Managing Temporal Uncertainty Under Limited Communication:

A Formal Model of Tight and Loose Team Coordination

by

John L. Stedl

B.S. in Aeronautics and Astronautics Engineering
University of Illinois, Urbana-Champaign, 1996

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

Masters of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2004
[Sep... a...]

Author.....
Department of Aeronautics and Astronautics
August 19, 2004

Certified by.....
Brian C. Williams
Associate Professor
Thesis Supervisor

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

Managing Temporal Uncertainty Under Limited Communication: A Formal Model of Tight and Loose Team Coordination

by

John L. Stedl

September 8, 2004

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

In the future, groups of autonomous robots will cooperate in large networks in order to achieve a common goal. These multi-agent systems will need to be able to execute cooperative temporal plans in the presence of temporal uncertainty and communication limitations. The duration of many planned activities will not be under direct control of the robots. In addition, robots will often not be able to communicate during plan execution. In order for the robots to robustly execute a cooperative plan, they will need to guarantee that a successful execution strategy exists, and provide a means to reactively compensate for the uncertainty in real-time. This thesis presents a multi-agent executive that enables groups of distributed autonomous robots to dynamically schedule temporally flexible plans that contain both temporal uncertainty under communication limitations.

Previous work has presented controllability algorithms that compile the simple temporal networks with uncertainty, STNUs, into a form suitable for execution. This thesis extends the previous controllability algorithms to operate on two-layer plans that specify group level coordination at the highest level and agent level coordination at a lower level. We introduce a Hierarchical Reformulation (HR) algorithm that reformulates the two-layer plan in order to enable agents to dynamically adapt to uncertainty within each group plan and use a static execution strategy between groups in order to compensate for communication limitations. Formally, the HR algorithm ensures that the two-layer plan is strongly controllable at the highest level and *dynamically controllable* at the lower level. Furthermore, we introduce a new fast dynamic controllability algorithm that has been empirically shown to run in $O(N^3)$ time.

The Hierarchical Reformulation algorithm has been validated on a set of hand coded examples. The speed of the new fast dynamic controllability algorithm has been tested using a set of randomly generated problems.

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

Acknowledgements

If you are reading this then you should believe in miracles. The experience of writing a thesis is one in which ...

I would like to thank Pooja Rajaram for all the support and love that made this thesis possible. I would like to thank her for of the late nights she spent supporting me in the rover lab and for the encouragement to finish, as well as all of the late nights editing this thesis.

I would like to thank all the members in the MERS group. Specifically, Rob Ragno whose few words set forth the whole direction of my thesis. I would also like to thank Andreas Wehowsky, Samidh Chakrabarthi, Stano Funiak, and Aisha Walcott for their technical support and friendship. Furthermore, I would like to thank Brad Hasegawa for his help in formatting.

I would like to thank my advisor Brian Williams for supporting me through the entire project.

Finally, I would like to thank my entire family for their love and support.

This thesis also could not have been completed without the sponsorship of the DARPA NEST program under contract F33615-01-C-1896.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In the future, groups of distributed autonomous robots will need to cooperate in order to solve complex problems. In the not too distant future, teams of autonomous robots will be exploring the surface of Mars and aiding humans in the exploration of the planet. NASA is currently exploring tight coordination between two autonomous rovers [Huntsberger 2004]. Furthermore, NASA's Earth and Space Science Enterprises are planning the creation of sensor webs that will quickly and accurately characterize weather, fire, planetary eruptions, and other scientific events in real time. These sensor webs will consist of heterogeneous robots that range from Earth orbiting satellites to autonomous Unmanned Aerial Vehicles (UAVs). Observations will require the coordinated activities of multiple spacecraft that will be organized in constellations, space interferometers, telecommunication clusters, or other organizations of Earth and Deep Space observing systems. Proposed and planned NASA missions involving the coordination of multiple spacecraft include Earth Observing missions, such as COACH, Leonardo, Global Precipitation, ATOMS, and A-train and deep space missions, such as Terrestrial Planet Finder, Constellation–X, LISA and Starlight. NASA has identified both formation flying and autonomy as enabling technologies in its current vision of space exploration [Aldridge 2004]. On the terrestrial front, large groups of tiny wireless processors forming ad hoc networks will soon pervade the landscape, performing traffic, weather, or building monitoring.

In general, these autonomous multi-agent systems will be composed of a set of self-reliant agents that plan, sense, act, and communicate in an uncertain world, in order to achieve a common goal. Moreover, the agents will require cooperative plans that encode temporal synchronization, in order to achieve mission critical goals. Distributing the intelligence across several agents will make these systems more robust, efficient and adaptive. Spacecraft autonomy has successfully been demonstrated for single agent space systems such as Remote Agent [Muscettola 1999]; however, distributed space autonomy has yet to be realized. One of the major components required for realization of these complex distributed multi-agent systems is a distributed robust executive that enables a set of agents to collectively schedule coordinated tasks, while dynamically reacting to uncertainty under limited communication.

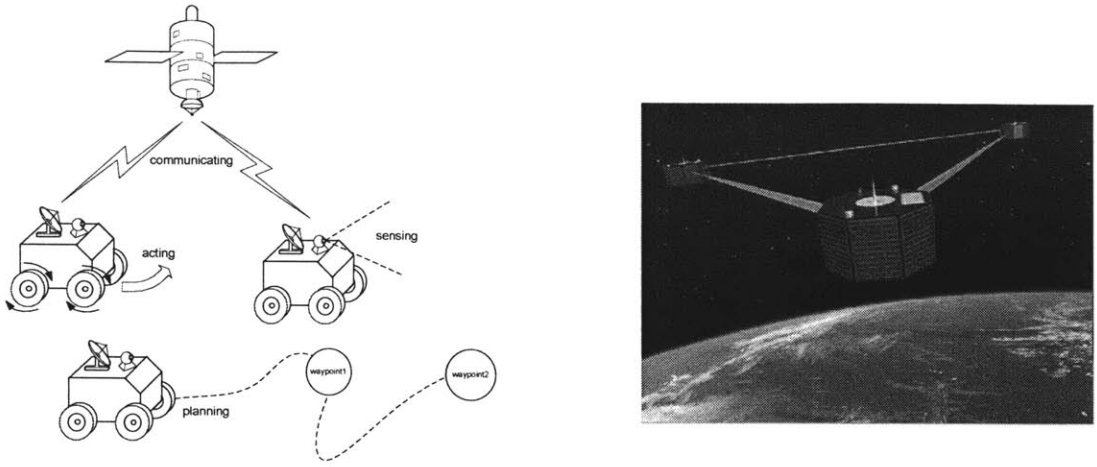


Figure 1-1 Examples of Future Multi-Agent Systems

(left) A group of autonomous rovers that will explore Mars, (right) a constellation of three satellites for NASA's proposed Terrestrial Planet Finder mission.

1.2 Distributed Multi-Agent Scenario

Consider a simple sensor web scenario that involves a cluster of low Earth orbiting satellites and a fleet of UAVs, in which the satellite observations guide the more detailed observations of the UAV fleet. The UAVs know the general location of the science targets; however, they require the imagery data from the satellites in order to identify specific science targets. In order to complete the mission, the satellite cluster and UAV fleet will need to loosely coordinate their activities. Specifically, in order for the UAVs to use the satellite imagery, the satellites must complete their imaging before the UAVs start their observations. In addition, the UAVs in the fleet and the satellites in the cluster will require tight coordination with one another. Some of the low level group behavior, such as station keeping or flying in formation, will be achieved by a purely reactive feed-back loop; however, other group behavior, such as coordinated observation, or reconfiguration, will require the agents to synchronize their actions through temporal planning. For example, the satellite cluster requires tight coordination in order to use interferometry, as shown in Figure 1-2. Interferometry is a technique which allows a cluster of satellites flying in a more or less uniform pattern to produce high resolution image using multiple apertures. Radar interferometry requires each satellite to illuminate the same area, "the footprint," simultaneously. An echo is received by each satellite and each portion of the image is constructed by pairwise interference of signals. The full image is then constructed by processing the data collected from all the satellites. This

entire process requires tight temporal coordination between the satellites. The satellites will also need to tightly coordinate their activities when the cluster reconfigures.

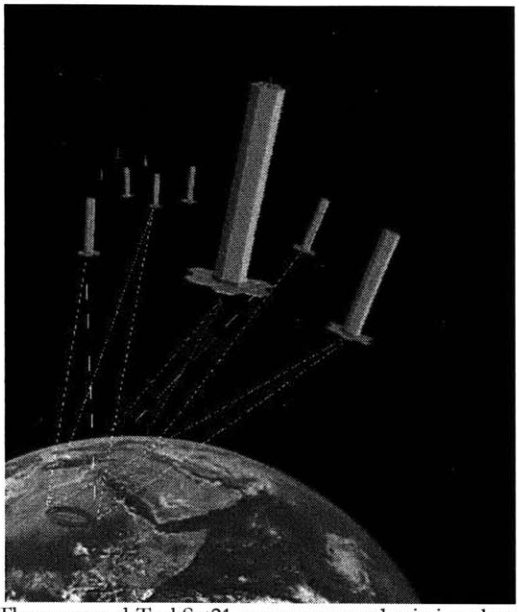


Figure 1-2 The proposed TechSat21 was a proposed mission that uses a cluster of small distributed satellites to perform space based sensing using interferometry. The satellites need to tightly coordinate their activities when imaging or reconfiguring the cluster.

