation of symbolic constraints are described in section 3.4.1. This representation is different from HCA in that there is no distinction made between states and transitions. This uniformity of this representation serves to simplify the planning algorithm.

The TPN also augments the STN with decision nodes that can be used to express a choice between a set of ways of performing some part of an activity. For example, in Figure 3.12b, the node with the double outline represents a decision node, and the dashed arcs out of that node represent the available choices, of which exactly one must be chosen. In the Enroute activity example in Figure 3.2, node 3 is a decision node, which represents the decision between the two paths for the group to travel to the target area. The dashed arc (3,4) represents the option to take path one, and the dashed arc (3,6) represents the option to take path two. Section 3.4.2 discusses the decision node and related representational issues.

One other subtle difference between the Temporal Planning Network and the Simple Temporal Network is that arcs represent both temporal constraints and dependencies or causal links. While each temporal constraint in an STN can be reversed, with the modification of the duration label, it cannot be reversed in the Temporal Planning Network because it changes the direction of dependency. The significance of this becomes more clear in chapter four with the description of the planning algorithm. Essentially, the planner discovers the plan by using a network search to explore trajectories or paths through the network, since the paths through the network correspond to executions of activities. The direction of the arcs is important because the network search used by the planning algorithm only follows forward arcs, not reverse arcs, which is necessary for the correctness of the planning algorithm. In addition, the arcs represent precedence constraints, so a directed path through the Temporal Planning Network defines a chronological ordering of activities that form an execution thread. The interpretation of the directedness of these arcs becomes more problematic when negative temporal constraints are permitted. This issue is discussed in the Future Work section of Chapter 5.

Symbolic Constraints

Recall that HCA models use constraint labels to represent both assertions and requirements. If the constraint was attached to a state, then it represented an assertion of the condition corresponding to the symbolic constraint, but if the constraint was attached to an arc, then it represented a requirement that the corresponding condition be true in order to follow a transition. Temporal Planning Networks also use this representation to model both the requirement and the assertion of conditions.

One difference is that, whereas in HCA the usage of the symbolic constraints was implicit in their placement on either a state or an arc, all symbolic constraints in a Temporal Planning Network are attached to arcs, so context cannot be used to distinguish one usage from the other. Therefore, symbolic constraints in a Temporal Planning Network consist of two parts, a symbol and a type classifier. The symbol represents a condition, or the negation of a condition if qualified by a `Not`. The type classifier indicates what the symbolic constraint is saying about the condition corresponding to the symbol. If the type is `Ask`, then the symbolic constraint represents the requirement that a condition be true. If the type is `Tell`, then it represents the assertion that a condition is true.

```
symbolic constraint = Tell(c) | Ask(c)
c = proposition | Not(proposition)
```

Figure 3.13 Symbolic constraints

Symbolic constraints in a Temporal Planning Network must always be temporally qualified by being attached to an arc. Since each arc represents an interval of time, the association of a symbolic constraint with an arc represents the requirement or assertion of a condition over this interval. A `Tell(c)` label on an arc (i, j) would assert that the condition represented by c is true over the interval between the temporal events modeled by the nodes i and j .

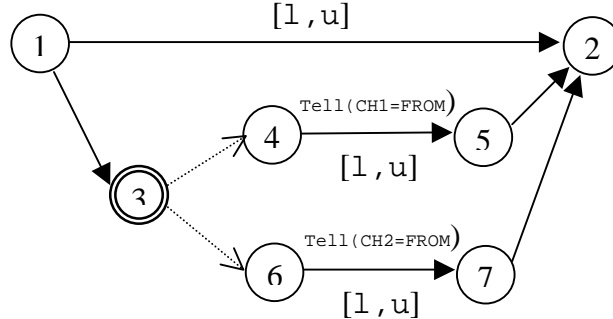


Figure 3.14 Model of the single vehicle *Transmit* activity (Note: zero-duration labels are omitted for clarity)

For example, consider the TPN model of the single vehicle *Transmit* activity in Figure 3.14, where node 1 represents the start event of the activity, node 2 represents the end event, and $[1, u]$ is the uninstantiated duration bound. The `Tell(CH1=FROM)` symbolic constraint label on arc (4,5) asserts that the first communication channel is being used over the duration of the transmission activity. This is similar to the concept of a token in most temporal planning systems, with the addition that the arc indicates what activities must precede and follow its corresponding activity.

Similarly, an $\text{Ask}(c)$ label on an arc (i, j) would require that the condition represented by c is true over the interval represented by this arc. For example, in Figure 3.2, the $\text{Ask}(\text{PATH1=OK})$ label on the arc $(3,4)$ represents the requirement for path one to be available for the interval of time corresponding to the interval of time between the temporal event modeled by node 3 and node 4. These Ask-type symbolic constraints allow for the encoding of conditional execution in the network, which is a key extension beyond traditional planning representations.

Decision Nodes

The decision nodes are used to explicitly introduce choices in activity execution into the activity models. These represent explicit decisions that the planner must make. For example, in the Enroute activity presented at the beginning of this chapter, there are two choices of paths for the group to use for flying to the target area, path one and path two. The activity model captures the two choices as out-arcs of decision node, node 3 in Figure 3.2, graphically designated as a decision node by the double outline and dashed out-arcs. All other nodes in this activity model are non-decision nodes.

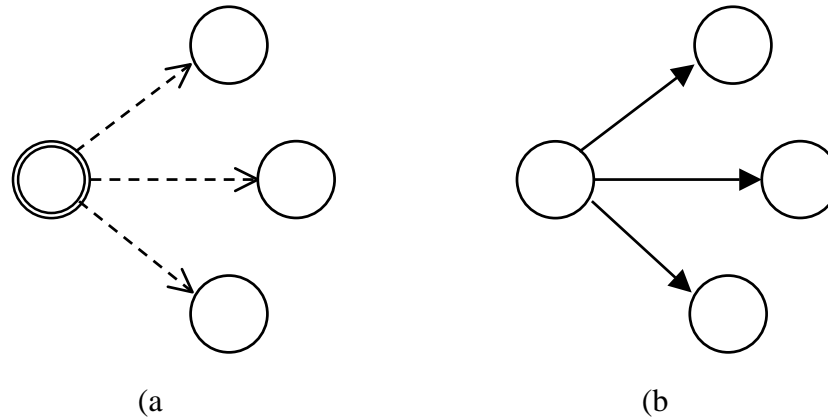


Figure 3.15 (a) Decision node, (b) Non-decision node

As mentioned briefly before in section 3.4, a TPN activity model can be viewed as an expansion of the possible executions of the activity, and executions consist of one or more paths through the network. Decision nodes encode non-deterministic choice in this sense because they represent points in the network where executions may diverge in one case versus another. This is the second key addition, besides the Ask-type symbolic constraints, that allow unconditional temporal plans to generalize to the full expressiveness of RMPL (with the exception of probabilistic and utility-based choice) and time constraints.

Composition

With the basic components of the Temporal Planning Network, it is possible to compose instances of very simple activity models, like those described in the previous section, into arbitrarily complex macro-activity models. One simple way to compose simpler activity models is by placing them in series to model a higher-level activity that performs a sequence of simple activities. Alternatively, these activity models can be placed in parallel to describe an activity in which multiple activities are performed at the same time.

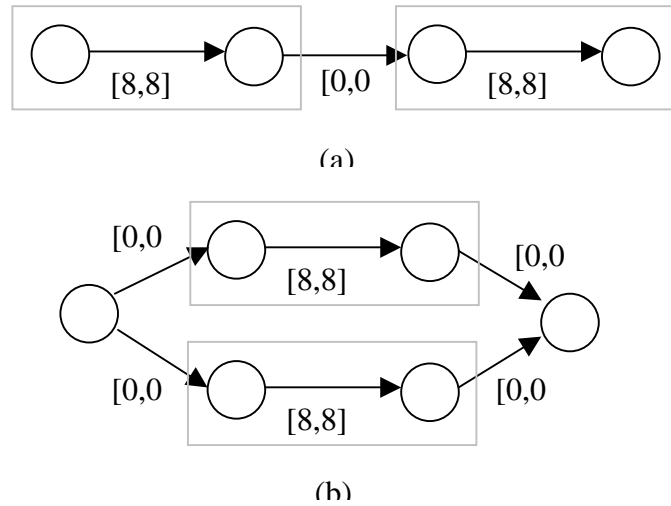


Figure 3.16 (a) Series composition, (b) Parallel composition (Note: Sub-activities are outlined)

Serial composition and parallel composition rely on the introduction of temporal constraint connectors between instances of activity models, but depending on the actual constraint, the composition may have different meanings. For example, constructing a sequence of two activities connected by zero duration arcs represents an activity that executes these activities one immediately after another.

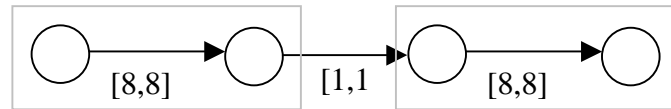


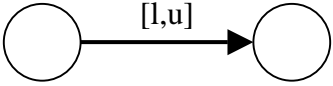
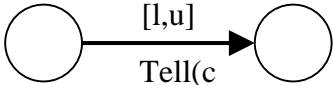
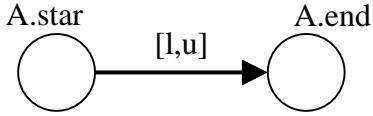
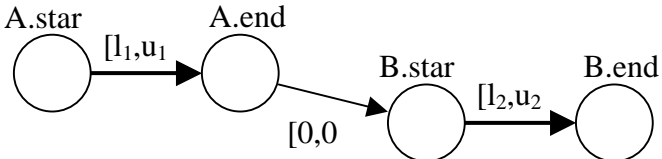
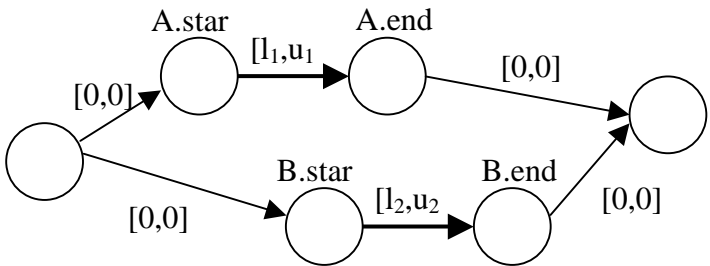
Figure 3.17 Example of a non-zero temporal constraint between activities

Labeling the intermediate arcs with a non-zero duration range (i.e. $[1, u]$ is not $[0, 0]$), changes this meaning. Note, 1 and u must both be greater than or equal to zero, for reasons described in the Future Work section of Chapter 5. This arc acts as a temporal spacer, separating the end of one activity from the start of another by at least 1 time units and at most u time units. Similarly, in the case of the parallel composition of two activities, if only zero duration arcs are used as the connectors, then both activities must commence and complete at the same time. By using positive duration arcs instead,

the parallel composition of the two activities means they should be performed asynchronously, although the activities may still begin and end at the same time.

AML to TPN Mapping

Given the behaviors that can be encoded by the Temporal Planning Network representation, it is possible to map each of the Activity Modeling Language combinators to a TPN model. Once the combinators of AML are mapped to a TPN representation, it is possible to compile arbitrarily complex AML descriptions into a TPN activity model. These mappings are listed in Figure 3.18.

Interval: [l,u]	
Interval + Assertion: c[l,u]	
Interval + Activity: A[l,u]	
Sequential Composition: A[l ₁ ,u ₁]; B[l ₂ ,u ₂]	
Parallel Composition: A[l ₁ ,u ₁], B[l ₂ ,u ₂]	

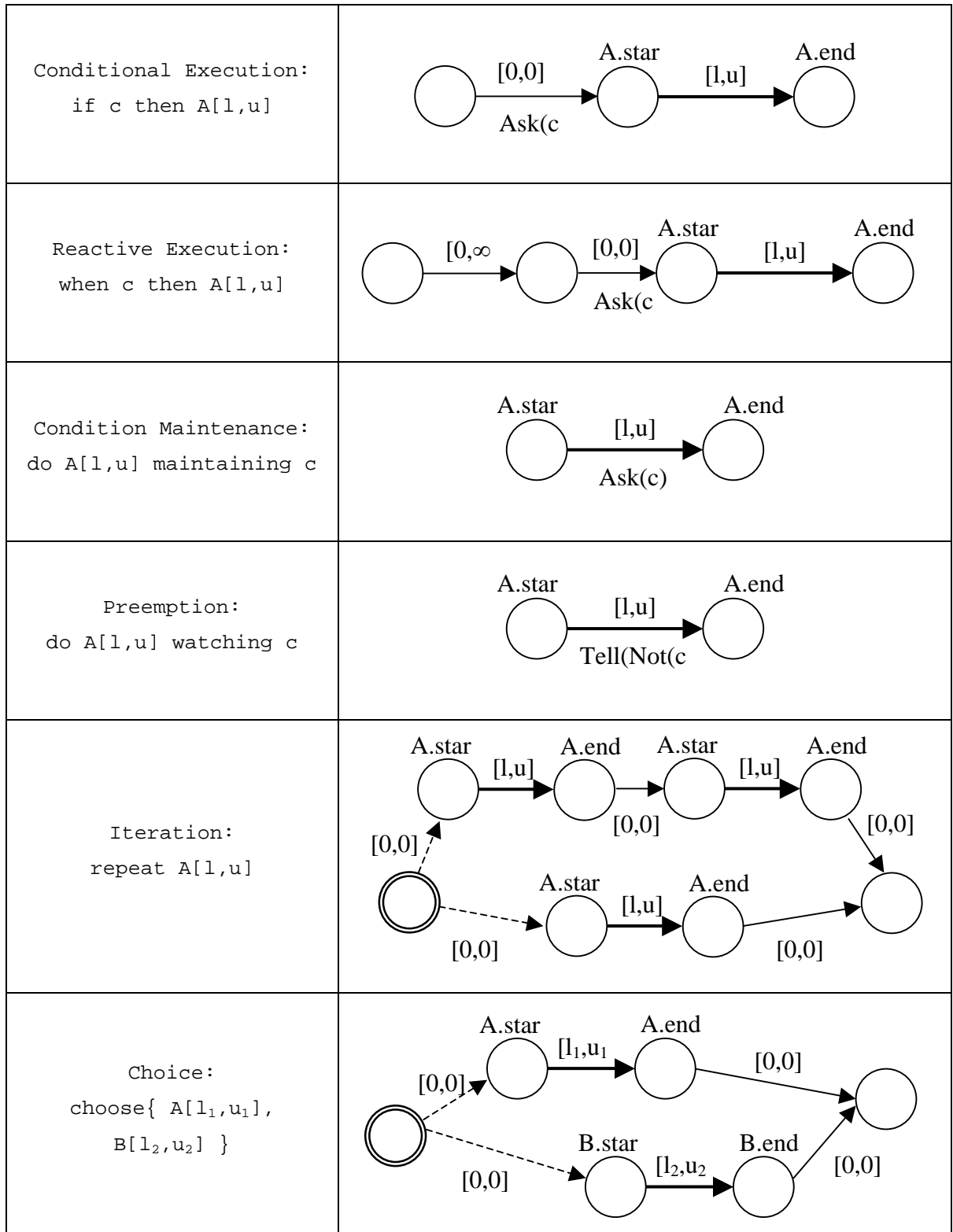


Figure 3.18 Mapping from AML combinators to TPN representation

Many of the TPN representations of combinators in Figure 3.18 contain zero-duration temporal constraints. For example, if two activities A and B are composed sequentially in the AML expression A; B then this maps to an TPN model in which the end-node of the TPN model of activity A is connected to the start-node of the TPN model of activity B with a zero-duration temporal constraint. This constrains the start time of the end time of the activity to correspond to the same point in time. This allows for the modeling of impossible behaviors such as $c[0,0]; \text{Not}(c)[0,0]$ and if c then Not(c), but this is not a problem because the planner rejects executions containing these. The planning algorithm rejects these behaviors because it is not possible for it to resolve the conflict introduced by these models, which require that two mutually exclusive propositions coexist at a point in time. This is explained further in section 4.3.3.