The UAVs will also need to tightly coordinate their activities. For example, consider a scenario in which the UAVs start from a base station, fly to a science target, disperse to make individual observations, then rendezvous before returning to a base station as a group. In this scenario, the UAVs will need to synchronize their activities at a set of distinct timepoints.

In the above scenario, portions of the full cooperative plan will require tight coordination, for example inter-cluster or inter-fleet coordination, whereas, other portions of the plan will only require loose coordination. The term tight coordination refers to a portion of the plan where the activities are both heavily coupled (the execution time of one activity affects the feasible execution times of many other activities) and relativity inflexible, meaning the constraints between the activities are tight. At the extreme, tight coordination corresponds to bunches of activities being rigidly constrained. Loose coordination is the opposite of tight coordination; the activities are relatively decoupled and the temporal constraints that do exist between the activities are flexible.

1.3 Research Challenges

One major challenge is to enable the agents to robustly synchronize their activities in the face of uncertainty and limited communication. In order for multiple agents to synchronize their activities, they require a cooperative temporal plan, some means to determine the feasibility of the plan, and some means to execute that plan. [Dechter 1991] introduced a formalism, called *Simple Temporal Networks* (STNs), in order to model the temporal constraints between activities along with a centralized means to detect if the plan is temporally consistent. Given a consistent plan, the simplest approach to execute the plan is to generate a fixed schedule offline, then dispatch the tasks at the pre-scheduled time. However, this approach does not allow the executive to adapt the schedule in response to uncertain events at execution time. A fixed schedule is inflexible to uncertainty.

An alternative approach defers the scheduling to execution time. [Muscettola 1998a] showed how to efficiently perform this type of online execution in order to adapt to some amount of unmodeled uncertainty for single agent systems. In this approach, the plan retains temporal flexibility and the executive exploits this flexibility by scheduling the activities based on previous scheduling decisions. The temporal constraints of the plan are compiled (reformulated) offline in order to enable the executive to consistently and efficiently perform this online scheduling. However, this approach makes no guarantee about the ability to respond to uncertain events.

In order to ensure some level of robustness to uncertainty, the executive needs a model of uncertainty so that it can reason about its ability to respond to events of uncertain duration. For example, one of the most basic tasks for a mobile robot is to move from one location to another. The duration of this activity is uncertain. The robot is unable to predict the exact time it reaches its intended location because of wheel slippage. However, the plan may contain other activities that are constrained with respect to this uncertain outcome. These plans are called partially controllable because the agent only has control over the execution time of a subset of the events. In general, these uncertain events are either controlled by nature (as in the case of wheel slippage) or are under the control of some other agent. In order to be robust to this temporal uncertainty, the robot should determine (prior to execution) whether it can appropriately adjust the schedule of the plan in all possible situations. In some cases, where there is a large amount of flexibility in the plan, it is possible to statically schedule the controllable events; however, if tight coordination is required, the agent must dynamically adapt to the uncertainty.

[Vidal 1996] introduced a formalism, called *Simple Temporal Networks with Uncertainty* (STNU), which introduced a means to model temporal uncertainty within STNs. [Vidal 2000] introduced a *strong controllability* algorithm to determine if the plan contained enough temporal flexibility in order to fix the schedule without knowing the uncontrollable outcomes. In the cases where the plans are not strongly controllable, [Morris 2001] introduced a *dynamic controllability* algorithm, which enables the executive to determine if there is enough flexibility in the plan to compensate (at execution time) for the temporal uncertainty in the plan. This dynamic controllability

algorithm reformulates the plan into a *dispatchable* form, in order to enable the executive to efficiently react to the uncertainty, by using a type of execution monitoring. [Morris 2001] also introduced a means to dynamically execute these partially controllable plans by using local updates at execution time. However, these techniques are centralized algorithms developed for single agent systems.

The simplest approach to extend these techniques to multi-agent systems is to use a leader-follower architecture in which one agent (the leader) performs all of the offline reasoning and online scheduling decisions. The leader schedules each task, then dispatches the tasks to the followers. The followers simply receive commands, executes them, and then sends updates to the leader when the activities are complete. This leader-follower architecture is illustrated in Figure 1-3 (a). There are several disadvantages to using this architecture. First, there is a communication bottleneck through the leader, which prevents this architecture from scaling to a large number of agents. Second, the leader introduces a single point of failure in to the system. Third, if one of the followers moves out of communication range, the follower loses its ability to receive commands and to send updates to the leader. Thus, the leader can not properly perform the online execution required to compensate for the uncertainty in the plan. In order to dynamically execute the plan under communication limitations, the plan needs to be distributed among several agents, such that each agent can participate in making the online execution decisions. The distributed architecture is shown in Figure 1-3 (b).

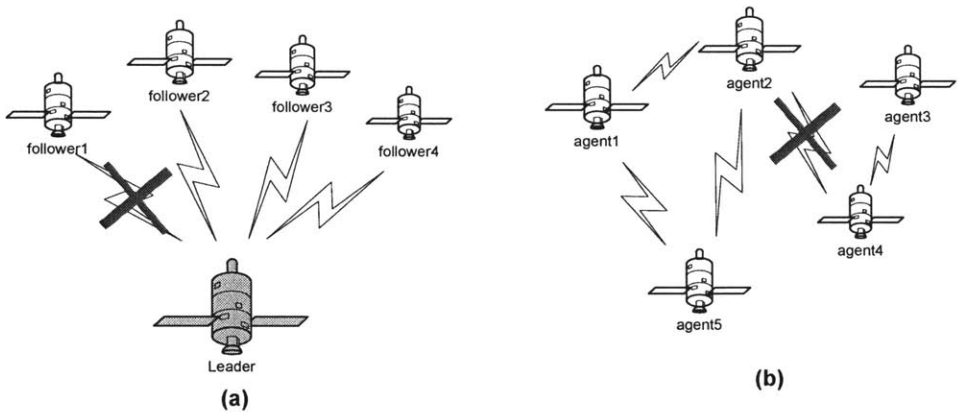


Figure 1-3 (a) In the leader-follower architecture the leader makes all of the scheduling decisions. If communication is unavailable during part of the mission, the leader is unable to dynamically schedule tasks to the follower. (b) In a distributed architecture, the scheduling decisions are made by multiple agents; therefore, the system is robust to communication limitations. For example, if the communication link between agent2 and agent4 is unavailable, as long as agent3 and agent4 do not need to synchronize their activities with the other agents, they will be able to successfully execute their local plan.

There exist several basic categories of distributed architectures. [Schetter 2003] discusses several basic types of distributed agent hierarchies for autonomous control of satellite clusters. The architectures differ based on how the intelligence (decision making, planning, scheduling, and execution) is distributed among the agents, and how they interact. The four basic general type of organization (as shown in Figure 1-4) are:

- Top-down coordination
- Centralized coordination
- Hierarchicalyl distributed coordination
- Fully distributed coordination

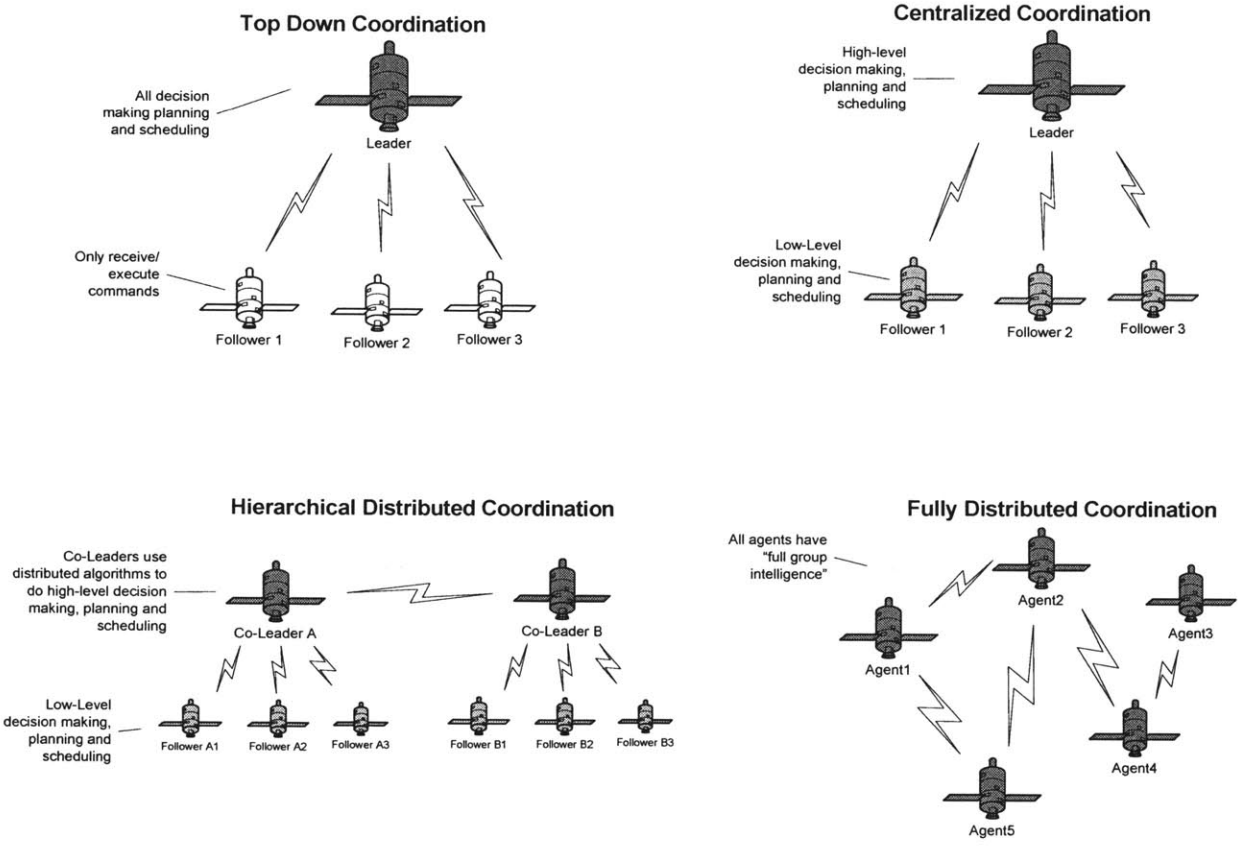


Figure 1-4 Different Types of coordination architectures for multi-agent systems

In the top-down architecture, there is one intelligent leader that does all of the decision making, planning and scheduling. The followers simply receive and execute commands.

In the centralized architecture there still is a single leader that coordinates the clusters as a whole; however, the follower agents have increased intelligence and can interact

with the leader to aid in the planning and scheduling. For example, the leader may send a task to the follower to move to a position; however, it is left up to the follower to generate the low level command sequence to achieve the task. Furthermore, the follower agents may perform local computations and may send information back to the leader, in order to enable the leader to determine the state of the whole system or aid it in doing the high level planning. Consider a case when the leader sends out a set of possible plans to a follower. The follower must then determine the best plan, and then inform the leader of its choice so that the leader can schedule the system as a whole.

In the hierarchical distributed coordination architecture, the cluster's high-level intelligence, including decision making, planning, and scheduling, is distributed among several co-leaders. Each co-leader uses a set of distributed planning and scheduling algorithms in order to coordinate the clusters. Each co-leader coordinates the decisions, planning and execution with a set of follower agents, organized similar to the centralized architecture. The distributed architecture reduces the communication bottleneck and makes better utilization of the computation resources of the cluster.

In a fully distributed architecture, each agent in the system has equal intelligence. All agents participate in decision making, planning, and scheduling for the cluster as a whole. [Schetter 2003] described these agents as having "full group intelligence". This system has the benefit of being highly adaptable, reliable and scalable; however, the architecture must handle increased communication cost.

In summary, in order to be robust to temporal uncertainty, the executive needs to dynamically schedule activities based on the outcome of uncertain durations. Dynamic execution requires the agents to communicate. However, communication between the agents may be limited during certain portions of the mission. In order to be robust to communication limitations, the plan needs to be distributed among the agents so that each agent can participate in making the online execution decisions. This leads us to the focus of this thesis, which is to create a distributed executive that can react efficiently and robustly to the temporal uncertainty, under limited communication.

1.4 Basic Centralized Architecture

The goal of this thesis is to extend the centralized autonomous planning and execution architecture, shown in Figure 1-5, to distributed Multi-Agent Systems (MAS). Specifically, this thesis focuses on developing an executive that enables a set of distributed agents to cooperatively schedule a partially controllable temporal plan in the presence of communication limitations.

The centralized architecture consists of several different functional layers, which collectively translate a set of mission goals into hardware commands. The different functional layers each play a part in converting abstraction to reality. At the top, either a human programmer or generative planner converts the mission goals into a temporal plan. The temporal plan consists of a set of strategies for achieving the mission goals. This plan contains a set of activities along with a set of temporal and symbolic

constraints, constraining the activities in the plan. The temporal plan is specified with a set of functionally redundant means to achieve the mission goals. The temporal planner's job is to select one means to achieve the goals based on viability or optimality [Williams 2001, Kim 2000]. It defers the scheduling of the plan to the executive. When the executive receives the plan it still contains some temporal flexibility.

The goal of the executive is to dynamically schedule each task of the plan. The executive is composed of two main components: a reformulator and a dispatcher. The reformulator compiles the temporal constraints of the plan offline in order to enable the dispatcher to efficiently schedule the tasks at execution time. The reformulator provides a guarantee that the dispatcher can consistently and dynamically execute the plan. The dispatcher uses a dynamic execution strategy, which schedules and executes the tasks simultaneously. The dispatcher uses a type of online execution monitoring in order to adapt the schedule of each task based on the outcome of uncertain events at execution time. The dispatcher sends the tasks to a reactive controller, which determines the current state of the agent and determines the low level means to achieve each task. The reactive controller directly interacts with the hardware by sending commands and receiving observations from the sensors. The executive schedules mid-level tasks such as "place manipulator at position x", whereas, the reactive controller is in charge generating the low-level commands to the actuators in order to achieve this task.

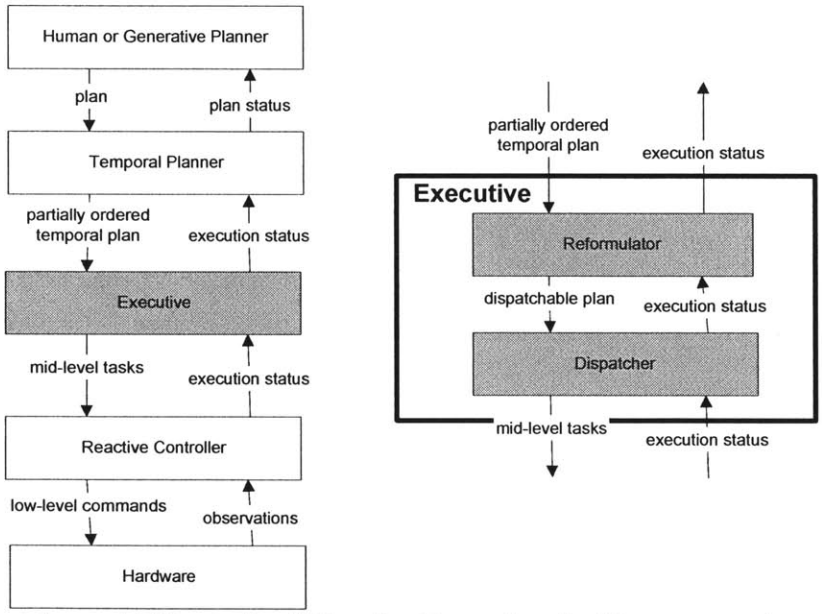


Figure 1-5 Centralized Planning/Execution Architecture An executive is a scheduler that converts a partially ordered temporal plan into a sequence of hardware commands. The executive consists of a reformulator, which prepares the plan for execution, and a dispatcher, which schedules tasks in real-time in order to adapt the schedule to the uncertain events.

The above planning/execution architecture achieves robustness by combining both reactive and deliberative components (a hybrid system). The reactive controller provides

the ability to adapt tasks to the current state of the agent and the dispatcher provides the ability to adapt to temporal uncertainty. In this thesis we are particularly interested in the reformulator, which reasons about the temporal uncertainty in the plan prior to execution. The reformulator guarantees that the execution will be successful barring some hardware failure, while allowing the agent to adapt the schedule to run-time uncertainty.

The architecture described above is in contrast to purely reactive systems, popularized by [Brooks 1986], which continue to gain popularity in the Multi-Agent Systems community through swarm intelligence [Eberhart 2001, et. al]. [Parker 2001] gives a good reference for current work in distributed robotics with emphasis on current behavior based approaches. Furthermore, NASA is investigating a distributed behavior based architecture called CAMPOUT [Huntsberger 2003]. Behavior based approaches generate actions through interaction – there is no explicit plan. These are able to adapt to novel situations and works well to control low level coordination when the agents are able to maintain constant communication. However, the architecture lacks the ability to do any complex temporal planning. Furthermore, it lacks the ability to coordinate the agents' tasks when they cannot communicate. For this reason, these systems cannot guarantee that their execution strategy will succeed. Theses systems rely on a set of emergent behaviors to achieve their mission goals, making them difficult to analyze. For this reason, there has been a strong reluctance to allow these purely reactive autonomous systems to manage mission critical tasks, particularly within the risk averse space exploration community.