Note that in the TPN representation, the arcs represent closed intervals. This means that if a Tell-type symbolic constraint were attached to an arc, the corresponding constraint would be asserted for the entire interval represented by the arc, including the time points corresponding to the end points. This means that $c[6,8]; \text{Not}(c)[3,3]$ asserts two conflicting symbolic constraints at the point in time corresponding to both the end time of the $c[6,8]$ and the start time of $\text{Not}(c)[3,3]$. This also means that for the AML expression $c[6,8]; \text{if } c \text{ then } A[1,1]$, the pre-condition on the execution of activity A is always satisfied by the preceding $c[6,8]$.

Consider for a moment the TPN encoding of the AML sequential composition combinator. One problem with representing transitions between activities, as recognized by Muscettola, et al. [12], is the accumulation of latency in executing this plan. The problem is that it takes time for the plan runner to compute which activity should be initiated next because it needs to perform updates to the plan based on when preceding activities have completed. This latency can be made very small, but cannot be eliminated. Currently, this latency is ignored by the TPN activity models on the assumption that this latency is negligible, but this may lead to inconsistent execution, especially with high latency or with very large plans, because this latency tends to be compounded with every activity that is executed. This might make it impossible for an activity to be executed without violating the temporal constraints of the plan. One fix for this might be to replace the $[0,0]$ temporal constraints that currently represent transitions between activities with a $[0,\lambda]$ temporal constraint, where λ is an upper bound estimate on system latency. This solution, however, requires further examination.

UCAV Activity Models

This section describes some of the models generated for multiple UCAV mission planning. The following sub-section introduces some extensions to AML to support the description of hierarchical, group activities. Section 3.6.2 describes some of the primitive activities of a single UCAV, and illustrates how they are composed to form single vehicle macro-activities. Next, these single vehicle activities can be combined to form more elaborate group activities, as described in following section. The hierarchical composition of activities allows the modeler to easily construct very complex coordinated activities out of few lower-level activities. An exhaustive list of AML activity descriptions is included in Appendix A.

AML Scoping

In order to describe a group behavior as the coordinated behaviors of multiple vehicles, it is necessary to have a way to distinguish between the members of the group. For example, in the group Fly-To activity, there is a leader that must navigate and one or more followers that simply listen for instructions and follow the leader. This behavior is described for a group consisting of two vehicles, by the AML code in Figure 3.19.

```
Group-Fly-To(x,y,z) [l,u] = {  
    ONE::Fly-To(x,y,z) [l,u] ,  
    { TWO::Follow(ONE) [l,u] , TWO::Listen() [l,u] }  
}
```

Figure 3.19 AML description of the group Fly-to activity

In this activity description, the two vehicles are distinguished by prefixing an activity instance with a scope specifier that consists of the name of the vehicle and the “: :” delimiter. In the example, ONE::Fly-To(x,y,z) specifies that vehicle ONE should navigate, while vehicle TWO should follow while listening for instructions.

Relative Duration Bounds

It is useful in describing an activity to be able to specify relative duration bounds. For example, in the description of the Enroute activity in Figure 3.1, the Enroute activity’s duration is bounded to be between l and u time units, which are imposed externally by the duration bounds of the overall SEAD mission. At the time of modeling, the particular lower and upper bounds are not known, but it is still useful to provide some guidance for the activity durations of sub-activities so that it can be determined more quickly whether or not the activity can be planned.

AML allows the description of sub-activity durations to be relative to their containing activity. In Figure 3.1, the Group-Fly-to activity is constrained to take at least 90% of the lower bound and at most 90% of the upper bound of the Enroute activity. The alternative to specifying duration bounds in relative terms is to leave them unspecified. The problem with this is that by not properly constraining activity durations it is possible for the planner to produce a valid plan with respect to the constraints modeled, but that the group of vehicles cannot execute. Relative bounds provide one solution to this problem, but another is described in the Future Work section.

Vehicle Activities

It is important to make a distinction between activity model primitives and activity execution primitives. For example, consider the single-vehicle Follow activity. The activity model for Follow is not primitive because its behavior is described in terms of other activity, but Follow is an execution primitive because it is among the set of activities that the vehicle knows how to execute. Note that although the vehicle knows how to execute the Follow activity, it is still necessary to model the Follow activity to represent its effects on the system. For example, while vehicle one is following vehicle

two, it is necessary for vehicle one to be listening for messages on both communication channels, therefore it should not be permissible for vehicle one to send out any messages at any time during this activity. Activity model primitives will be referred to as primitives, while activity execution primitives will be called execution primitives.

Some of the single vehicle execution primitives include `Fly-to(waypoint)`, `Bomb(location)`, and `Listen()`. These models can be described in AML or

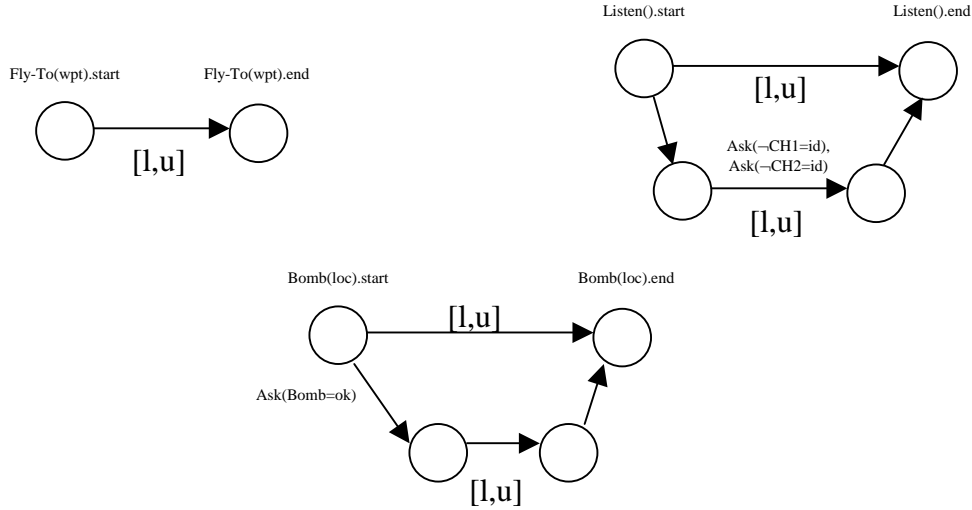


Figure 3.20 The TPN encoding of activity models of the single vehicle primitives Fly-To, Bomb, and Listen (Note: zero-duration bound labels are omitted for clarity)

directly encoded because they are very simple. For illustration, both the TPN models of these execution primitives are provided in Figure 3.20.

```

Fly-Path(wpt1,wpt2,wpt3)[l,u] := {
    Fly-To(wpt1)[l*33%,u*34%];
    Fly-To(wpt2)[l*33%,u*34%];
    Fly-To(wpt3)[l*33%,u*34%]
}

Attack(encrypt,droppt,exitpt,targetpos)[l,u] := {
    Fly-To(encrypt)[l*33%,u*34%];
    Fly-To(droppt)[l*33%,u*34%];
    {
        { Bomb(targetpos)[l,3]; [0,∞] },
        Fly-To(exitpt)[l*33%,u*34%]
    }
}

```

Figure 3.21 AML description of two single-vehicle macro-activities

With these primitives, it is possible to describe higher-level single vehicle activities. For example, $\text{Fly-Path}(\text{wpt1}, \text{wpt2}, \text{wpt3})$ describes the behavior of a vehicle flying along a path defined by three waypoints in terms of three sequential $\text{Fly-To}(\text{waypoint})$ activities. The Attack activity is described in terms of a series of $\text{Fly-To}(\text{waypoint})$ activities and a $\text{Bomb}(\text{location})$ activity, as shown in Figure 3.21. Each of these can be compiled into a TPN encoding in which the activity is represented by the composition of these execution primitives, as shown in Figure 3.22 for the Attack activity.

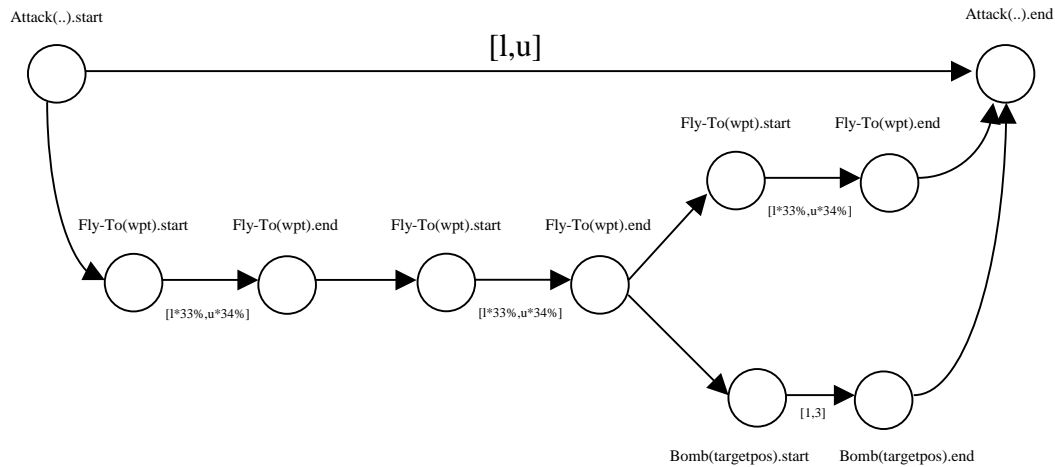


Figure 3.22 TPN encoding of the single-vehicle Attack activity

Group Activities

Once the set of single-vehicle activities are modeled, they can be composed to describe multiple-vehicle activities. The activity models described in this section involve only two vehicles, but there is no limit to the number of vehicles in a group. Group-activities may be composed of any combination of single-vehicle activities. For example, the Group-Attack activity is composed of the single-vehicle Attack activity and a Lookout activity. These activities are assigned to the members of the group by using the scope operator. In Figure 3.23, vehicle one is assigned the Attack activity ($\text{ONE}::\text{Attack}(\dots)$) and vehicle two is assigned the Lookout activity ($\text{TWO}::\text{Lookout}(\dots)$).

```
Group-Attack(encrypt,droppt,exitpt,targetpos)[l,u] := {
    ONE::Attack(encrypt,droppt,exitpt,targetpos)[l,u],
    TWO::Lookout(ONE)[l,u]
}
```

Figure 3.23 AML description of a multiple-vehicle activity

For example, in the AML description of the Group-Attack activity in Figure 3.23 specifies that vehicle one is responsible for attacking the target, and vehicle two serves as lookout for vehicle one. The corresponding TPN network is shown in Figure 3.24.

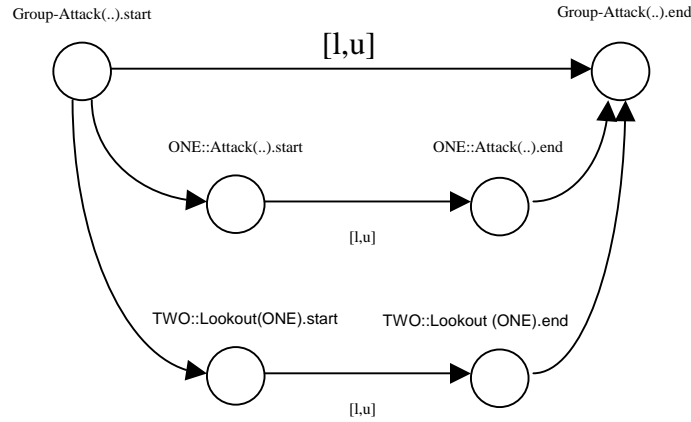


Figure 3.24 TPN encoding of the multiple-vehicle Group-Attack activity

Summary

The Activity Modeling Language extends reactive programming to handle the expression of time critical events and contingencies. AML is a variant of the Reactive Model-based Programming Language [25] that inherits its combinators, with the exception of probabilistic and utility-based choice, and augments this set with an interval construct that is used to express temporal constraints. The Temporal Planning Network brings together the representations of Hierarchical Constraint Automata [25] and temporal plans [7,6,24], by extending unconditional, concurrent, temporal plans to allow for the encoding of conditional execution and non-deterministic choice. This provides a simple and elegant unification of domain modeling and planning.

Chapter 4

Planning Algorithm

Overview

The planner described in this thesis works by searching over the space of all plans to find one that is both complete and consistent. A plan is complete if choices have been made for each relevant decision point and it contains only primitive-level activities, and a plan is consistent if it does not violate any of its temporal constraints or symbolic constraints. One problem with planning in general is the incredibly large search-space. For generative planners [21,17], the search-space is exponential in the plan length because, in the worst-case, operators can be chained together in arbitrary orderings. As with hierarchical planners [17], this planner uses activity models which restrict this type of explosion in the search-space of plans by specifying, at least partially, the precedence relations of activities and by limiting the choices of activities at explicitly defined decision points. These algorithms are exponential in the depth of the hierarchy, which is typically shallow, and therefore perform significantly faster than non-hierarchical planners on non-trivial problems.

However, this planner has the added complexity of dealing with metric time. Even though the activity models cut down the number of possible plans, the expansion into the time dimension makes this space intractably large, which makes it time consuming for the planner to explore anything but a small portion of the space. Therefore, rather than examining individual plans in the space, the planner uses ideas from abstract planning [17] to make decisions that, in effect, rule out sections of the plan-space until all the plans left in the plan-space are both complete and consistent. If no feasible plan is found, then the planner backtracks and makes a new set of decisions that rule out different parts of the space, repeating until a plan is found or it is determined that no valid plan exists.

This planning algorithm uses the same fast temporal reasoning techniques as other temporal planners such as HSTS, but gains additional efficiency by pre-generating networks, in the form of TPN models, representing the possible executions of activities, and searching over this pre-generated structure to identify valid executions. This technique is faster than HSTS and similar temporal planners that employ classical partial order planning techniques [18] because it avoids having to compose the activities online to examine possible executions. This concept of using pre-generated structures to gain run-time efficiency has emerged recently in Artificial Intelligence Planning with methods such as Graphplan [2], SAT-plan [18,20], and Livingstone [23], and has also been used in the verification community for model checking. It is only beginning to be explored for temporal planning, most notably by Temporal Graphplan [26].

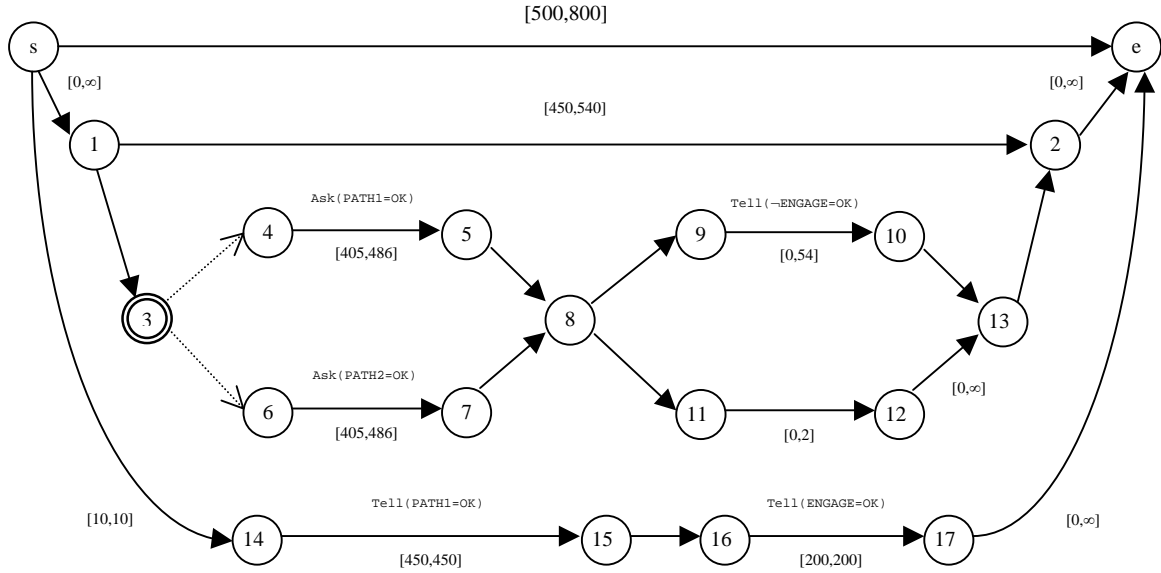


Figure 4.1 A Temporal Planning Network activity model of a scenario

The input to this planner is an incomplete plan in the form of a Planning Network, which describes the mission scenario. A scenario consists of an activity model of the mission or other top-level activity and any mission-specific constraints. For example, Figure 4.1 illustrates a sample input to the planner, whose top-level activity (defined by nodes 1-13) is the Enroute activity from the previous chapter (Figure 3.2). The scenario also defines the time ranges over which path one is available (defined by nodes 14 and 15) and the interval over which the vehicles are allowed to engage the target (defined by nodes 16 and 17). These intervals are defined with respect to the beginning of the scenario, which represents at a fixed time, such as 8:00AM.