At the other end of the spectrum, there are complete deliberative systems that offer the programmer a stronger level of control over the systems behavior, by explicitly generating a consistent temporal plan prior to execution, such as Kirk, ASPEN, Europa, and LPGP. [Kim 2001, Chien 2000, Johnson 2000, Long 2002]. These systems allow the programmer to generate plans for tight temporal coordination and provide a level of assurance that the plan will succeed. Furthermore, they provide some limited ability to react to temporal uncertainty at execution time by using a dynamic execution strategy. However, without using an explicit model of uncertainty, these systems cannot guarantee that plans that require tight coordination will succeed. Uncertainty makes these plans susceptible to execution failure. In order to deal with plan failure at execution time, the system must re-plan. In order to efficiently re-plan, the system must either pre-compute a set of contingent plans that it chooses from upon failure or must provide a fast means to re-plan. For example, [Drummond 1994] proposed a method to compute contingencies for the most likely failure mode; however, it requires exponential space in the worst case. Furthermore, recent methods of incremental temporal consistency checking [Shu 2003] have mitigated the cost of temporal reasoning during replanning; however, replanning still carries the burden of requiring exponential time in the worst case.

1.5 Problem Statement

The problem addressed in this thesis is to efficiently reformulate cooperative multi-agent temporal plans that contain an explicit model of uncertainty, in order to enable a set of distributed agents to robustly and efficiently execute these plans when communication is limited between the agents at execution time.

1.6 Proposed Approach

In this thesis we introduce a two layer approach that clusters the tightly coordinating portions of the cooperative multi-agent plan into a set of group plans such that each group plan loosely coordinates with one another. The group plan clustering is done such that the agents that participate in each tightly coordinating group plan are able to reliably communicate with each other at execution time; however, they may not be able to communicate with agents outside their group. This isolates the problem of dealing with communication limitations to inter-group communication limitations. Given this two-layer plan structure, we introduce a *hierarchical reformulation* (HR) algorithm that compiles the temporal constraints of the plan such that each group plan are scheduled statically with respect to one another, while still enabling the activities within each group plan to be scheduled dynamically. Statically scheduling the group plans with respect to one another removes the need for the agents to communicate outside their group and dynamically scheduling the tightly coordinating group plans enables the agents to robustly adapt to temporal uncertainty at execution time. The basic approach is represented in Figure 1-6.

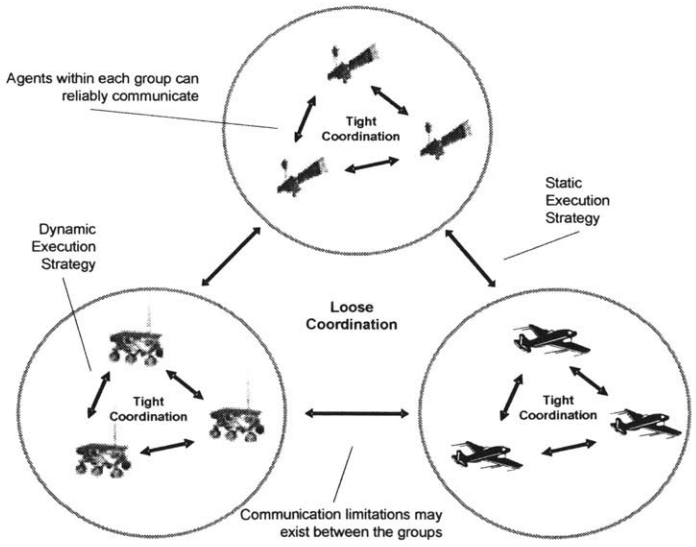


Figure 1-6 In order to adapt to runtime uncertainty within each tightly coordinating group plan, the agents use a dynamic execution strategy, and in order to cope with limited communication between the groups, the agents use a static execution strategy.

The proposed approach exploits the observation that communication availability tends to be conjunct with tight coordination and tight coordination is necessary when you need to communicate. In a distributed multi-agent system, when a set of agents need to tightly coordinate their activities and some of their activities have uncertain durations, they need to send one another updates (either explicitly or implicitly) so the rest of the agents can dynamically adjust their schedules. Note that it is the relative flexibility of the temporal constraints between the activities compared to the uncertainty of the activities themselves that determine whether communication is required. For example, even though a dance troupe requires tight synchronization between the dancers, they can achieve this tight synchronization without communicating with one another by practicing. Practicing removes the uncertainty of their actions to a point in where they can synchronize their actions using a fixed schedule (i.e. triggered by the beat). The uncertainty is small compared to the synchronization requirements. However, when the uncertainty is large compared to temporal requirements imposed between the activities, the agents need to communicate. In other cases communication is required in order to be efficient. For example, consider a scenario when two UAVs plan to rendezvous, then fly off together to another common location. In this case, the UAVs should leave as soon as both UAVs have arrived at the rendezvous point, rather than waiting around (and wasting fuel) for some pre-specified departure time.

In general, tight coordination requires communication; however, communication may be limited. Fortunately, the agents require tight coordination only when the agents are in close proximity with one another and this is when they can communicate. Therefore, when communication is needed, it is available. Conversely, when the agents are far apart, communication may be impossible or expensive. However, in these cases we expect the agents only to require loose coordination, and this loose coordination enables the agents to synchronize their actions by fixing their activities with respect to one another. Therefore, when communication is unavailable, it is not needed.

Consider the sensor web scenario. The satellite cluster requires tight coordination and hence communication, in order to perform its interferometry task; the satellites' close proximity enables them to maintain reliable and relatively inexpensive communication. Therefore, the satellites are able to send execution updates to one another in order to dynamically adjust their schedules. In contrast, the satellite cluster may not always be able to communicate with the UAV fleet. However, the satellites' tasks and the UAVs' tasks are fairly decoupled. In this case, the UAV fleet can synchronize their activities with the satellite cluster by statically scheduling the start of their group plans with respect to one another.

The two layer approach proposed in this thesis is similar to the type of schedule that people use to manage a large project, such as designing a spacecraft, building a home, or managing a sports organization. The total number of people and activities required in order to complete the entire job is enormous. Furthermore, there typically contains many concurrent activities with complex temporal constraints between the activities. However, the project manager is able to manage the project as a whole without getting into the

details of each activity. For example, a spacecraft project manager can fix the schedule of the design, build, test, and launch phases without considering the interactions of every engineer. Similarly, the baseball schedule is fixed at the beginning of the season without getting into the detailed time constraints of each player. The project manager is able to generate this schedule by considering the expected duration for each high level activity, then scheduling them to satisfy the inherent temporal constraints. The detailed scheduling of each low level activity is left up to the individuals within the group. Each group is able to dynamically adjust their schedules on a day-by-day, hour-by-hour, or minute-by-minute basis, as needed. By statically scheduling at the high level, each team is able to work independently. However, the project can get into trouble if the project manager does not adequately assess the uncertainty of the system or does not appropriately consider the temporal constraints between each activity.

In this thesis we provide a set of algorithms that analyze temporal uncertainty of the activities and the constraints between these activities, in order to enable the executive to schedule the high level activities prior to execution and the low level activities at execution time. In general, a large project may contain several layers of management. Each level of management may consider the project at a different level of detail. However, in this thesis we limit our approach to two layers.

In this thesis we introduce a two layer plan in order to simplify the executive's task of reasoning about both temporal and communication constraints. The two-layer plan consists of a top level *mission plan*, and a set of lower level *group plans*. The mission plan specifies the temporal constraints between each group plan and each group plan specifies a set of temporal constraints between the agent activities. The set of agents that participate in each group plan are simply referred to as a *group*. We assume that agents within a group are able to maintain reliable communication with one another; however, this is not necessarily true for agents in different groups. The details of each group plan are hidden from the mission plan. This is done by using a simple abstraction for each group plan, called a *macro,* in the mission plan. The macro is an executive summary of the group plan that represents the feasible duration of the group plan. Replacing the group plan with the macro enables the executive to reason about the group plan interaction at a high level without getting in to the details of each group plan.

Example 2-1:

In order to ground our discussion, consider the real life scheduling problem called the master's student's birthday party problem. The two layer plan is shown in Figure 1-7 and is representative of the types of plans our executive intends to solve. This rather simple scheduling problem successfully exposes several interesting aspects of multi-agent scheduling. If the reader is unfamiliar with STNs or STNUs, the reader should read the appropriate sections in Chapter 2 before proceeding with this example.

The plan is carried out by three "agents": Mother, Brother, and Student. All three of them must coordinate their activities in order to have a successful birthday party.