In general, the input network is an incomplete plan that encodes the explicit decisions and the implicit decisions that have to be made by the planner. The explicit decisions are encoded as decision nodes in the Planning Network, and the implicit decisions, including decisions about when activities should be executed and decisions about how to satisfy symbolic constraints, are determined from the temporal and symbolic constraints of the network.

As stated in the Chapter 3, each TPN encoding specifies the valid executions of an activity in terms of a set of temporal and symbolic constraints. A path through the network, from the start-node to the end-node of the top-level activity, represents a thread of execution. Since a plan simply describes a set of threads of execution, the output of the planner consists of a set of paths through the input network. For example, Figure 4.2 illustrates a possible plan for the Enroute activity. The portions of the TPN input scenario model that the planner did not select for execution are shown in gray.

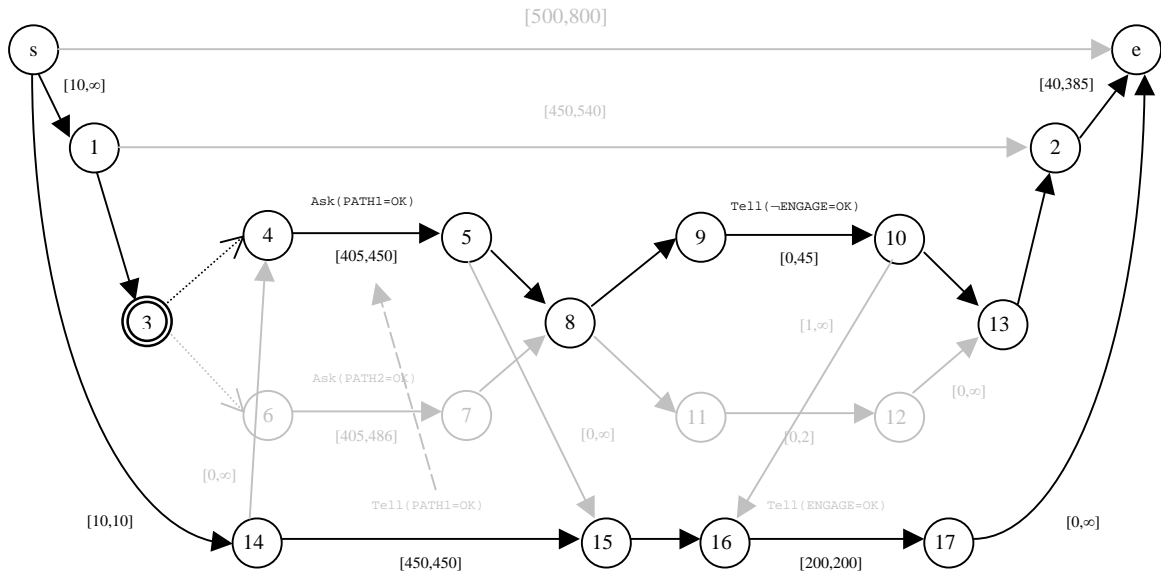


Figure 4.2 Example of a plan (in black) for the scenario described in Figure 4.1

The paths $s-1-3-4-5-8-9-10-13-2-e$ and $s-14-15-16-17-e$ define a consistent execution of the scenario described in Figure 4.1. The first path defines the execution of the group of vehicles, and the second path defines the execution of the rest of the world in terms of the assertion or requirement of relevant conditions over the duration of the scenario. Note that the duration bounds on some of the arcs have been tightened, for example, the duration bound label on arc (4,5). These bounds are tightened by the planning algorithm in order to be consistent with both temporal and symbolic constraints imposed by the scenario definition. This particular duration bound on (4,5) is tightened in order for its $Ask(PATH1=OK)$ symbolic constraint to be consistent with the $Tell(PATH1=OK)$ constraint asserted over the interval defined by arc (14,15).

The planning algorithm can be broken up into three phases. The first phase resembles a network search that discovers the sub-network, or alternatively the set of paths, that constitute a feasible plan, while incrementally checking for temporal consistency. The second phase is analogous to the repair step of a hierarchical planner, in which symbolic constraint conflicts are detected and resolved by promotion or demotion and open conditions are covered [17]. The third phase performs the decomposition of macro-activities and recursive planning of these activities. The Planning Algorithm pulls these phases together as described in Figure 4.3. The remainder of this chapter will describe these phases in more detail.

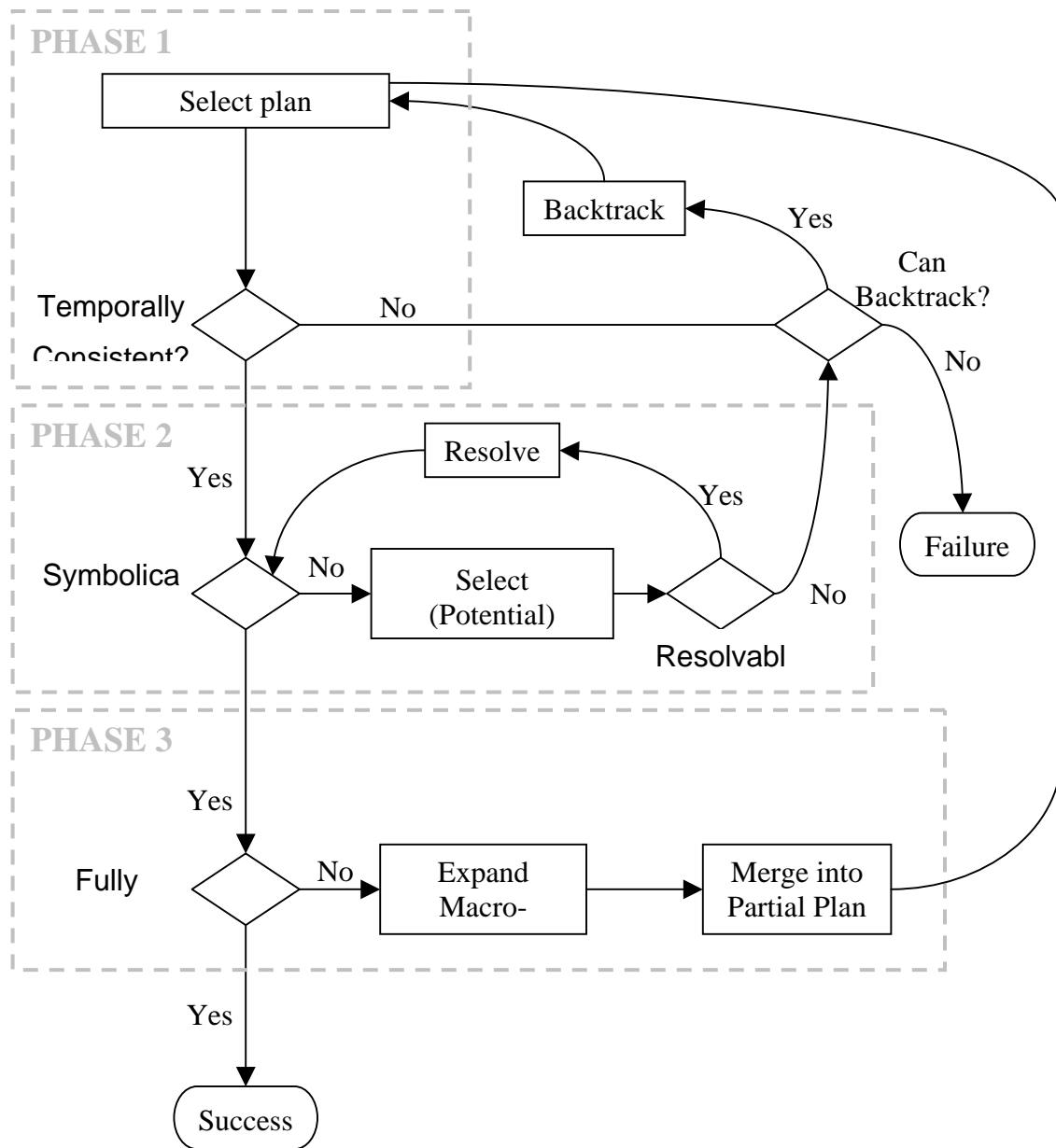


Figure 4.3 Flowchart representation of the Planning Algorithm

Phase One: Select Plan Execution

Network Search

This phase of the planning algorithm selects a set of paths from the start-node to the end-node of the top-level activity (the activity which is being planned). These paths correspond to threads of execution that together describe a plan for executing the top-level activity. The planner handles this execution selection problem as a network search rooted at the start-node of the TPN encoding of the top-level activity. If there were no decision nodes, this search would incrementally extend a set of paths from the start-node, through all forward arcs, until all paths reached the end-node of the top-level activity. This search is slightly modified to handle decision nodes.

As stated in Chapter 3, each node of a Temporal Planning Network is either a decision

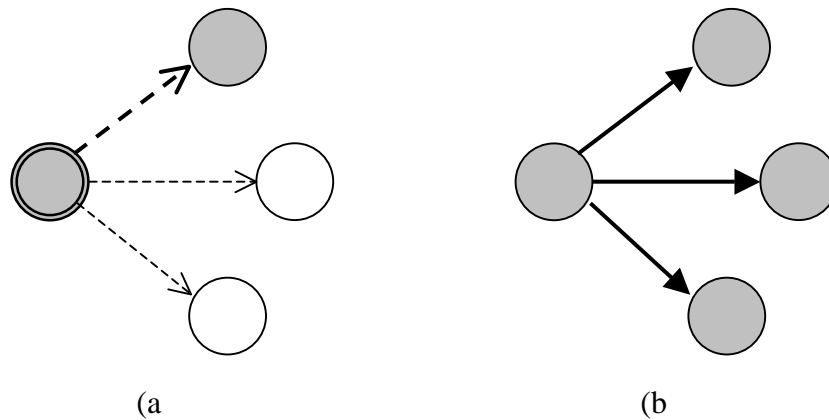


Figure 4.4 (a) Decision node, (b) Non-decision node; Shaded nodes and bold arcs are selected.

node or a non-decision node. If a plan includes a non-decision node with multiple out-arcs, then all of these arcs and their tail nodes are also included in the plan. If a plan includes a decision node with multiple out-arcs, then the arcs represent alternate choices, and the planning algorithm selects exactly one.

The first phase of the planning algorithm is complete only when all paths reach the end-node of the top-level activity and all of these paths, which define a sub-network of the original Planning Network, are temporally consistent. For now, assume that there is an efficient function for testing temporal consistency. The first phase of the planning algorithm is summarized in pseudo-code as the Modified Network Search algorithm in Figure 4.6. For comparison, a generic network search algorithm [1] that is used for network exploration and reachability analysis is presented in Figure 4.5. The set A , is the set of active nodes, which are those nodes from which paths have yet to be fully extended. The sets S_N and S_A are the sets of selected nodes and selected arcs, respectively.

```
1  Network-Search( N )  
2      A = { start-node of N };
```

```

3      SN = { start-node of N };
4      SA = { };
5      While ( A is not empty )
6          Node = Select and remove a member of A;
7          For each Arc that is an out-arc of Node
8              If ( tail of Arc is not in SN )
9                  Add Arc to SA and
10                 Add tail of Arc to A and SN;
11              End-If
12          End-For
13      End-While
14  End-Function

```

Figure 4.5 A generic network search algorithm

There are a few differences between the generic network search and the modified network search used for the first phase of the planning algorithm. The first difference is that for a generic network search, every node is handled as a non-decision node (Figure 4.5 lines 7-12), whereas this modified search has an added clause to handle decision nodes differently (Figure 4.6 lines 8-13 and lines 15-20), restricting the extension of paths through out-arcs to a single arc.

The second difference is that at the end of each iteration of the main While-loop, the modified network search tests for temporal consistency (Figure 4.6 lines 24-26). If the test fails, then the search calls the Backtrack(..) function in line 25 which reverts S_N, S_A, and A to their states before the most recent decision for which there remain unmarked choices, and selects a different out-arc. In this planning algorithm, chronological backtracking is used but a wealth of more efficient search algorithms exist.

Checking for temporal consistency after every iteration of the While-loop is unnecessary because as long as no cycles are induced in the network, there is no way for a temporal inconsistency to be induced (see next section for explanation). Determining whether a cycle has been created can be done for each arc that is selected by checking whether the arc's tail node has already been selected. Since this can be done in constant time, this can be significantly more efficient than testing temporal consistency after every iteration, although in the worst case these two methods take the same asymptotic running time.

```

1      Modified-Network-Search( N )
2      A = { start-node of N };
3      SN = { start-node of N };
4      SA = { };
5      While ( A is not empty )
6          Node = Select and remove a member of A;
7          If ( Node is a decision-node )
8              Arc = Select any unmarked out-arc of Node and
9              Mark Arc and
10             Add Arc to SA;
11             If ( tail of Arc is not in SN )
12                 Add tail of Arc to A and SN;
13             End-If

```

```

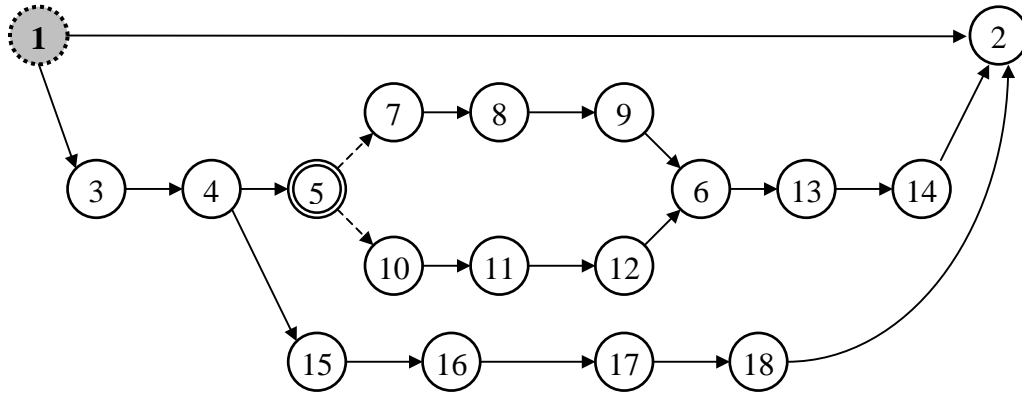
14         Else
15             For each Arc that is an out-arc of Node
16                 Add Arc to  $S_A$ ;
17                 If ( tail of Arc is not in  $S_N$  )
18                     Add tail of Arc to A and  $S_N$ ;
19                 End-If
20             End-For
21         End-If
22
23         If ( Cycle-Induced( $S_N$ ,  $S_A$ ) )
24             If ( Not(Temporally-Consistent( $S_N$ ,  $S_A$ )) )
25                 Backtrack( $S_N$ ,  $S_A$ , A);
26             End-If
27         End-If
28     End-While
29 End-Function

```

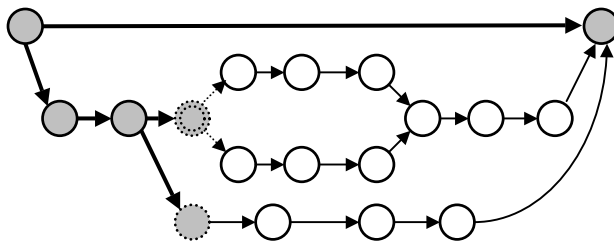
Figure 4.6 Planning Algorithm Phase One: Modified Network Search

Finally, after a generic network search is complete, the set of selected nodes and arcs, S_N and S_A , define a tree rooted at the start-node of N and extending to all nodes that are reachable from it. However, after a modified network search, the selected nodes and arcs instead define a set of paths from the start-node of N to the end-node of N. This is why lines 10-13 and 16-19 of Figure 4.6 differ from the analogous lines of the generic network search algorithm, lines 8-11 of Figure 4.5.

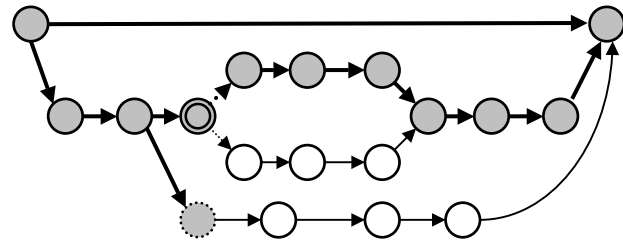
The modified network search algorithm in Figure 4.6 does not seem to fully extend paths from the start-node to the end-node. In fact, it stops extending paths when it encounters a node that is already in S_N . However, the fact that this node is already in S_N implies that two concurrent threads of execution have merged. Continuing the search by extending both paths would lead to the redundant selection of the set of paths from this node to the end-node. Since there is nothing gained by this redundant selection, fully extending only one of these paths is sufficient.



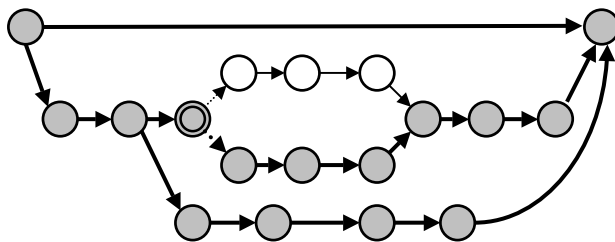
(a)



(b)



(c)



(d)

Figure 4.7 Example of the Modified Network Search Algorithm; (a) Initially, (b) After 3 iterations, (c) Temporal inconsistency detected, (d) Complete

To illustrate the modified network search, consider the input network illustrated in Figure 4.7a, in which node 1 is the start-node and node 2 is the end-node. Initially, node 1 is selected, which is indicated by its darker shade, and it is active, which is indicated by its dashed outline. In the first iteration, it chooses node 1 from the set of active nodes, and since node 1 is not a decision node, it selects all out-arcs and adds their tails to the selected and active set. This continues until both node 5 and node 15 are selected as in Figure 4.7b. At this point, the modified network search chooses node 5 from the active set. Since node 5 is a decision node, the algorithm must choose either arc $(5, 7)$ or arc $(5, 10)$. It selects arc $(5, 7)$ and continues extending until it reaches the state shown in Figure 4.7c.

Note that arc $(14, 2)$ is selected in Figure 4.7c, forming the cycle, $1-3-4-5-7-8-9-6-13-14-2-1$, so the algorithm checks for temporal consistency. In this example, this selected sub-network is temporally inconsistent, so the algorithm backtracks to the most recent decision point at which there are options that have not already been tried. Node 5 is the most recent decision node and out-arc $(5, 10)$ has not yet been tried, so the algorithm reverts the sets of selected nodes and arcs and the set of active nodes to their state before the last decision at node 5, as in Figure 4.7b. The algorithm then selects the arc $(5, 10)$, extends the path through this arc to the end-node, and finally extends the path through arc $(15, 16)$ to the end-node, which results in the temporally consistent sub-network of selected nodes and arcs shown in Figure 4.7d.

Temporal Constraint Consistency

Consider any sub-network of a Planning Network. Disregarding the symbolic constraint labels on the arcs, this sub-network of a Planning Network is a Simple Temporal Network. Since this is the case, testing for temporal consistency of a partial or completed plan can be performed using the same methods as used for Simple Temporal Networks [7,19].

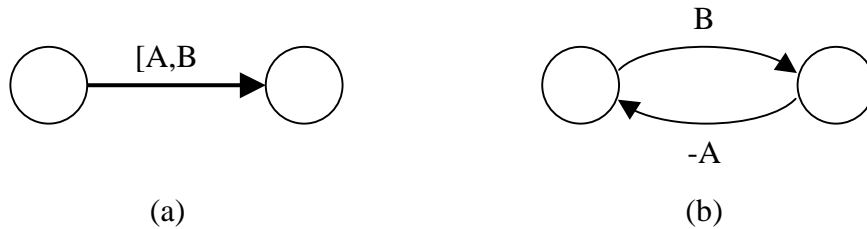


Figure 4.8 Temporal constraint in (a) STN form, (b) Distance graph form

Recall from Section 2.2.2 that each STN can be represented as a distance graph, and an STN is temporally consistent if and only if its distance graph contains no negative cycles [7]. The existence of a negative cycle implies there is a set of temporal constraints that cannot be satisfied. To illustrate a temporal consistency, consider the STN representation of an activity whose duration is lower-bounded by A time units and upper-bounded by B time units (Figure 4.8a). Looking at the distance graph form of this STN in Figure 4.8b,

it is clear that there is a cycle formed by the forward arc and reverse arc, but as long as B is greater than or equal to A the cycle is non-negative. However, if B is less than A , then this becomes a negative cycle. This temporal inconsistency corresponds to the impossible condition of the activity duration's upper bound being less than its lower bound.

Negative Cycle Detection

Fortunately, there are well known algorithms for detecting the presence of negative cycles in polynomial time. The simplest method is to use an all-pairs shortest path algorithm, for example, Floyd-Warshall algorithm or the matrix-multiplication-based all-pairs shortest path algorithm [5]. These algorithms return a distance matrix, D , of n rows and n columns, where n is the number of nodes in the network, such that the $D[i][j]$ is the shortest path length from node i to node j . Note that the diagonal elements ($D[0][0]$, $D[1][1]$, $D[2][2]$, etc.) must always be zero, because there is no distance from a node to itself. However, if there are negative cycles then some of these diagonal elements will be negative.

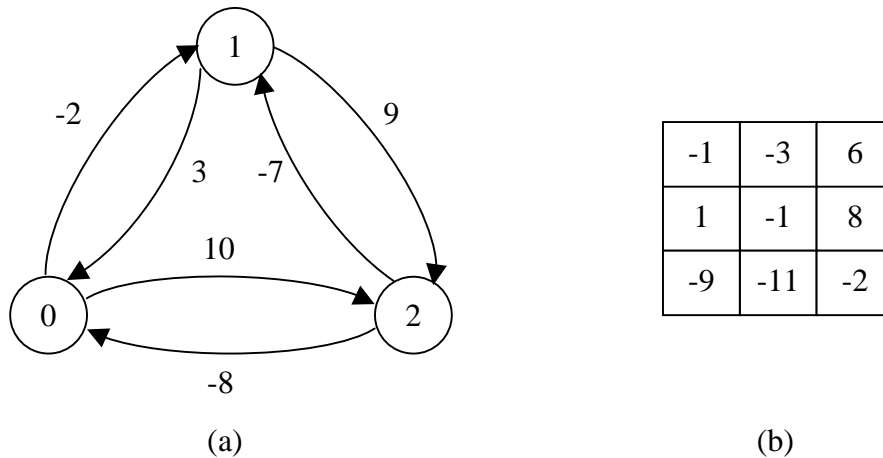


Figure 4.9 (a) Network with a negative cycle 0-1-2-0, (b) All-pairs shortest path distance matrix for this network returned by the Floyd-Warshall algorithm

The Floyd-Warshall algorithm takes $\Theta(n^3)$ time and requires $\Theta(n^2)$ space to store the distance matrix, and the matrix-multiplication-based all-pairs shortest path algorithm takes $\Theta(n^3 \log n)$ time and requires $\Theta(n^2)$ space as well. However, there are methods of detecting negative cycles that are both faster and require less space. The Bellman-Ford algorithm is used to compute single-source shortest paths [5] but also can be used to check for negative cycle in $\Theta(nm)$ time, where m is the number of arcs in the distance graph. In addition, this algorithm only needs to maintain one distance label at each node, which only takes $\Theta(n)$ space. A variant of this algorithm is used by HSTS [12] for fast inconsistency detection.

The algorithm used by the planner described in this thesis is a particular implementation of the generic label-correcting single-source shortest-path algorithm [1], which takes $O(nm)$ worst-case asymptotic running time, but performs faster in many situations. This algorithm also requires only $\Theta(n)$ space. It is very similar to the Bellman-Ford algorithm, which is just a different implementation of the label-correcting single-source shortest path algorithm, except that it uses a different strategy for examining nodes.

<pre> Bellman-Ford (N,source) For i = 1 to nodes of N -1 d(i) = +∞; End-For d(source) = 0; For k = 1 to nodes of N -1 For each node i of N For each arc (i,j) in N If d(j) > d(i)+c(i,j) d(j) = d(i)+c(i,j); End-If End-For End-For End-For End </pre>	<pre> FIFO-label-correcting (N,s) For i = 1 to nodes of N -1 d(i) = +∞; examined_count(i) = 0; End-For d(source) = 0; list = {source}; While (list is not empty) i = pop head of list; examined_count++; If examined_count(i) > n print "Negative Cycle"; Exit-Function; End-If For each arc (i,j) in N If d(j) > d(i)+c(i,j) d(j) = d(i)+c(i,j); If j is not in list push j to end of list; End-If End-If End-For End-While End </pre>
(a)	(b)

Figure 4.10 Single-source shortest path algorithms: (a) Bellman-Ford, (b) FIFO label-correcting

A label-correcting shortest path algorithm works by incrementally updating or correcting distance labels in a monotonically decreasing fashion until the single-source shortest path optimality condition is satisfied, that is, the distance label of every node must be less than or equal to the distance label of any other node plus the distance between them. The Bellman-Ford algorithm always examines each of the nodes n times, performing distance label corrections, which is guaranteed to complete and return the shortest path distance labels as long as there are no negative cycles.

The FIFO implementation, which is used by this planner, of the label-correcting shortest path algorithm only examines each node as many times as the node's distance label may be invalidated [1]. The distance label for node i becomes invalidated only if some node j is examined by the algorithm whose distance label plus the distance from node j to node i is less than the distance label of node i . It is possible for nodes only to be examined a few times, although in the worst case all nodes are examined n times. Another advantage of the FIFO implementation is that it can incrementally check for negative cycles and stop early if one is detected, whereas the Bellman-Ford algorithm must complete examining all the nodes n times before it can detect a negative cycle.

Phase Two: Refine Plan

Symbolic Constraint Consistency

There are two types of symbolic constraint inconsistencies, incompatibilities and open conditions. An incompatibility exists when there are two arcs in the network, representing overlapping intervals of time, which are labeled with symbolic constraints that conflict. Two symbolic constraints conflict if one is either asserting or requesting that a condition is true, and the second is asserting or requesting that the same condition is false. For example, $\text{Tell}(\text{Not}(c))$ and $\text{Ask}(c)$ conflict, as do $\text{Ask}(c)$ and $\text{Ask}(\text{Not}(c))$. Clearly, since such condition pairs can never both be satisfied at the same time, they represent one form of plan inconsistency.

The second type of symbolic constraint inconsistency is an open condition, which is defined as any unsatisfied condition. In the Planning Network representation, open conditions appear as Ask constraints, which are used to model pre-conditions, post-conditions, and conditional execution as described in the previous chapter. An Ask constraint represents the need for some condition to be true over the interval of time represented by the arc labeled with the Ask constraint.

The second phase of the planning algorithm finds these symbolic constraint inconsistencies and tries to resolve them. If there is an inconsistency that cannot be resolved, then the planner returns to the first phase of planning which needs to make a new set of decisions.

Conflict Detection

Detection of open conditions can be done by scanning through all arcs and checking for Ask constraints. Detecting incompatibilities requires more work because the planner must first compute the feasible time bounds for each temporal event (node) in the network, and then use these bounds to identify potentially overlapping intervals that are labeled with conflicting symbolic constraints.

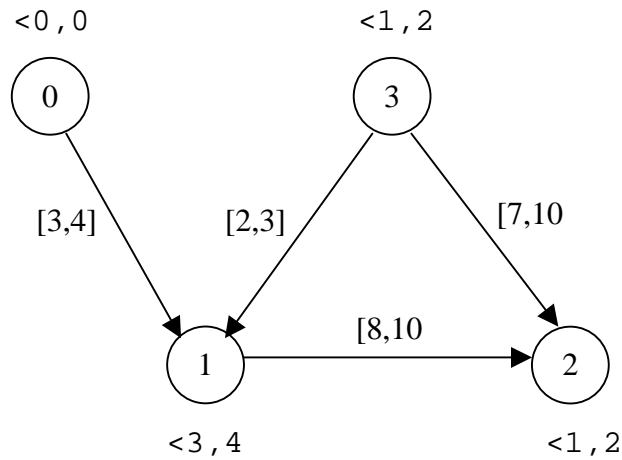


Figure 4.11 Plan fragment with feasible time bound labels

As stated in section 2.2.2, these bounds can be computed by solving an all-pairs shortest-path problem over the distance graph representation of the partially completed plan [7]. The upper bound of the feasible time range for each temporal event is given by the shortest path distance from the origin node to the node representing the temporal event. The lower bound is given by the negative shortest path distance from the node representing the temporal event to the origin. These bound the time of the event with respect to the fixed time of the origin node.

For example, consider the plan fragment shown in Figure 4.11, in which node 0 is the artificially introduced origin node whose time is fixed to 8:00am, and the time units are minutes. The feasible times for event 2 to begin are any time between 8:11am and 8:12am, because the shortest path distance from the origin to node 2 is 12 time units, and the shortest path distance from node 2 to the origin is -11 time units, as shown with the analogous distance graph in Figure 4.12.

This planner uses the Floyd-Warshall algorithm for computing all-pairs shortest paths because of ease of implementation. However, there are alternative algorithms that may outperform this one. For example, Johnson's algorithm (also for computing all-pairs shortest paths) has better asymptotic running time than Floyd-Warshall on networks in which the number of arcs is much less than $O(n^2)$. While Floyd-Warshall runs in $\Theta(n^3)$ time, Johnson's algorithm [5] can be implemented to run in $O(n^2 \log n + mn)$, which becomes $O(n^2 \log n)$ if $m=O(n)$. Recall from section 2.2.2 that while temporal inconsistencies can be detected by solving a single-source shortest path, computing the

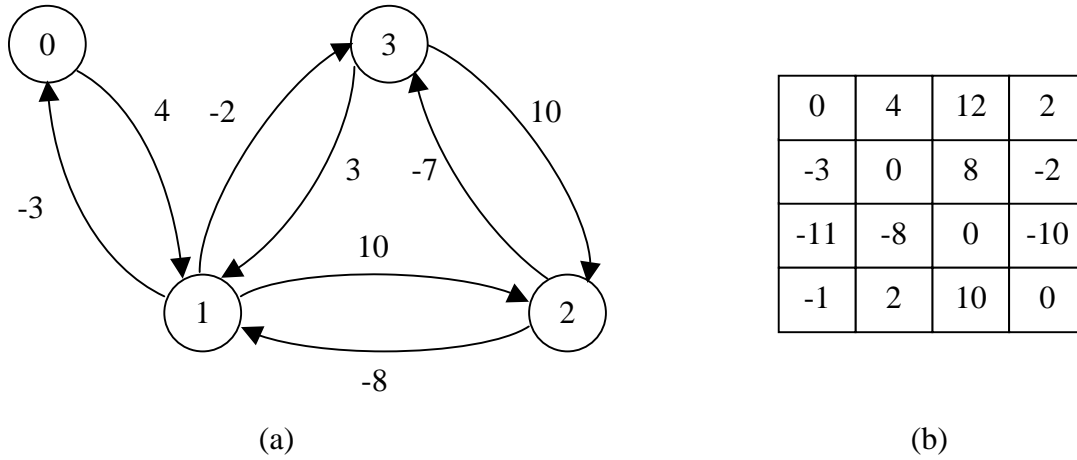


Figure 4.12 (a) Distance graph representation of temporal constraint system, (b) All-pairs shortest path distance matrix

feasible time bounds of temporal events requires solving an all-pairs shortest path problem.

Once these feasible time ranges are determined, the planner can detect which arcs may overlap in time. If there are two arcs that may overlap and that are labeled with conflicting symbolic constraints, then they are resolved by ordering the intervals if possible, as described in the next section. However, it can be expensive to go through all pairs of arcs to check for conflicting constraints. In fact, if there are s different symbols and m arcs in the network, then this method takes $\Theta(sm^2)$ time.

For each constraint in the network, the planner maintains an interval set data structure that keeps track of all of the intervals that assert or require the condition represented by that constraint or its negation. In order to identify conflicts, the planner need only check each interval set for conflicts. This takes $O(si^2)$ asymptotic running time, where i is the maximum cardinality over all interval sets. This is at least as good as the brute-force method described in the previous paragraph, since in the worst case $i=O(m)$. However, it performs much better in practice because most of the interval sets have very few elements.