However, it is undesirable to have everybody updating one another after completing every action. The two-layer plan structures the plan such that the agents only need to update members of their own group. The high-level plan structure is represented by the mission plan shown in Figure 1-7(a). In this plan the brother and student go to the movies (Watch-Movie group plan) while the mother bakes the birthday cake (Bake-Cake group plan). This is followed by all three of them getting together for the birthday party (Birthday-Party group plan). The mission plan specifies a set of loose temporal constraints between each high-level activity. Specifically, the mission plan specifies that that the brother and student must be done watching the movie between [10, 90] minutes before the party starts. Similarly, the mother must finish baking the cake between [10, 60] before the party starts. Furthermore, the entire mission must not take longer than 360 minutes. Note that the mission plan places no constraints on the duration of the group activities, hence the [0, INF] bound is initially placed on each group activity. Furthermore, there does not exist any explicit constraints restricting when the brothers need to start the Watch-Movie group plan nor when the mother needs to start Bake-Cake group plan. These constraints need to be deduced.

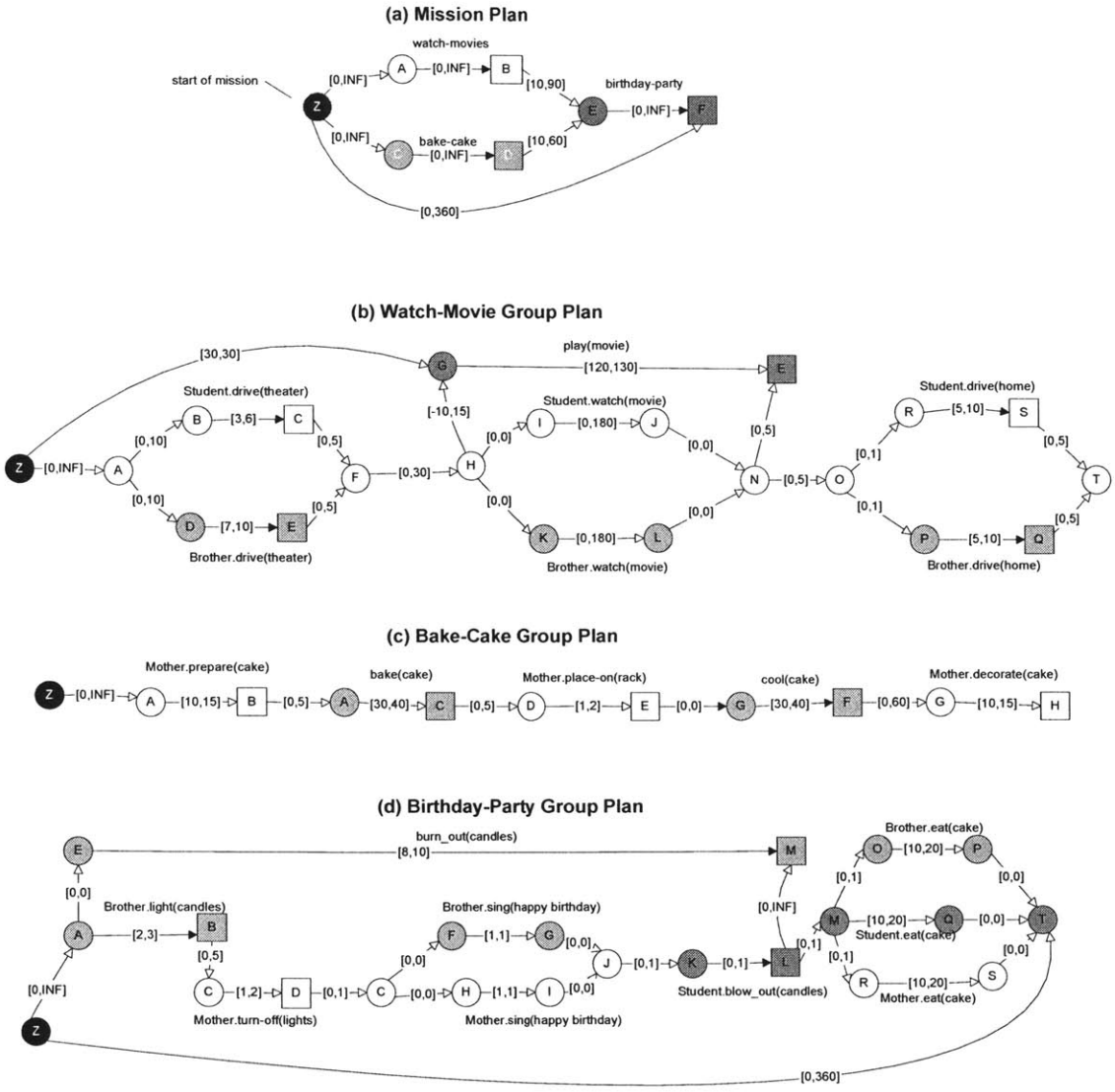


Figure 1-7 The Masters Student's Birthday Party Problem (a) The mission plan shows the interaction between the three group plans: Watch-Movie, Bake-Cake, and Birthday-Party. (b) The Watch-Movie group plan involves the Student and Brother going to see a movie. (c). The Bake-Cake group plan involves the mother baking a cake. (d) The Birthday-Party group plan involves everyone participating in a traditional the birthday cake eating ritual.

The three group plans are shown in Figure 1-7(b,c,d). Each group plan specifies a set of tightly coordinated activities, performed by one or more agents, that are needed in order accomplish each high level activity. Note that each "agent" participates in multiple group plans but never participates in the same group plan at the same time. Furthermore, each group plan shares a special timepoint Z (zero timepoint) with the mission plan. This

timepoint is executed before all others. Specifically, this timepoint is always executed at time = 0. By sharing this common timepoint, the group plans have the same temporal reference frame. It is a reference point from which to express constraints in absolute time. For example, in the Watch-Movie group plan there is a constraint ZG that specifies the movie start exactly 30 minutes after the start of the plan. Assuming the mission starts at 4:00 PM, this constraint specifies that the movie starts at 4:30 PM. Without this fixed point Z, the group plan would not be able to represent these types of constraints. Note that our approach is to fix the start of each group plan; however, this start time is not known *a priori.*

In order to schedule each group plan, the agents need to resolve several different types of scheduling issues. First, the executive needs to be able to dynamically respond to uncertain events. This includes responding to the uncertainty of one another's activities, as well as responding to the uncertainty of "natural" events. For example, in the Bake-Cake group plan, the mother needs to respond to the uncertainty in baking time of the cake. The cake will take between 30 and 40 minutes to bake; and the plan specifies that she needs to remove the cake from the oven no more than 5 minutes after it is baked through. In order to satisfy this constraint she must monitor the cake and adjust her schedule accordingly. Second, the executive needs to reason about uncertainty prior to execution, placing additional constraints on the plan where necessary, in order to ensure that the explicit constraints in the plan are satisfied at execution time. For example, in the Watch-Movie group plan, the executive needs to reason about the uncertainty in both the brother's and student's drive times and must restrict their start times with respect to one another, in order to ensure that neither one is waiting at the theater for more than 5 minutes alone. Third, the executive needs to reason about the relationship between the constraints within each group plan and the constraints of the mission plan, in order to ensure that the execution times selected for start of each are consistent.

This plan is typical of multi-agent plans, in that the plan as a whole consists of a set of concurrent and serial activities with a set of complex temporal and communication constraints relating these activities. However, the reasoning required in order to schedule the plan is simplified by breaking the plan down into a set of smaller sub-plans and by coordinating the sub-plans by using a set of fixed synchronization points.

In order to prepare the plan for execution, we introduce a novel hierarchical reformulation (HR) algorithm, illustrated in Figure 1-8. The HR algorithm first structures the plan into a two layer plan as previously described. The HR algorithm applies a decoupling algorithm based on the strong controllability algorithm introduced by [Vidal 2000] in order to generate a fixed schedule for each high level activity. This removes the need for communication between each group plan at execution time. Then we apply a new dynamic controllability algorithm in order to ensure that each group plan can be scheduled dynamically, in order to respond to uncertainty at execution time. We also apply an edge trimming algorithm to the resulting group plan, in order to efficiently execute each group plan.