An alternative to the interval set that was not implemented is the interval tree data structure [5]. Interval trees are used to store a set of intervals keyed by their low

endpoint, and they support interval insertion, deletion, and overlap search in $\log n$ time, where n is the number of stored intervals. Each arc is represented by an interval whose start-time is the earliest absolute time of either end-point of the arc, and whose end-time is the latest absolute time of either end-point of the arc. Using this data structure results in the same worst-case running time of $O(sm^2)$, but may it may lead to even better performance in practice than using the interval set.

Conflict Resolution

Both incompatibilities and open conditions are handled by introducing additional temporal constraints into the plan. Each incompatibility consists of two arcs that represent intervals of time that may overlap, but in some cases it is possible to further constrain the time ranges of the start- and end-points of the intervals to ensure that they will not overlap. This is done by employing the standard threat resolution technique of hierarchical planners (promotion/demotion) [17,21], by introducing temporal constraints that force orderings.

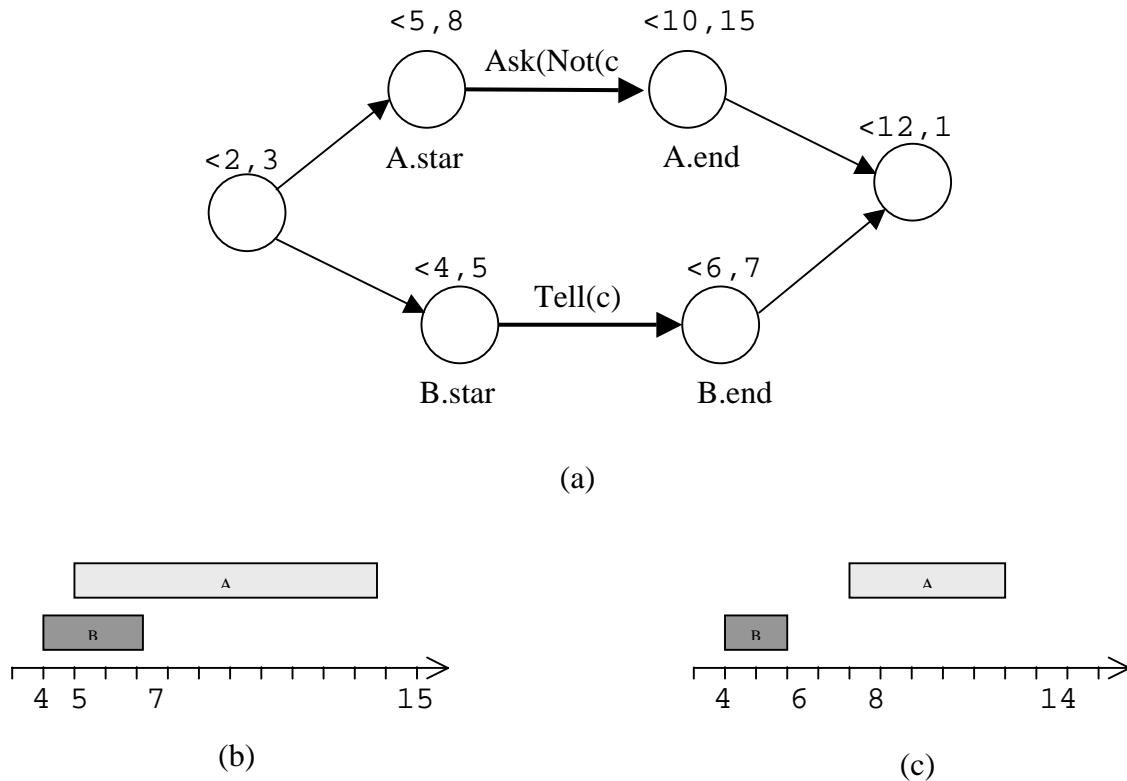


Figure 4.13 (a) Plan fragment containing an incompatibility, (b) & (c) Two possible scenarios of how activities A and B may be performed

For example, consider the plan fragment containing two activities with conflicting symbolic constraints in Figure 4.13a, in which the feasible times for each event are contained in angled brackets at their corresponding nodes. Both Figure 4.13b and Figure 4.13c are valid executions of these planned activities according to the feasible time

ranges of their start and end events. Since $\text{Ask}(\text{Not}(c))$ and $\text{Tell}(c)$ cannot both be satisfied over the period from time 5 to time 7, the execution illustrated in Figure 4.13b is invalid. However, the execution shown in Figure 4.13c is valid, which demonstrates that it is possible to resolve incompatibilities in some cases by further constraining the feasible time ranges of events.

Rather than arbitrarily constraining the time ranges of the interval start- and end-points, the planner introduces orderings to resolve each incompatibility. An ordering pushes one interval before another interval by adding a non-negative temporal constraint from the end-point of the first to the start-point of the second, or vice versa. Note, the temporal constraint used to represent this ordering cannot have a zero lower bound because that would still allow for the end-time of the first activity to be the same as the start-time of the second. Therefore, the temporal constraint used to represent the ordering has a lower bound of ϵ , where ϵ represents the granularity of the time representation. For example, if time were represented in milliseconds, then ϵ would equal 1 millisecond. The need for this positive ϵ is a limitation of this planner that might be resolved by incorporating a dense model of time; this is left for future work.

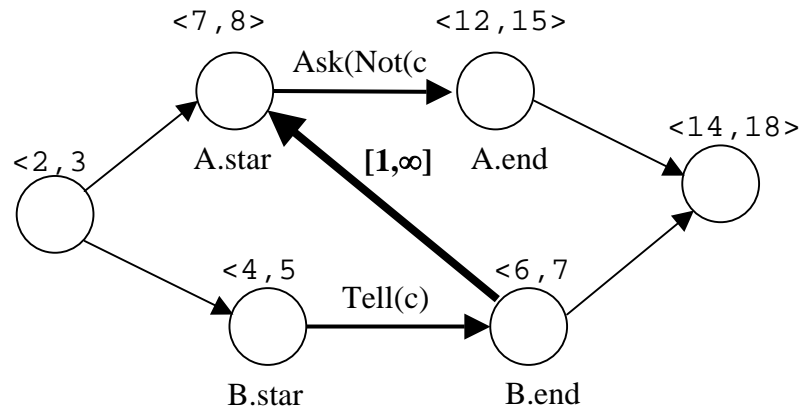


Figure 4.14 The temporal constraint between B.end and A.start represents an ordering (with $\epsilon = 1$) used to resolve the incompatibility illustrated in Figure 4.13

Figure 4.14 shows an ordering, with $\epsilon = 1$, which would have resolved the incompatibility in the plan fragment from Figure 4.13. The other possible ordering in this example, which would force activity B after activity A, induces a temporal inconsistency so it is not an option in this case. Using orderings to constrain the temporal events can repair a plan while retaining as much temporal flexibility as possible.

An open condition is represented by an arc labeled with an Ask constraint, which represents the request for a condition to be satisfied over the interval of time represented by the arc. If this interval of time is contained by another interval over which the condition is asserted by a Tell constraint, then the open condition is satisfied or closed. Finding potentially overlapping intervals can be done using the same method as described in the previous section for detecting incompatibilities.

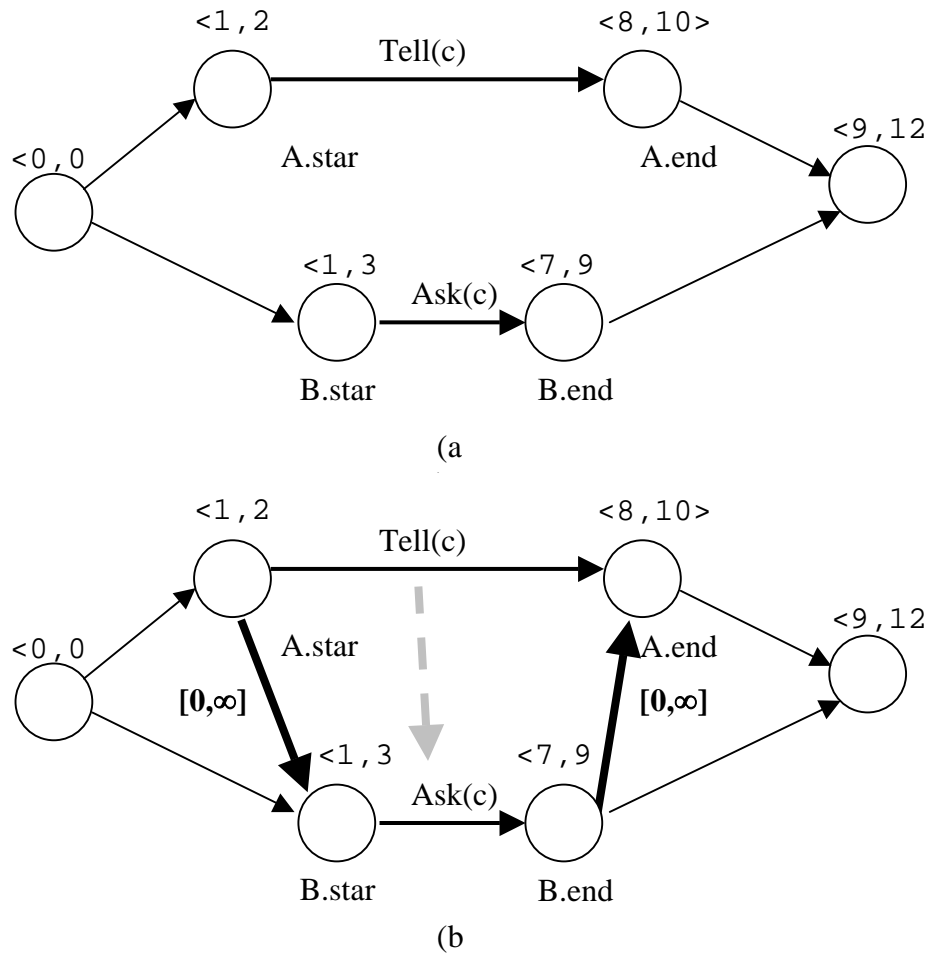


Figure 4.15 (a) Plan fragment in which activity B has an open maintenance condition, (b) Temporal constraints are introduced to satisfy the open condition

Once an interval that may satisfy this open condition is found, temporal constraints can be added to force the interval to contain the interval of the open condition. In classical partial order planners, open conditions are typically preconditions of activities, and each of these open conditions is closed by introducing a causal link from an activity that asserts the condition (as a post-condition) to the activity whose open precondition was satisfied. The method of resolution is the same except that the open conditions may have extended temporal duration, and in order to be satisfied they must be covered by another

interval over which the condition is asserted. This method of closing of open conditions is also closely related to the way that HSTS satisfies compatibilities [12].

In Figure 4.15a, activity B has a maintenance condition represented by the $\text{Ask}(c)$ symbolic constraint label on the temporal constraint between the nodes corresponding to the start of B and the end of B. Figure 4.15b shows how temporal constraints can be added to satisfy this open condition by forcing the interval over which condition c is asserted by the $\text{Tell}(c)$ constraint to contain the interval of the $\text{Ask}(c)$ constraint. In this example, the start of B is constrained to be at the same time or after the start of A, and the end of B is constrained to be at the same time or after the end of A. The gray, dashed arc from the $\text{Tell}(c)$ to the $\text{Ask}(c)$ indicates that the $\text{Ask}(c)$ open condition was closed by this $\text{Tell}(c)$.

Phase Three: Hierarchical Decomposition

The third phase of the planning algorithm performs the incremental decomposition of the portions of the plan representing macro-activities, similar to the decomposition performed by other hierarchical planners [17]. The current implementation of the planner applies this decomposition iteratively during planning, but an alternative would be to fully decompose the top-level TPN activity model offline so that during planning, the planner only needs to perform the network search and refinement (phases one and two). Performing the decomposition online saves memory but may take longer than pre-expanding the top-level activity model if online expansion becomes the efficiency bottleneck. Other methods of addressing this tradeoff should be considered in future work. Only the iterative, online method of hierarchical decomposition is addressed in this section.

All activities are represented in a Temporal Planning Network by a start- and end-node pair, labeled with the name of the activity. The planning algorithm maintains a list of all macro-activity names, and after the first two phases of the planning algorithm complete, the planner scans through the name labels to identify any unexpanded macro-activities. If an unexpanded macro-activity is recognized, then the planner selects a single macro-activity, instantiates a copy of the TPN model of the activity, and merges this network into the partially completed plan. The merge simply superimposes the expanded TPN activity model onto the plan, lining up the start- and end-nodes of the expanded activity model with the start- and end-nodes in the plan.

Figure 4.16 illustrates a plan containing the non-primitive Enroute activity, whose start- and end-events correspond to nodes 1 and 2 in the figure. The planner scans through the partially complete plan, looking for non-primitive activities, and recognizes the Enroute activity whose name is among the set of macro-activity names. The planner then performs a lookup to retrieve and instantiate a copy of the TPN activity model corresponding to the macro-activity. A graphical depiction of the lookup table for the

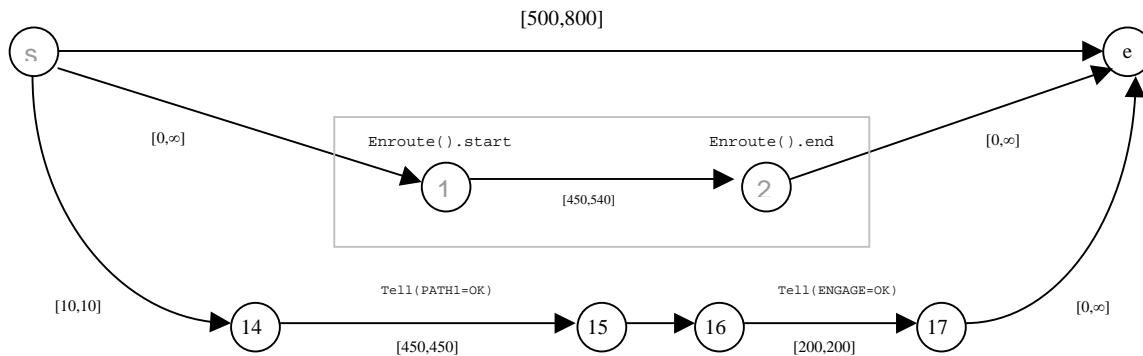


Figure 4.16 A partially complete plan containing a macro-activity represented by nodes 1 and 2
macro-activities is shown in Figure 4.17.

The instantiation first constructs a new copy of the activity model network, and then binds variables, including the activity argument variables and the duration bound variables. For example, the lower and upper duration bounds of the Enroute activity as defined in by the partially complete plan, are passed through to the sub-activities whose duration bounds were defined relatively. The relative bounds for the Group-Fly-Path sub-activity of Enroute, for example, was $[l * 90\%, u * 90\%]$, and would be instantiated to $[405, 486]$ in this case since $l=450$ and $u=540$ for this instance of the Enroute activity.

Once the planner expands the macro-activity, by instantiating a copy of the corresponding activity model, it merges the expanded activity model into the network between the node pair representing the macro-activity's start and end events. For example, the planner would merge the $\text{Enroute}() [450, 540]$ activity model instance into the partial plan in Figure 4.16 between nodes 1 and 2, yielding the TPN shown in Figure 4.1, at the beginning of this chapter. The final step of merging is adding the start-node of the macro-activity to the set of active nodes maintained for the modified network search of phase one.

Activity Name	TPN Activity Model
---------------	--------------------

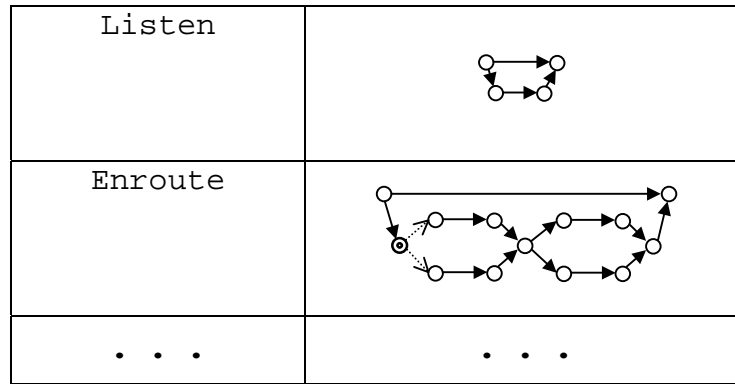


Figure 4.17 Partial macro-activity map maintained by the planner

After the macro-activity is expanded and merged into the partial plan, the planning algorithm returns to phase one to search through the sub-network corresponding to the newly expanded macro-activity for a set of paths that define a valid execution of this macro-activity within the larger context of the plan. The planning algorithm iteratively decomposes all macro-activities in the plan in this manner until they have all been fully expanded, and this plan is returned.

Chapter 5

Conclusions

The research described in this thesis focused on the development of a planning system for coordinated air vehicle missions. In order to support this planning system, the Activity Modeling Language (AML) was created for facilitating the construction of complex coordinated activity models, by extending the Reactive Model-based Programming Language to allow the expression of metric time constraints. Furthermore, a new encoding for the hierarchical activity models described in AML, the Temporal Planning Network (TPN), was introduced by drawing together ideas of temporal constraint representation and reasoning with Simple Temporal Networks [7] and hierarchical, constraint-based modeling with Hierarchical Constraint Automata [25]. These together provide a natural and expressive language for describing complex coordinated activities, and an encoding for the activity models that support efficient planning, respectively. Finally, this thesis describes a planning algorithm for rapidly generating multiple-UCAV mission plans. This chapter provides a description of the implementation of Kirk, a planner that brings together these research contributions, and summarizes results of applying the planner to several mission scenarios.

Kirk makes significant progress towards the goal of applying model-based programming techniques to the problem of planning for coordinated air vehicle mission planning. However, many issues remain to be explored. Therefore, this chapter points the reader to a number of interesting and worthwhile research and implementation issues to be addressed in future work.

Results

Planner Implementation

Kirk consists of three main functional modules as shown in Figure 5.1. The *Plan Manager* performs the planning and related tasks, the *Plan Runner* takes a plan produced by the Plan Manager and executes it in a simulated environment, and the *AML Compiler*, which has not yet been implemented, is supposed to read in AML description files and compile AML activity descriptions into a set of TPN specification files.

The *PlanNet* data structure was implemented to represent the TPN models that were described in Chapter 3. This data structure maintains and supports the insertion, removal, and access of temporal events, temporal constraints, and symbolic constraints. In addition, it supports operations for testing temporal consistency, computing feasible time bounds for each temporal event, and identifying symbolic constraint incompatibilities. Finally, it is augmented with methods that allow it to save and restore planning state.

The Plan Manager is responsible for performing the tasks of the planning algorithm described in Chapter 4. It can be used to construct a PlanNet object from a Temporal Planning Network specification file that describes a mission scenario, and apply the planning algorithm to this network to generate a plan, which is then passed to the Plan

Runner. The Plan Manager also maintains additional information, such as the list of macro-activities, necessary to support the functions of the planning algorithm.

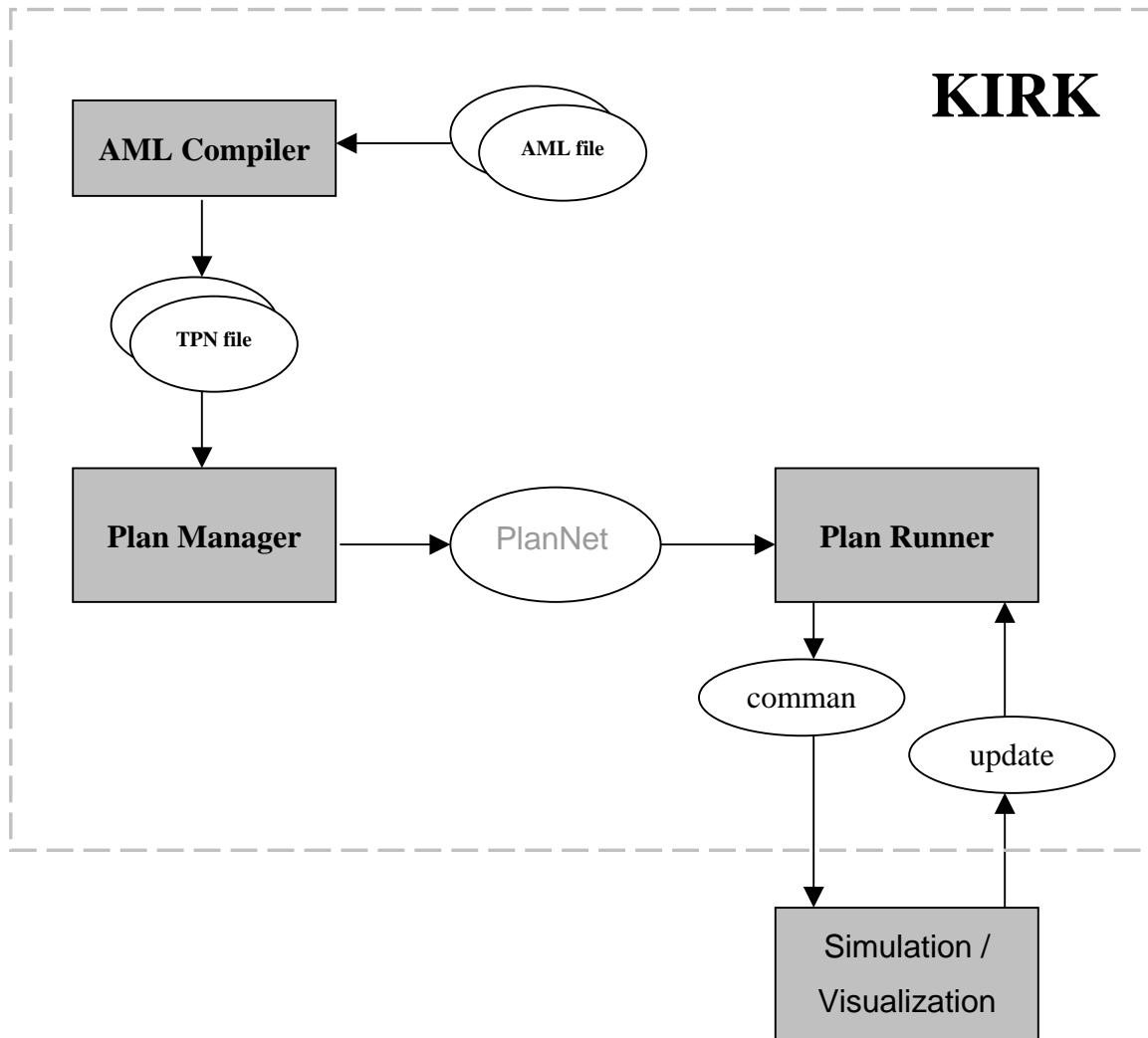


Figure 5.1 Block diagram of the Kirk planning system

The Plan Runner is used to execute a plan generated by the Plan Manager. In order to do this, the Plan Runner interfaces with a multiple vehicle dynamics simulation and visualization system, which is used to simulate the state and behavior of a group of vehicles. The Plan Runner relies on a basic implementation of the STN-plan dispatching algorithm that was used by HSTS [19] for plan execution. The Plan Runner generates commands corresponding to the vehicle execution primitives in the plan, and performs incremental updates to ensure that execution is consistent with the plan.

Kirk is implemented in ISO/ANSI compliant C++, with the exception of a small portion of the Plan Runner implementation responsible for communicating with the Simulation

and Visualization module. Currently, this communication relies on non-standard message passing libraries.

Performance

Kirk takes as input an activity instance and a lower and upper duration bound. For example, `Group-Rendezvous(1000,2000,1000)[130,180]` would be a valid input. For testing, Kirk was used to generate plans for various activities in the nominal case. Given more time, it would have been better to construct more complex test scenarios that included additional externally imposed constraints on activities.

The primary activity used for testing Kirk was the `Group-Sead` activity, which was based on descriptions of current manned combat air vehicle SEAD missions [27]. This activity was designed to model the mission described in section 1.2.2. The AML description of the `Group-Sead` activity is included, along with the other activity descriptions, in Appendix A. In the absence of the AML Compiler, these were compiled by hand into TPN files. The TPN specification file format is described in Appendix B, along with the actual TPN specification for an example scenario. The fully expanded TPN generated from the primary SEAD test case included 273 nodes. A planner output dump that lists the states of each of these nodes after planning is provided in Appendix C.

Figure 5.2 summarizes some quantitative performance results of having Kirk generate nominal plans for several different activities. The testing platform was an IBM Aptiva E6U with an Intel 400Mhz Pentium II processor and 128MB of RAM, running Redhat Linux version 6.1.

Activity Instance	Number of Nodes	Number of Activities	Time for Planning
<code>Group-Sead()</code>	273	47	404 s
<code>Group-Enroute()</code>	112	19	16 s
<code>Group-Attack(...)</code>	27	8	235 ms
<code>Follow(...)</code>	4	1	4 ms

Figure 5.2 Summary of Kirk's runtime performance on several test cases

The Activity Instance refers to the top-level activity that was being planned. The Number of Nodes is the size of the expanded TPN after planning. Usually, about half of

these were included in the final plan, with the rest corresponding to unselected executions. The Number of Activities indicates the number of primitive activities included in the final plan. Finally, the Time for Planning gives the time that it took for Kirk to generate a plan corresponding to each of these activities.

The time required for planning in many cases was heavily dominated by the time required for phase two of the planning algorithm, and in particular the computation of feasible time bounds for events. Section 5.2.3 addresses this issue by outlining some suggestions for future optimizations of the algorithm to avoid this performance degradation.

Future Work

This thesis has laid the groundwork for a variety of interesting future research. This section describes some of the open research issues and some suggested extensions of the planner. These suggestions for future work fall into three main categories. The first category includes some ideas for making the planner more robust. The second describes some potential limitations of AML and TPN and suggests ways to correct and improve the activity models. The third category describes some methods for further improving the efficiency of the planning algorithm.

Handling Contingencies

One way that planners can be more robust is by planning for all, or at least many, possible contingencies [17]. Contingent planners, also referred to as conditional planners, plan for different contingencies by keeping track of different plans for each possible combination of uncontrollable events. At the time of plan execution, the agents can query the state of the exogenous event to decide which plan should be executed. The planner described in this thesis is able to support the description of contingencies, but the current version of the planning algorithm does not generate contingent plans. In the context of this planner, contingent plans can be encoded along with the nominal mission plan using the choice operator that represent several different executions, each conditioned on a possible state of some exogenous factor, as shown in Figure 5.3. In this example, depending on the number of enemy targets detected, the group performs a different type of attack. In order to fully support contingency planning, it will be necessary to research the issues of accurately modeling sensing actions and incorporating these actions into the mission plans.

```
choose {
  { if target_count=1 then Group-Attack(target1) },
  { if target_count=2 then Group-Split-
Attack(target1,target2) }
}
```

Figure 5.3 Example of how the AML can represent contingent executions

One shortcoming of the Simple Temporal Network plan representation used by this planner is that it assumes that activity durations are controllable to the degree that it is possible to execute each activity of the plan within the duration bounds specified by the

plan. Although this provides more flexibility than a plan that predetermines the start times for every activity, it still may not adequately model the uncertainty of activity execution, especially for activities with a large variance in duration. There has been recent research exploring how to handle this type of execution uncertainty both at plan-time [20] and at execution-time [11], in the context of temporal planning with the STN. Since the Temporal Planning Network is a direct extension of the Simple Temporal Network, it should be possible to extend these methods to further enhance the robustness of this planner.

Another way to increase the robustness of this planner is to incorporate the methods of continuous planning and iterative plan repair that can be used to resolve failures in the plan that arise during plan execution. This method of fast, incremental repair is used by CASPER [3] to increase the reactivity of spacecraft to unexpected events, and it seems plausible to support fast plan repair given the ability of this planner to very quickly (less than a second) generate plans for very high-level activities of the mission. The next step is to examine the issue of how to quickly and smoothly modify the current plan or transition to a new plan during plan execution. Note, this poses an interesting problem because it isn't possible to put the air vehicles into a safe-mode while performing the transition to the updated plan, especially if the vehicles are in hostile territory.

Improving the Activity Models

The correctness of a plan generated by any planner depends on the correctness of the activity models on which it relies. This section describes some open representational issues of AML and TPN, and describes improvements to the activity models described in this thesis that may be explored in the future.

At the level of abstraction at which activities were modeled for this planner, it was sufficient to encode the transitions between activities, for example, when two activities were composed serially, using zero-duration temporal constraints. The effect of these temporal constraints was to constrain the start time of the second activity to be the same as the end time of the first. The problem with this representation is that in any real system that is executing activities that are not fully controllable, there will always be a delay between the completion of one activity and the start of the next. The reason for this is that it requires some time, however miniscule, for the system to process that the first activity has completed and to issue the command for the second to begin. By not modeling this latency in the plan, it is possible for even small execution delays to accumulate and cause activities to run past their latest allowed completion times, as discovered by Muscettola, et al. [12]. Therefore, future research is necessary to explore first whether it is necessary to model this latency accumulation in the context of this planner, and second what implications this has on the TPN activity model encoding. If it is found that zero-duration temporal constraints do not accurately model activity behavior, then these may be replaced with the estimated bounds on system latency.

Another representational issue that was mentioned in Chapter 3 was the interpretation of the direction of arcs in the Temporal Planning Network. Currently, the arcs signify both temporal constraints and also precedence relations, which allow for directed paths

through the network to represent a thread of execution composed of chronologically ordered temporal events. However, the interpretation of the arc as both a temporal constraint and a precedence relation breaks down when negative temporal constraints are allowed, which might be useful for future activity modeling. If this is the case, then this issue of the interpretation of arc direction must be revisited.

One problem with the activity models used by this planner is that the models may not accurately represent the duration bounds of their respective activities. For example, the duration of any instance of a Fly-To activity should be bounded roughly as $[d/\max v, d/\min v]$, where d =distance to the destination, $\min v$ =minimum velocity of the vehicle, and $\max v$ =maximum velocity of the vehicle. However, the current incarnation of the Fly-To activity model does not impose such bounds. The compile-time computation of activity duration bounds based on system limitations would enhance the correctness of the activity, so it is certainly worth investigating in the future. In addition, to support onboard replanning, it might be worthwhile to explore having the ability to estimate activity duration bounding as an online capability.

Finally, the TPN activity model encoding does not currently support the representation of post-conditions. While generative planners that use STRIPS activity models rely on pre- and post-conditions for constructing valid executions, the TPN activity models already encode these executions, so post-conditions are not necessary for this purpose. However, post-conditions are important because they specify the correct behavior of activities, which is critical for execution monitoring. Therefore, it may be useful to augment the TPN representation to support this in the future.

Optimizing the Planner

This section describes some potential optimizations that should be considered for future versions or implementations of the planner. One of the qualitative observations on the performance of the planner during testing was that the bottleneck operation seemed to be the computation of feasible time bounds for all temporal events at the beginning of Phase Two of the planning algorithm. This implies that if it is possible to speed up this operation, then this may significantly reduce the time it takes to generate a plan. Fortunately, several potentially powerful optimizations may be applied.

Recall that computing the feasible time bounds for all temporal events is done by solving an all-pairs shortest path problem. The current implementation of the planning algorithm uses the Floyd-Warshall algorithm for this, but as mentioned in section 4.3.2, the same problem is solved by Johnson's all-pairs shortest paths algorithm, which has better asymptotic running time in sparse networks. Since the TPN activity networks are typically sparse, using Johnson's algorithm may significantly improve the planner's running time.

In the current implementation of the planning algorithm, after a macro-activity is expanded and merged into the partial plan, the planner computes the feasible time bounds of all temporal events. This is more work than necessary in many cases because it may be possible to compute the time bounds of the macro-activity with respect to its start- and

end-nodes, and then separately re-compute the time bounds of the rest of the events in the rest of the plan. This too may result in a significant improvement in planner performance, especially for iterations of the planning algorithm in which large macro-activities (one with many nodes) are expanded.

There is a simple method of contracting Simple Temporal Network plans, used by HSTS [12], which may also be applied to Temporal Planning Networks. The contraction merges all nodes that correspond to the same point in time (i.e. those connected by zero-duration temporal constraints) into a single node, and reasons about all of them together. This is very useful for when planning for activities whose models contain many zero-duration temporal constraints.