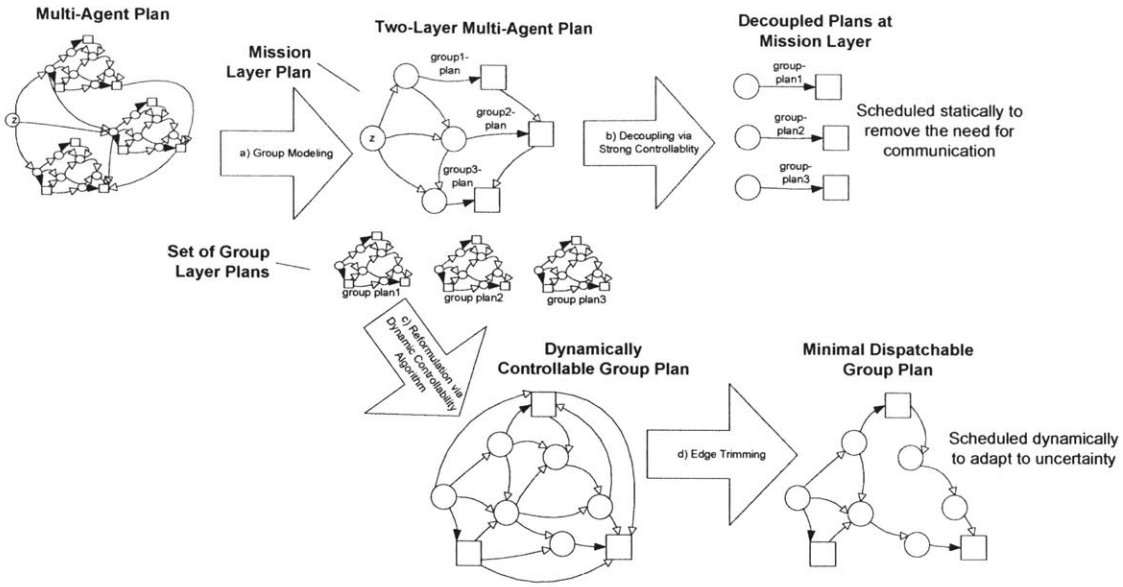


Figure 1-8 Overview of Hierarchical Reformulation Algorithm

The overall block diagram of our approach is shown in Figure 1-9. The two layer plan is created either by compiling a plan that is specified in an augmented version of the Reactive Model-Based Programming Language (RMPL), called the Group Planning Language (GPL), or by clustering the tightly coordinated portions of a fully elaborated plan. The temporal constraints of the two-layer plan are compiled using a centralized reformulator. The heart of the reformulator is the Hierarchical Reformulation (HR) algorithm, which uses a combination of a dynamic controllability algorithm and a decoupling algorithm in order to prepare the plan for execution. After reformulation, each group plan is distributed to a leader of each group. The leader of each group is in charge of dispatching commands to each agent in the group.

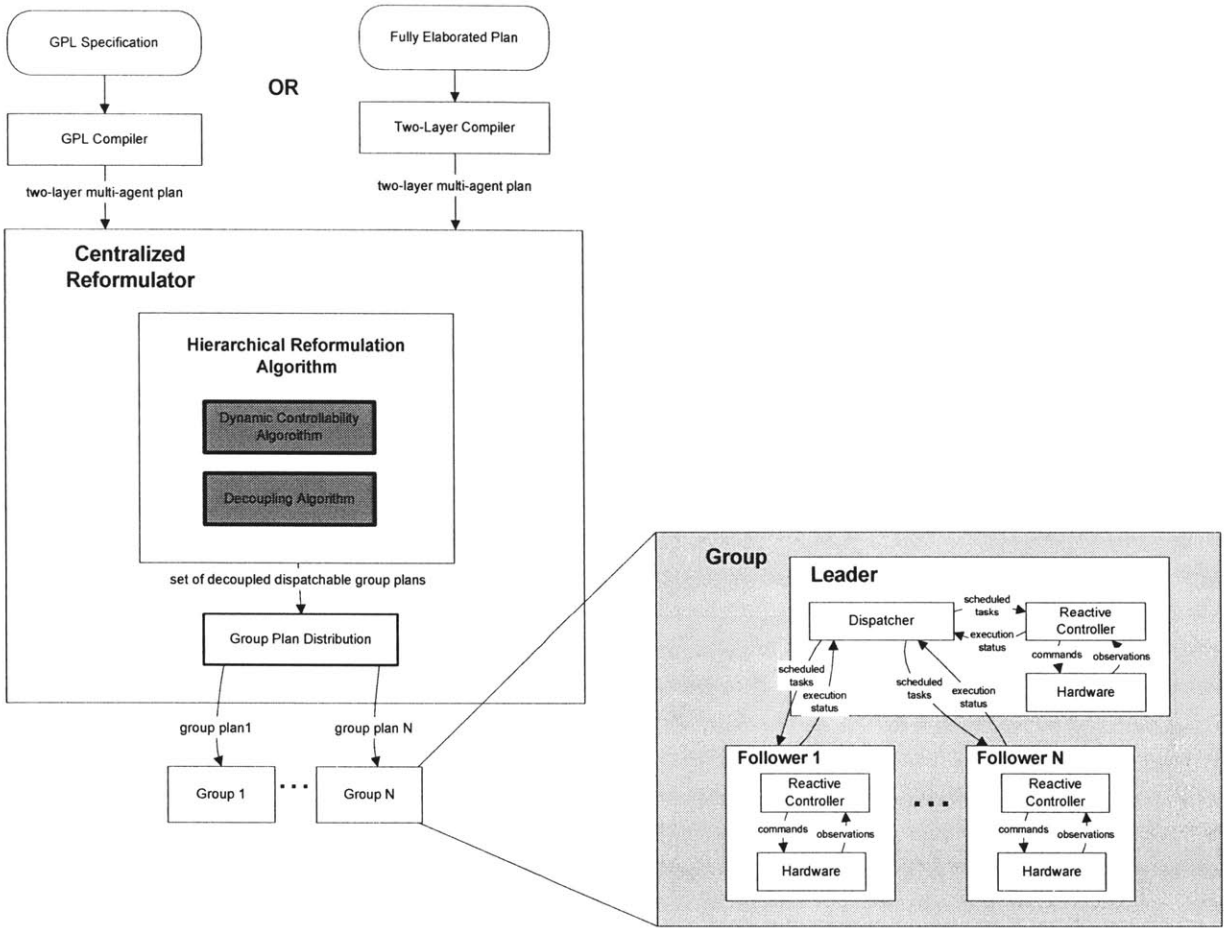


Figure 1-9 Distributed Executive Block Diagram

There are several benefits to organizing the plan in a two layer structure. First it provides an efficient way to deal with communication limitations between agents. Second, the two layer structure enables a divide-and-conquer approach in which the dynamic controllability algorithm, the most expensive operation, is only applied to plans of limited size. Third, although we focus on addressing communication requirements associated with dynamic scheduling, this two-layer structure also ensures that any communication required by the reactive controller, such as state updates, is available between agents within the group.

The algorithms presented in this thesis are presented in the context of execution; however, this work also falls into the realm of temporal planning. Specifically, the task of the hierarchical reformulation algorithm is to reformulate the plan for execution; however, this reformulation is not guaranteed to succeed. The techniques presented in this thesis can be applied during planning, in order to detect if a candidate plan is

feasible. Furthermore, the approach taken in this thesis should not be seen as a replacement for the behavior based multi-agent approaches, but rather as a complement to them. In the future, autonomous robots will need to use both deliberative planning and scheduling techniques, as well as more purely reactive behaviors in order to be robust and adaptive

1.7 Key Technical Contributions

This thesis makes three technical contributions. First, we introduce the Hierarchical Reformulation (HR) algorithm, which exploits strong and dynamic controllability in order to enable a group of agents to dynamically schedule their activities under communication limitations. Second, we present a formal treatment of the dynamic controllability of systems that contain communication limitations, termed *communication controllability*. Third, we provide a novel, fast dynamic controllability (FAST-DC) algorithm. This algorithm is applicable to both multi-agent and single agent systems.

1.8 Grand Vision

The approach introduced by this thesis makes steps toward a completely distributed planning and execution architecture. In the grand vision, each agent in the system operates autonomously using a set of distributed algorithms in order to plan and execute their cooperative plans. In the future work section, we provide initial work on how both the reformulation and dispatching algorithms can be mapped to a distributed algorithm.

1.9 Range of Applicability

Although most of the examples focus on robotics, particularly Mars exploration, the ideas presented in this thesis are applicable to a wide variety of real-world scheduling problems, including but not limited to the following:

- Groups of rovers exploring Mars,
- Cluster of satellites for interferometry missions,
- Coordinated activities within a single spacecraft, where each component is treated as a separate agent, and
- Execution of commands on a set of distributed tiny processors.

1.10 Roadmap for Thesis

Chapter 2 presents background on Simple Temporal Networks (STNs) and Simple Temporal Networks with Uncertainty (STNUs). Specifically, we show how to reformulate and dispatch STNs in the centralized case. Chapter 3 starts by presenting the formal definition of communication controllability. Next, it formally defines the two-layer MTPNUs and shows how to construct these two-layer plans. Then it presents the decoupling algorithm and the Hierarchical Reformulation (HR) algorithm. Chapter 4 presents the fast dynamic controllability (Fast-DC) algorithm. Chapter 5 discusses the

implementation of the Hierarchical Reformulation algorithm, along with empirical results for Fast-DC algorithm. It also discusses directions for future work.

2 Background

2.1 Introduction

This chapter provides the necessary technical background needed to understand dynamic execution of temporally flexible plans. As such, the reader may need to refer back to this chapter many times, while reading the subsequent technical chapters. The reader should focus on gaining a good working knowledge of the high level ideas by studying the canonical examples given in this chapter.

The outline of the chapter is as follows. First, we review the theory of Simple Temporal Networks (STNs) and the Temporal Plan Networks (TPNs) which encode temporally flexible plans. Then we review the reformulation and execution algorithms introduced by [Tsarmardinos 1998] and [Muscettola 1998a]. Executing a temporally flexible plan is a two step process that consists of 1) the offline *reformulation phase* that compiles the temporal constraints of the plan, and 2) an efficient online dynamic *dispatching phase*, which dynamically schedules the timepoints in the reformulated plan. The reformulation phase enables the dispatcher to efficiently and consistently schedule the plan during execution. Then we review Simple Temporal Networks with Uncertainty (STNUs) [Vidal 1996] and the associated Temporal Plan Networks with Uncertainty (TPNU), which introduce an explicit model of uncertainty into the temporal plan. A TPNU enables the plan to represent activities whose durations are not controlled by the executive, but rather by nature. Hence, these plans are referred to as partially controllable plans.

By the end of this chapter, the reader should have a good working knowledge of STNs. Specifically, the reader should 1) understand how to reformulate STNs into a dispatchable network, 2) understand how the dispatcher dynamically executes these reformulated networks, and 3) understand how to represent an explicit model of uncertainty within temporal plans.

2.2 Temporal Constraint Satisfaction Problem

In this thesis we are concerned with *temporally flexible plans*. These temporal plans are a set of partially ordered activities along with a set of temporal constraints, which model the duration of each activity and constrain the execution time of each activity with respect to one another. Each activity in the plan is associated with two instantaneous events, called *timepoints*. Specifically, the start of each event is associated with a start timepoint and the end of each activity is associated with an end timepoint. The start and end timepoints are separated by a non-negative duration. The temporal constraints of the plan are represented by a set of inequalities, which constrain the timepoints of the activities with respect to one another. The actual time occurrence of a timepoint A is written T(A), or T_A.

For example consider how the following statement is converted into temporal constraints involving inequalities.

- *"Study for the exam for at least 1 hour and at most 3 hours"*
  1 hour $\leq$ T(end_study) – T(start_study) $\leq$ 3 hours

In general, the set of constraints form an instance of a Temporal Constraint Satisfaction Problem (TCSP) [Dechter 1991].

A TCSP:

- A set of timepoints X_i at which the events occur
- A set of disjunctive unary constraints: $(a_o \leq X_i \leq b_o)$ **or** $(a_1 \leq X_i \leq b_1)$ ...
- A set of disjunctive binary constraints: $(a_o \leq X_j - X_i \leq b_o)$ **or** $(a_1 \leq X_j - X_i \leq b)$..

A solution to the TCSP is a *schedule,* T, which is an assignment to each timepoint X_i such that all of the temporal constraints are satisfied. In general, solving the TCSP is NP hard [Dechter 91], because the algorithm that generates the schedule, T, must consider every possible combination of disjunctive constraints. In the next section, we review simple temporal networks that only use non-disjunctive binary constraints.

2.3 Simple Temporal Network and Temporal Plan Networks

In this section we review Simple Temporal Networks (STNs) and Temporal Plan Networks (TPNs). An STN only contains simple binary temporal constraints between timepoints. The STN is simple, because it does not allow disjunctive temporal constraints. Specifically, it only allows one interval between each pair of timepoints. STNs have been widely used in planning and scheduling, because they enable fast temporal consistency checking and can be scheduled dynamically, yet they are expressive enough to represent many real-world problems. The temporal consistency of an STN can be checked in polynomial time using a Single-Source Shortest-Path (SSSP) algorithm such as Bellman-Ford SSSP [CLR 1990, Dechter 1991]. Furthermore, other well known graph algorithms, such as Floyd-Warshall All-Pairs Shortest-Path (APSP) [CLR 1990] algorithm can be used to derive a set of implied constraints encoded by the explicit constraints.

An STN is visualized as a directed graph G = <N,E>, where the timepoints of the graph represent the timepoints, and the directed edges represent the simple temporal constraints. Each edge, AB, between timepoints A and B, contains a lower and upper bound [lb,ub] such that, $lb \leq T_B - T_A \leq ub$. For example, consider the STN shown in Figure 2-1(a). The STN contains four timepoints and four directed edges constraining the execution time of the timepoints. Consider the edge AB, the interval [0,8] imposes one upper bound constraint, $T_B - T_A \leq 8$, and one lower bound constraint, $T_B - T_A \geq 0$.

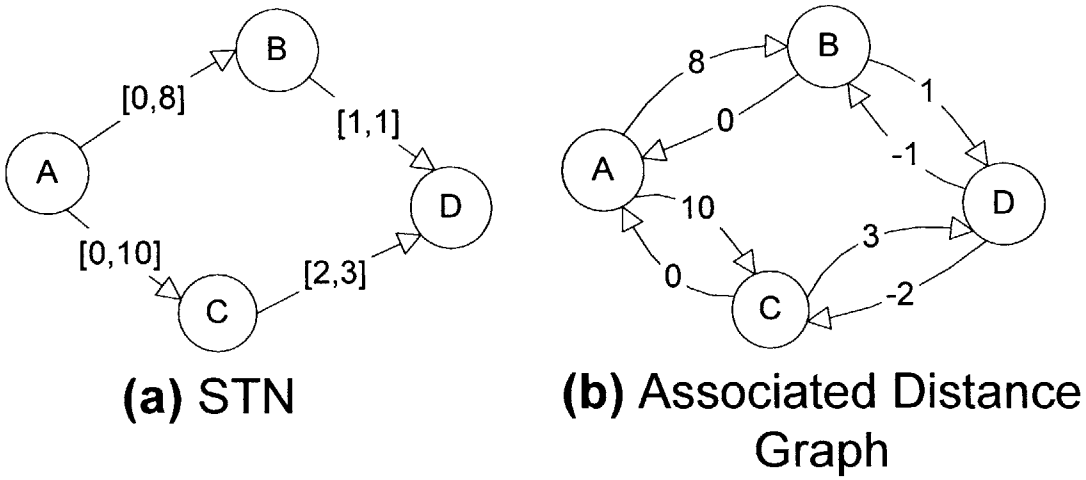


Figure 2-1 (a) Simple Temporal Network (STN) (b) The associated distance graph of the STN

The formal definition of a STN is given below:

Definition 2-1 (STN [Dechter 1991]): *A STN is a 4-tuple <N, E, l, u> where, N is a set of timepoints, E is a set of directed edges. Each edge E, between timepoints A and B contains a lower and upper bound temporal constraint, where, $l : E \rightarrow \Re \cup \{-\infty\}$ and $u : E \rightarrow \Re \cup \{+\infty\}$ are functions mapping edges to the extend Real Number such that $l(AB) \leq T(B) - T(A) \leq u(AB)$.*

A Temporal Plan Network (TPN) generalizes the STN to include activities. A TPN is a set of activities to be performed, each of which includes a start and end time, together with a set of temporal constraints that specify the valid start and end times for each activity. The temporal constraints are specified as simple temporal constraints. Hence, a TPN is a generalization of a STN consisting of a set of activities A, and a mappings, T^+: $A \rightarrow N$, and T^-:$A \rightarrow N$, mapping the start and end times to the timepoints in the STN. We say a TPN is constrained by a STN.

Consider the simple TPN shown in The TPN specifies a plan where the two rovers explore two different regions. The rovers are free to start exploring any time between [0,10] minutes. Each rover is free to explore for [30,60] minutes before returning to some rendezvous location. Once at the rendezvous location, the plan specifies that the rovers should not wait more than 5 minutes. Furthermore, the rovers must complete their exploring at least 10 minutes before the sunsets, which occurs 60 minutes after the start of the plan. Note that the +INF bound on the link EF corresponds to places no constraints on the plan. Also, note that the TPN also contains a special timepoint Z

which is not associated with any activity. The plan is flexible, and only partially ordered.

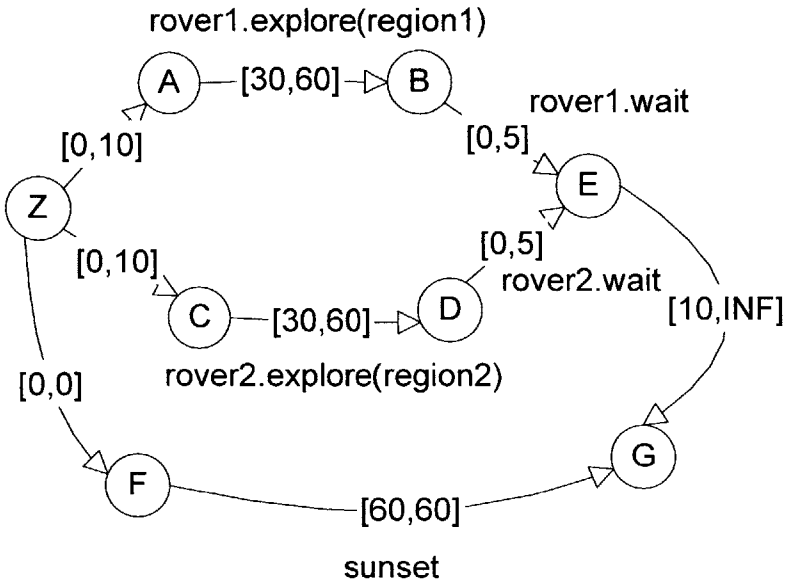


Figure 2-Error! Not a valid bookmark self-reference.**2-2** The TPN specifies a plan where two rovers explore separate regions then wait for the other to arrive. Their activities are constrained with respect to one another as well as with respect to the sunset, which occurs 60 minutes after the start of the plan.