Finally, the current implementation of the planning algorithm relies on an interval set data structure for maintaining symbolic constraints of the Planning Network. This data structure is used for detecting overlapping intervals for detecting symbolic constraint incompatibilities and for identifying possible ways to cover open conditions. Replacing the interval set with an interval tree data structure [5] may improve running time in practice.

Summary

Although there remain many issues to be considered by future work, the research described in this thesis takes several steps toward the goal of applying model-based programming techniques to the problem of planning for coordinated vehicle missions. The Activity Modeling Language addresses the challenge of developing activity models by providing a natural and expressive means of describing complex coordinated activities. The Temporal Planning Network serves as an encoding of the activity models that addresses the issue of compactness and, along with the planning algorithm presented in Chapter 4, addresses the challenge of efficient planning. These contributions were brought together and implemented in the Kirk planning system, but future research will be required to more completely develop the ideas presented in this thesis.

References

- [1] R. Ahuja, T. Magnanti, J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [2] A.L. Blum, M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281-300, 1997.
- [3] S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau. Using Iterative Repair to Increase the Responsiveness of Planning and Scheduling for Autonomous Spacecraft. *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS2000)*, Breckenridge, CO, April 2000.
- [4] L. Console, C. Picardi, M. Ribaudo. Diagnosis and diagnosibility analysis using Process Algebras. *Proc. 11th International Workshop On Principles of Diagnosis*, Morelia, Mexico, June 2000.
- [5] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.
- [6] T. Dean, B. McDermott. Temporal Database Management. *Artificial Intelligence*, 1-56, 1987.
- [7] R. Dechter, I. Meiri, J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61-95, May 1991.
- [8] 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.
- [9] V. Gupta, R. Jagadeesan, V. Saraswat. Models for Concurrent Constraint Programming. *Proc. of CONCUR'96: Concurrency Theory*, edited by Ugo Montanari and Vladimiro Sassone, LNCS 1119, Springer Verlag, 1996.
- [10] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231-274, 1987.
- [11] P. Morris, N. Muscettola. Execution of Temporal Plans with Uncertainty. *Proc. 16th National Conference on Artificial Intelligence (AAAI-99)*, Orlando, FL, 1999.
- [12] N. Muscettola, P. Morris, B. Pell, B. Smith. Issues in Temporal Reasoning for Autonomous Control Systems. *Proc. 2nd International Conference on Autonomous Agents*, Minneapolis, MI, 1998.
- [13] N. Muscettola, B. Smith, S. Chien, C. Fry, G. Rabideau, K. Rajan, D. Yan. On-board planning for autonomous spacecraft. *Proc. 4th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (ISAIRAS)*, July 1997.

- [14] N. Muscettola, P.P. Nayak, B. Pell, B. Williams. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5-48, August 1998.
- [15] J. Penberthy, D. Weld. Temporal planning with continuous change. *Proc. 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, 1994.
- [16] 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.
- [17] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
- [18] D. Smith, J. Frank, A.K. Jonsson. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review*, Volume 15, Number 1, 2000.
- [19] I. Tsamardinos, N. Muscettola, P. Morris. Fast transformation of temporal plans for efficient execution. *Proc. 15th National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, 1998.
- [20] T. Vidal. Dealing with Temporal Uncertainty and Reactivity in a space mission plan. *Proc. 2nd NASA International Workshop on Planning and Scheduling for Space*.
- [21] D. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 27-61, Winter 1994.
- [22] D. Weld. Recent Advance in AI Planning. *AI Magazine*, 93-123, Spring 1999.
- [23] B.C. Williams, P.P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. *Proc. 13th National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, 1996.
- [24] B.C. Williams. Doing Time: Putting Qualitative Reasoning on Firmer Ground. *Proc. 5th National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, August 1986.
- [25] B.C. Williams, V. Gupta. Unifying Model-based and Reactive Programming in a Model-based Executive. *Proc. 10th International Workshop on Principles of Diagnosis*, Scotland, June 1999.

- [26] S. Wolfman, D. Weld. Temporal Planning with Mutual Exclusion Reasoning.
Proc. 16th International Joint Conference on Artificial Intelligence.
- [27] "Hierarchical Decomposition of Autonomy Requirements for Naval UCAVs", for the Uninhabited Combat Air Vehicle Demonstrations, prepared for Office of Naval Research (under subcontract to Scientific Systems Company Incorporated), Charles Stark Draper Laboratory Document Control #387031, July, 2000.

Appendix A

AML Activity Descriptions

```
//-----

Sead-Scenario()[l,u] {
    Group-Sead()[l*50%,u],
    { PATH1=ok }[700,700],
    { [300,300]; { PATH2=ok }[500,500] }
}

Group-Sead()[l,u] := {
    Group-Enroute()[l*40%,u*40%];
    Group-Engage()[l*20%,u*20%];
    Group-Return()[l*40%,u*40%];
}

//-----

Group-Takeoff()[l,u] {
    {
        { ONE::Takeoff()[l*25%,u*25%]; [0,+INF] },
        { TWO::Takeoff()[l*25%,u*25%]; [0,+INF] }
    };
    Group-Rendezvous(RVPT)[l*75%,u*75%]
}

Group-Enroute()[l,u] := {
    choose {
        if PATH1=ok then {
            Group-Move-to(PATH1_1)[l*20%,u*20%];
            Group-Move-to(PATH1_2)[l*20%,u*20%];
            Group-Move-to(PATH1_3)[l*20%,u*20%];
            Group-Move-to(TAI)[l*20%,u*20%]
        },
        if PATH2=ok then {
            Group-Move-to(PATH2_1)[l*20%,u*20%];
            Group-Move-to(PATH2_2)[l*20%,u*20%];
            Group-Move-to(PATH2_3)[l*20%,u*20%];
            Group-Move-to(TAI)[l*20%,u*20%]
        }
    };
    Group-Xmit(FAC,GROUP_ARRIVED_TAI)[3,5];
    do {
        Group-Wait(TAI_HOLD1, TAI_HOLD2)[0,+INF]
    }
    watching GROUP::ENGAGE=ok
}

Group-Engage()[l,u] := {
    choose {
        if TARGET_COUNT=1 then {
            Group-Attack(T1_ENTRY_PT, T1_DROP_PT, T1_EXIT_PT, T1_POS)[l,u]
        },
        if TARGET_COUNT=2 then {
            { ONE::Attack(T1_ENTRY_PT, T1_DROP_PT, T1_EXIT_PT, T1_POS)[l,u];
              [0,+INF]
            },
            { TWO::Attack(T2_ENTRY_PT, T2_DROP_PT, T2_EXIT_PT, T2_POS)[l,u];
              [0,+INF]
            }
        },
        if TARGET_COUNT=2 then {
            { TWO::Attack(T1_ENTRY_PT, T1_DROP_PT, T1_EXIT_PT, T1_POS)[l,u];
              [0,+INF]
            },
        }
    }
}
```

```

        { ONE::Attack(T2_ENTRY_PT, T2_DROP_PT, T2_EXIT_PT, T2_POS)[1,u];
          [0,+INF]
        }
      },
      if TARGET_COUNT=2 then {
        Group-Attack(T1_ENTRY_PT, T1_DROP_PT, T1_EXIT_PT,
          T1_POS)[1*50%,u*50%];
        Group-Attack(T2_ENTRY_PT, T2_DROP_PT, T2_EXIT_PT,
          T2_POS)[1*50%,u*50%]
      },
      if TARGET_COUNT=2 then {
        Group-Attack(T2_ENTRY_PT, T2_DROP_PT, T2_EXIT_PT,
          T2_POS)[1*50%,u*50%];
        Group-Attack(T1_ENTRY_PT, T1_DROP_PT, T1_EXIT_PT,
          T1_POS)[1*50%,u*50%]
      },
      if TARGET_COUNT=2 then {
        Group-Attack(T1_ENTRY_PT, T1_DROP_PT, T1_EXIT_PT, T1_POS)[1,u]
      },
      if TARGET_COUNT=2 then {
        Group-Attack(T2_ENTRY_PT, T2_DROP_PT, T2_EXIT_PT, T2_POS)[1,u];
      }
    }
  }

Group-Return()[1,u] := {
  Group-Rendezvous(TAI);
  choose {
    if PATH1=ok then {
      Group-Move-to(PATH1_3)[1*25%,u*25%];
      Group-Move-to(PATH1_2)[1*25%,u*25%];
      Group-Move-to(PATH1_1)[1*25%,u*25%];
    },
    if PATH2=ok then {
      Group-Move-to(PATH2_3)[1*25%,u*25%];
      Group-Move-to(PATH2_2)[1*25%,u*25%];
      Group-Move-to(PATH2_1)[1*25%,u*25%];
    }
  }
};

{ ONE::Move-to(HOME1)[1*25%,u*25%],
  TWO::Move-to(HOME2)[1*25%,u*25%]
}

}

Group-Land()[1,u] {
  ONE::Land()[1,u],
  TWO::Land()[1,u]
}

//-----

Group-Rendezvous(RVPT)[1,u] := {
  { ONE::Move-to(RVPT)[1,u*90%];
    ONE::Xmit(ALL,ONE_RVPT_ARRIVED)[0,6];
    do { ONE::Hold(RVPT_HOLD1)[0,+U*5%],
        ONE::Listen()[0,U*5%]
    } watching TWO_RVPT_ARRIVED
  },
  { TWO::Move-to(RVPT)[1,u*90%];
    TWO::Xmit(ALL,TWO_RVPT_ARRIVED)[0,6];
    do { TWO::Hold(RVPT_HOLD1)[0,+U*5%],
        TWO::Listen()[0,U*5%]
    } watching ONE_RVPT_ARRIVED
  }
}

Group-Move-to(X,Y,Z)[1,u] := {
  choose {
    if NOT(ONE::NAV=DAMAGED) then {
      ONE::Move-to(X,Y,Z)[1,u],

```

```

        TWO::Follow(ONE)
        TWO::Listen[l,u]
    },
    if NOT(TWO::NAV=DAMAGED) then {
        TWO::Move-to(X,Y,Z)[l,u],
        ONE::Follow(TWO)[l,u],
        ONE::Listen[l,u]
    }
}
}

Group-Xmit(Target,Message)[l,u] := {
    choose {
        if NOT(ONE::COMM=DAMAGED) then ONE::Xmit(Target,Message)[l,u],
        if NOT(TWO::COMM=DAMAGED) then TWO::Xmit(Target,Message)[l,u]
    }
}

Group-Attack(Entry_Pt, Drop_Pt, Exit_Pt, Target_Pos)[l,u] := {
    choose {
        if ONE::BOMB=ok then {
            ONE::Attack(Entry_Pt, Drop_Pt, Exit_Pt, Target_Pos)[l,u],
            TWO::Follow(ONE_ID)[l,u],
            TWO::Sense()[l,u]
        },
        if TWO::BOMB=ok then {
            TWO::Attack(Entry_Pt, Drop_Pt, Exit_Pt, Target_Pos)[l,u],
            ONE::Follow(TWO_ID)[l,u],
            ONE::Sense()[l,u]
        }
    }
}

Group-Wait(Hold1, Hold2)[l,u] := {
    ONE::Wait(Hold1)[l,u],
    TWO::Wait(Hold2)[l,u]
}

//-----

Wait(Pt1, Pt2, Pt3)[l,u] := {
    repeat {
        Move-to(Pt1)[20,30];
        Move-to(Pt2)[20,30];
        Move-to(Pt3)[20,30];
    }[l,u]
}

Attack(Entry_Pt, Drop_Pt, Exit_Pt, Target_Pos)[l,u] := {
    Move-to(Entry_Pt)[1*30%,u*30%];
    Move-to(Drop_Pt)[1*30%,u*30%];
    { { Bomb-at(Target_Pos)[20,30]; [0,+INF] },
      Move-to(Exit_Pt)[1*30%,u*30%]
    }
}

//-----

Move-to(X,Y,Z)[l,u] := {
    {ID::DST=set}[l,u]
}

Bomb-at(X,Y)[l,u] := {
    if ID::BOMB=ok then {
        Target()[l,u]
    }
}

Follow(Target)[l,u] := {
    Listen()[l,u],
    Sense()[l,u],

```

```

    ID::DST_SET[l,u]
}

Xmit(Target,)[l,u] := {
    choose {
        {ch1=ID}[l,u],
        {ch2=ID}[l,u]
    }
}

Listen()[l,u] := {
    NOT(ch1=ID)[l,u],
    NOT(ch2=ID)[l,u]
}

Sense()[l,u] := {
    {ID::Sensor=sense}[l,u]
}

Target()[l,u] := {
    {ID::Sensor=target}[l,u]
}

Takeoff()[l,u] := {
    {ID::DST=set}[l,u]
}

Land()[l,u] := {
    {ID::DST=set}[l,u]
}

```


Appendix B

TPN specification format

```

TPN file := argument_name*
          node_count
          node_data*
          arc_data*
          -1 -1
          symbolic_constraint_data*

```

Each TPN file begins with a list of zero or more argument names (argument_name*). This is followed by an integer number of nodes in the network (node_count). For each node in the network, there must be a corresponding node description in the node_data format. Following the node descriptions must be a list of arc descriptions in the arc_data format. For each arc in the Temporal Planning Network described by the file, there is one forward and one backward arc in this list of arcs, corresponding to the two arcs in its distance graph representation. The list of arc descriptions is terminated by "-1 -1". Finally, all the symbolic constraints in the network are listed.

```

node_data := decision_node?
           node_name
           activity?
           start_node?
           *

```

Each node description consists of four pieces of information delimited by whitespace. First is a flag that indicates whether the node is a decision node (0=no, 1=yes). Second is the node name. Third is a flag that indicates whether the node corresponds to either the start or end event of an activity (0=no, 1=yes). Fourth is a flag that is checked only if the activity? flag is 1. It indicates whether the node corresponds to the start or the activity (0=end node, 1=start node). All node descriptions are terminated by an asterisk.

```

arc_data := head_node_index
           tail_node_index
           forward_arc?
           distance
           *

```

Each arc description begins with the indices of the head and tail nodes of the arc. This is followed by a flag that indicates whether the arc is a forward arc (0=backward arc, 1=forward arc). After this is the distance associated with the arc, which can be an integral value, positive or negative infinity, or a relative value. Arc descriptions are also terminated by an asterisk.

```

symbolic_constraint_data :=
  head_node_index
  tail_node_index
  proposition
  type
  *

```

Each symbolic constraint description also begins with head and tail node indices that specify with which arc the symbolic constraint is associated. Next is the proposition of the symbolic constraint, followed by a type which is one of the following: ASK, TELL, ASK_NOT, TELL_NOT. Each symbolic constraint description is terminated by an asterisk.

An example of a TPN specification file follows.

Sead-Scenario.tpn

No arguments.

12

node_count = 12

0 Group-Sead()	1 1 *	<i>12 node descriptions</i>
0 Group-Sead()	1 0 *	.
0 Group-Enroute()	1 1 *	.
0 Group-Enroute()	1 0 *	.
0 Group-Engage()	1 1 *	.
0 Group-Engage()	1 0 *	.
0 Group-Return()	1 1 *	.
0 Group-Return()	1 0 *	.
0 PATH1_begin	0 0 *	.
0 PATH1_end	0 0 *	.
0 PATH2_begin	0 0 *	.
0 PATH2_end	0 0 *	.

0 1 1 +U *	<i>Arc descriptions</i>
0 2 1 +0 *	.
0 8 1 +0 *	.
0 10 1 +300 *	.
1 0 0 -L *	.
1 7 0 -0 *	.
2 0 0 -0 *	.
2 3 1 +U*40% *	.
3 2 0 -L*35% *	.
3 4 1 +0 *	.
4 3 0 -0 *	.
4 5 1 +U*20% *	.
5 4 0 -L*20% *	.
5 6 1 +0 *	.
6 5 0 -0 *	.
6 7 1 +U*40% *	.
7 1 0 -0 *	.
7 6 0 -L*35% *	.
8 0 0 -0 *	.
8 9 1 +700 *	.
9 8 0 -700 *	.
10 0 0 -300 *	.
10 11 1 +500 *	.
11 10 0 -500 *	.
-1 -1	<i>Arc data terminator</i>

8 9	PATH1=OK TELL *	<i>Symbolic constraint descriptions</i>
10 11	PATH2=OK TELL *	.

Appendix C

Raw Output Dump: Sead-Scenario()[3000,3600]

```
plan node[0]: {Group-Sead():0,0} decomposed?=1 isactivity?=1 isstart?=1
plan node[1]: {Group-Sead():3000,3600} decomposed?=0 isactivity?=1 isstart?=0
plan node[2]: {Group-Enroute():0,0} decomposed?=1 isactivity?=1 isstart?=1
plan node[3]: {Group-Enroute():1108,1440} decomposed?=1 isactivity?=1 isstart?=0
plan node[4]: {Group-Engage():1108,1440} decomposed?=1 isactivity?=1 isstart?=1
plan node[5]: {Group-Engage():1708,2088} decomposed?=1 isactivity?=1 isstart?=0
plan node[6]: {Group-Return():1708,2088} decomposed?=1 isactivity?=1 isstart?=1
plan node[7]: {Group-Return():2816,3528} decomposed?=1 isactivity?=1 isstart?=0
plan node[8]: {PATH1_begin():0,0} decomposed?=0 isactivity?=0 isstart?=0
plan node[9]: {PATH1_end():700,700} decomposed?=0 isactivity?=0 isstart?=0
plan node[10]: {PATH2_begin():300,300} decomposed?=0 isactivity?=0 isstart?=0
plan node[11]: {PATH2_end():800,800} decomposed?=0 isactivity?=0 isstart?=0
plan node[12]: {Decision-1():0,0} decomposed?=0 isactivity?=0 isstart?=0
plan node[13]: {Group-Move-to(18000,28000,5000):0,0} decomposed?=1 isactivity?=1 isstart?=1
plan node[14]: {Group-Move-to(18000,28000,5000):262,360} decomposed?=1 isactivity?=1 isstart?=0
plan node[15]: {Group-Move-to(25000,30000,6000):262,360} decomposed?=1 isactivity?=1 isstart?=1
plan node[16]: {Group-Move-to(25000,30000,6000):524,720} decomposed?=1 isactivity?=1 isstart?=0
plan node[17]: {Group-Move-to(32000,30000,5000):524,720} decomposed?=1 isactivity?=1 isstart?=1
plan node[18]: {Group-Move-to(32000,30000,5000):786,1080} decomposed?=1 isactivity?=1 isstart?=0
plan node[19]: {Group-Move-to(15000,5000,4000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[20]: {Group-Move-to(15000,5000,4000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[21]: {Group-Move-to(25000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[22]: {Group-Move-to(25000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[23]: {Group-Move-to(36000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[24]: {Group-Move-to(36000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[25]: {Group-Move-to(40000,25000,4000):786,1080} decomposed?=1 isactivity?=1 isstart?=1
plan node[26]: {Group-Move-to(40000,25000,4000):1048,1380} decomposed?=1 isactivity?=1 isstart?=0
plan node[27]: {Group-Xmit(FAC,ARRIVED_TAI):1048,1380} decomposed?=1 isactivity?=1 isstart?=1
plan node[28]: {Group-Xmit(FAC,ARRIVED_TAI):1048,1380} decomposed?=1 isactivity?=1 isstart?=0
plan node[29]: {Group-Wait(41000,26000,5000,41000,24000,5000,39000,25000,5000,41000,26000,6000,41000,24000,6000,39000,25000,6000):1048,1380}
decomposed?=1 isactivity?=1 isstart?=1
plan node[30]: {Group-Wait(41000,26000,5000,41000,24000,5000,39000,25000,5000,41000,26000,6000,41000,24000,6000,39000,25000,6000):1108,1440}
decomposed?=1 isactivity?=1 isstart?=0
plan node[31]: {Intermediate-1():0,0} decomposed?=0 isactivity?=0 isstart?=0
plan node[32]: {Intermediate-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[33]: {Decision-1():1108,1440} decomposed?=0 isactivity?=0 isstart?=0
plan node[34]: {Group-Attack(45000,22000,4000,49500,20500,3000,50000,25000,4000,50000,20000):1108,1440} decomposed?=1 isactivity?=1 isstart?=1
plan node[35]: {Group-Attack(45000,22000,4000,49500,20500,3000,50000,25000,4000,50000,20000):1708,2088} decomposed?=1 isactivity?=1 isstart?=0
plan node[36]: {Choice1-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[37]: {ONE:Attack(45000,22000,4000,49500,20500,3000,50000,25000,4000,50000,20000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[38]: {ONE:Attack(45000,22000,4000,49500,20500,3000,50000,25000,4000,50000,20000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[39]: {TWO:Attack(45000,22000,4000,49500,15500,3000,45000,15000,4000,50000,15000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[40]: {TWO:Attack(45000,22000,4000,49500,15500,3000,45000,15000,4000,50000,15000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[41]: {Group-Attack(45000,22000,4000,49500,20500,3000,50000,25000,4000,50000,20000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[42]: {Group-Attack(45000,22000,4000,49500,20500,3000,50000,25000,4000,50000,20000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[43]: {Group-Attack(45000,22000,4000,49500,15500,3000,45000,15000,4000,50000,15000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[44]: {Group-Attack(45000,22000,4000,49500,15500,3000,45000,15000,4000,50000,15000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[45]: {OR-1():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[46]: {OR-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[47]: {Decision-1():1708,2088} decomposed?=0 isactivity?=0 isstart?=0
plan node[48]: {Group-Move-to(18000,28000,5000):1708,2088} decomposed?=1 isactivity?=1 isstart?=1
plan node[49]: {Group-Move-to(18000,28000,5000):1970,2448} decomposed?=0 isactivity?=1 isstart?=0
plan node[50]: {Group-Move-to(25000,30000,6000):1970,2448} decomposed?=1 isactivity?=1 isstart?=1
plan node[51]: {Group-Move-to(25000,30000,6000):2232,2808} decomposed?=0 isactivity?=1 isstart?=0
plan node[52]: {Group-Move-to(18000,28000,5000):2232,2808} decomposed?=1 isactivity?=1 isstart?=1
plan node[53]: {Group-Move-to(18000,28000,5000):2494,3168} decomposed?=0 isactivity?=1 isstart?=0
plan node[54]: {Group-Move-to(36000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[55]: {Group-Move-to(36000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[56]: {Group-Move-to(25000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[57]: {Group-Move-to(25000,4000,4500):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[58]: {Group-Move-to(15000,5000,4000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[59]: {Group-Move-to(15000,5000,4000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[60]: {Group-Move-to(10000,22500,4000):2494,3168} decomposed?=1 isactivity?=1 isstart?=1
plan node[61]: {Group-Move-to(10000,22500,4000):2756,3468} decomposed?=1 isactivity?=1 isstart?=0
plan node[62]: {Group-Xmit(ATC,ARRIVED_HOME):2756,3468} decomposed?=1 isactivity?=1 isstart?=1
plan node[63]: {Group-Xmit(ATC,ARRIVED_HOME):2756,3468} decomposed?=1 isactivity?=1 isstart?=0
plan node[64]: {Group-Wait(11000,21500,5000,11000,23500,5000,9000,22500,5000,11000,21500,6000,11000,23500,6000,9000,22500,6000):2756,3468}
decomposed?=1 isactivity?=1 isstart?=1
plan node[65]: {Group-Wait(11000,21500,5000,11000,23500,5000,9000,22500,5000,11000,21500,6000,11000,23500,6000,9000,22500,6000):2816,3528}
decomposed?=1 isactivity?=1 isstart?=0
plan node[66]: {Intermediate-1():1708,2088} decomposed?=0 isactivity?=0 isstart?=0
plan node[67]: {Intermediate-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[68]: {Decision-1():0,0} decomposed?=0 isactivity?=0 isstart?=0
plan node[69]: {Choice-1-1():0,0} decomposed?=0 isactivity?=0 isstart?=0
plan node[70]: {Choice-1-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[71]: {ONE:Move-to(18000,28000,5000):0,0} decomposed?=1 isactivity?=1 isstart?=1
plan node[72]: {ONE:Move-to(18000,28000,5000):262,360} decomposed?=1 isactivity?=1 isstart?=0
plan node[73]: {TWO:Follow(ONE):0,0} decomposed?=1 isactivity?=1 isstart?=1
plan node[74]: {TWO:Follow(ONE):262,360} decomposed?=1 isactivity?=1 isstart?=0
plan node[75]: {TWO:Move-to(18000,28000,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[76]: {TWO:Move-to(18000,28000,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[77]: {ONE:Follow(TWO):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[78]: {ONE:Follow(TWO):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[79]: {Decision-1():262,360} decomposed?=0 isactivity?=0 isstart?=0
plan node[80]: {Choice-1-1():262,360} decomposed?=0 isactivity?=0 isstart?=0
plan node[81]: {Choice-1-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[82]: {ONE:Move-to(25000,30000,6000):262,360} decomposed?=1 isactivity?=1 isstart?=1
plan node[83]: {ONE:Move-to(25000,30000,6000):524,720} decomposed?=1 isactivity?=1 isstart?=0
plan node[84]: {TWO:Follow(ONE):262,360} decomposed?=1 isactivity?=1 isstart?=1
plan node[85]: {TWO:Follow(ONE):524,720} decomposed?=1 isactivity?=1 isstart?=0
plan node[86]: {TWO:Move-to(25000,30000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[87]: {TWO:Move-to(25000,30000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[88]: {ONE:Follow(TWO):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[89]: {ONE:Follow(TWO):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[90]: {Decision-1():524,720} decomposed?=0 isactivity?=0 isstart?=0
plan node[91]: {Choice-1-1():524,720} decomposed?=0 isactivity?=0 isstart?=0
plan node[92]: {Choice-1-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[93]: {ONE:Move-to(32000,30000,5000):524,720} decomposed?=1 isactivity?=1 isstart?=1
plan node[94]: {ONE:Move-to(32000,30000,5000):786,1080} decomposed?=1 isactivity?=1 isstart?=0
```

[illegible]

```

plan node[209]: {ONE::Move-to(41000,24000,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[210]: {ONE::Move-to(41000,24000,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[211]: {ONE::Move-to(39000,22500,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[212]: {ONE::Move-to(39000,22500,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[213]: {TWO::Decision-1():1048,1380} decomposed?=0 isactivity?=0 isstart?=0
plan node[214]: {TWO::Move-to(41000,26000,6000):1048,1380} decomposed?=1 isactivity?=1 isstart?=1
plan node[215]: {TWO::Move-to(41000,26000,6000):1068,1400} decomposed?=1 isactivity?=1 isstart?=0
plan node[216]: {TWO::Move-to(41000,24000,6000):1068,1400} decomposed?=1 isactivity?=1 isstart?=1
plan node[217]: {TWO::Move-to(41000,24000,6000):1088,1420} decomposed?=1 isactivity?=1 isstart?=0
plan node[218]: {TWO::Move-to(39000,25000,6000):1088,1420} decomposed?=1 isactivity?=1 isstart?=1
plan node[219]: {TWO::Move-to(39000,25000,6000):1108,1440} decomposed?=1 isactivity?=1 isstart?=0
plan node[220]: {TWO::Move-to(41000,26000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[221]: {TWO::Move-to(41000,26000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[222]: {TWO::Move-to(41000,24000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[223]: {TWO::Move-to(41000,24000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[224]: {TWO::Move-to(39000,25000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[225]: {TWO::Move-to(39000,25000,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[226]: {ONE::Move-to(45000,22000,4000):1108,1440} decomposed?=1 isactivity?=1 isstart?=1
plan node[227]: {ONE::Move-to(45000,22000,4000):1288,1656} decomposed?=1 isactivity?=1 isstart?=0
plan node[228]: {ONE::Move-to(49500,20500,3000):1288,1656} decomposed?=1 isactivity?=1 isstart?=1
plan node[229]: {ONE::Move-to(49500,20500,3000):1492,1872} decomposed?=1 isactivity?=1 isstart?=0
plan node[230]: {ONE::Bomb-at(50000,20000):1492,1872} decomposed?=1 isactivity?=1 isstart?=1
plan node[231]: {ONE::Bomb-at(50000,20000):1492,1892} decomposed?=1 isactivity?=1 isstart?=0
plan node[232]: {ONE::Move-to(50000,25000,4000):1492,1872} decomposed?=1 isactivity?=1 isstart?=1
plan node[233]: {ONE::Move-to(50000,25000,4000):1708,2088} decomposed?=1 isactivity?=1 isstart?=0
plan node[234]: {TWO::Listen():1108,1440} decomposed?=1 isactivity?=1 isstart?=1
plan node[235]: {TWO::Listen():1708,2088} decomposed?=1 isactivity?=1 isstart?=0
plan node[236]: {TWO::Listen():1708,2088} decomposed?=1 isactivity?=1 isstart?=1
plan node[237]: {TWO::Listen():1048,1380} decomposed?=1 isactivity?=1 isstart?=0
plan node[238]: {TWO::Listen():1970,2448} decomposed?=1 isactivity?=1 isstart?=1
plan node[239]: {TWO::Listen():262,360} decomposed?=1 isactivity?=1 isstart?=0
plan node[240]: {TWO::Listen():2232,2808} decomposed?=1 isactivity?=1 isstart?=1
plan node[241]: {TWO::Listen():524,720} decomposed?=1 isactivity?=1 isstart?=0
plan node[242]: {TWO::Listen():2494,3168} decomposed?=1 isactivity?=1 isstart?=1
plan node[243]: {TWO::Listen():262,360} decomposed?=0 isactivity?=1 isstart?=0
plan node[244]: {ONE::decision-1():2756,3468} decomposed?=0 isactivity?=0 isstart?=0
plan node[245]: {ONE::intermediate-1():2756,3468} decomposed?=0 isactivity?=0 isstart?=0
plan node[246]: {ONE::intermediate-2():-INF,+INF} decomposed?=0 isactivity?=0 isstart?=0
plan node[247]: {ONE::Decision-1():2756,3468} decomposed?=0 isactivity?=0 isstart?=0
plan node[248]: {ONE::Move-to(11000,21500,5000):2756,3468} decomposed?=1 isactivity?=1 isstart?=1
plan node[249]: {ONE::Move-to(11000,21500,5000):2776,3488} decomposed?=1 isactivity?=1 isstart?=0
plan node[250]: {ONE::Move-to(11000,24000,5000):2776,3488} decomposed?=1 isactivity?=1 isstart?=1
plan node[251]: {ONE::Move-to(11000,24000,5000):2796,3508} decomposed?=1 isactivity?=1 isstart?=0
plan node[252]: {ONE::Move-to(9000,22500,5000):2796,3508} decomposed?=1 isactivity?=1 isstart?=1
plan node[253]: {ONE::Move-to(9000,22500,5000):2816,3528} decomposed?=1 isactivity?=1 isstart?=0
plan node[254]: {ONE::Move-to(11000,21500,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[255]: {ONE::Move-to(11000,21500,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[256]: {ONE::Move-to(11000,24000,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[257]: {ONE::Move-to(11000,24000,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[258]: {ONE::Move-to(9000,22500,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[259]: {ONE::Move-to(9000,22500,5000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[260]: {TWO::Decision-1():2756,3468} decomposed?=0 isactivity?=0 isstart?=0
plan node[261]: {TWO::Move-to(11000,21500,6000):2756,3468} decomposed?=1 isactivity?=1 isstart?=1
plan node[262]: {TWO::Move-to(11000,21500,6000):2776,3488} decomposed?=1 isactivity?=1 isstart?=0
plan node[263]: {TWO::Move-to(11000,23500,6000):2776,3488} decomposed?=1 isactivity?=1 isstart?=1
plan node[264]: {TWO::Move-to(11000,23500,6000):2796,3508} decomposed?=1 isactivity?=1 isstart?=0
plan node[265]: {TWO::Move-to(9000,22500,6000):2796,3508} decomposed?=1 isactivity?=1 isstart?=1
plan node[266]: {TWO::Move-to(9000,22500,6000):2816,3528} decomposed?=1 isactivity?=1 isstart?=0
plan node[267]: {TWO::Move-to(11000,21500,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[268]: {TWO::Move-to(11000,21500,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[269]: {TWO::Move-to(11000,23500,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[270]: {TWO::Move-to(11000,23500,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[271]: {TWO::Move-to(9000,22500,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=1
plan node[272]: {TWO::Move-to(9000,22500,6000):-INF,+INF} decomposed?=0 isactivity?=1 isstart?=0
plan node[273]: {ONE::Have-Bomb?():1492,1872} decomposed?=0 isactivity?=0 isstart?=0

```