Every STN has an equivalent representation, called the *distance graph* [Dechter 1991]. The distance graph is a simple, but extremely useful representation of the temporal constraints. In the distance graph, every edge only contains one constraint. Specifically, each edge AB in the distance graph has a distance, b(A,B), which specifies the constraint: $T_B - T_A \leq b(A,B)$.

A distance graph edge represents an upper bound on B with respect to A if the b(A,B) ≥ 0, and, the edge represents a lower bound on B with respect to A if $b(A,B) < 0$. The reason a negative edge represents a lower bound is easily seen by performing some simple algebra. Consider a situation where the time of B must occur at least 5 minutes after A. This is represented by the inequality: $T_B - T_A \geq 5$. However, this does not fit the form of the distance graph edge. Simply multiplying the inequality by -1, results in a new inequality: $T_A - T_B \leq -5$, which is in the proper form. This corresponds to an distance graph edge BA with distance b(B,A) = -5.

In order to avoid confusion, the constraints in the STN are referred to as links, whereas, the constraints in the distance graph are referred to as edges. An STN can be converted into a distance graph by replacing each link in the STN with a pair of directed edges, such that each link AB $\in$ [lb,ub] in the STN is replaced by one upper bound edge AB with b(A,B) = ub, and one lower bound edge BA with b(B,A) = –lb. Note that the

lower bound value is negated. For example, Figure 2-1(a) shows a STN and **Figure 2-1**(b) shows the associated distance graph. In particular, consider the STN link CD of [2,3], in **Error! Reference source not found.**(a). This STN link is converted into one upper bound edge CD with b(C,D) = 3, and one lower bound edge DC with b(D,C) = -2, in **Figure 2-1**(b).

The formal definition of a distance graph is given below.

Definition 2-2 (Distance Graph [Dechter 1991]) *A distance graph, $D = < N, E, b >$, is a weighted directed graph, where N is the set of timepoints, E is the set of directed edges and b: $E \rightarrow \Re \cup \{+\infty,\} \cup \{-\infty,\}$ maps the edges, E, to the extended Real Numbers. A distance b(AB) imposes an constraint that $T(B) - T(A) \leq b(A,B)$.*

The links of the STN fall into different categories depending on the values of the lower and upper bound. Consider a STN link AB $\in$ [x,y]. **Figure 2-3** shows the 5 different types of STN links along with the associated distance graph. If $x,y \geq 0$, and $y > x$, then STN link represent a true upper and lower bound on B with respect to A, as shown in **Figure 2-3** (a). In this case timepoint A must always precede timepoint B. If $x = y$, then the B is rigidly constrained with respect to A (i.e. there is no flexibility in the execution time), as shown in **Figure 2-3** (b). In the special case when $x = y = 0$, then the timepoints A and B are *zero-related* by a *zero-zero constraint*, as shown in **Figure 2-3** (c). If $x < 0$ and $y > 0$, then the edge represent two upper bounds. In this case the execution order between A and B is undetermined, as shown in **Figure 2-3** (d). If $x > y$ then the constraint is inconsistent, as shown in **Figure 2-3** (e). In this case the lower bound is greater than the upper bound, which is a direct inconsistency. Specifically, B must wait at least 5 time units after A, but no more that 3 time units. Finally, if $x \geq 0$ and $y < 0$, then this corresponds to two lower bound edges, which is an extreme example of temporal inconsistency, as shown in **Figure 2-3** (f). In this case, A must execute before B, and B must execute before A.

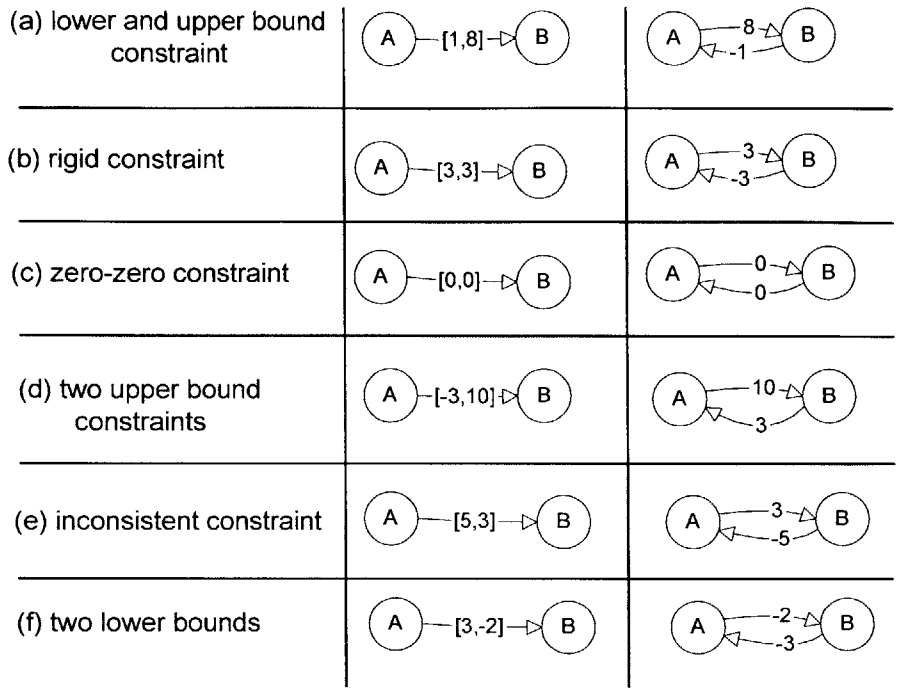


Figure 2-3 Different types of STN links

The distance graph provides a means to derive the implicit constraints. The implicit constraints are simply derived by computing shortest paths in the distance graph. The implicit constraints are derived by simply combining inequalities imposed by the edges. For example, consider the distance graph in Figure 2-4(a). The implied constraint AC is derived by combining the constraint on edge AB and edge BC as follows.

$$\begin{aligned} T_B - T_A &\le 8 && \text{: edge AB} \\ T_C - T_B &\le 1 && \text{: edge BC} \\ \hline T_C - T_A &\le 9 && \text{: derived edge AC} \end{aligned}$$

In this simple example, there is only one possible path from A to C; however, in a general distance graph there exist many possible paths. In order to compute the tightest (most restricted) implied constraint we must consider all possible paths. Fortunately, finding shortest paths in directed graphs is a well studied problem. Specifically, the tightest constraints between every pair of timepoints can by computed by either the well known Floyd-Warshall All-Pair Shortest-Path (APSP) algorithm which runs in $\Theta(N^3)$ time or computed using Johnson's algorithm, which runs in O(NE lg N) time when implemented using a binary-heap [CLR 1990].

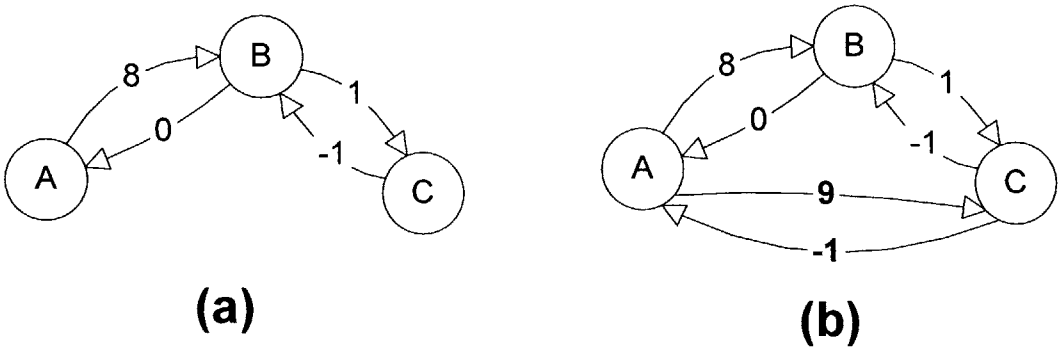


Figure 2-4 (a) The distance graph containing only the original constraints (b) The edges AC and CA are deduced by computing the shortest path ABC and CBA, respectively.

The consistency of the STN is also determined by computing the shortest paths in the distance graph. [Dechter 1991] showed that a STN is temporally consistent iff the associated distance graph contains no negative cycles. By definition, an STN is consistent if there is an assignment to the timepoints that is consistent with every temporal constraint. Consider the distance graph shown in **Error! Reference source not found.** The negative cycle ABDCA implies an edge AA of -1. This means $T(A) - T(A) \leq -1$, which is inconsistent because $T(A) - T(A)$ is always 0.

Negative cycles can be checked by computing the APSP graph then checking for negative distances on the diagonals [CLR 1990]. There also exist other well known algorithms, such as Bellman-Ford Single-Source Shortest-Path, which detects negative cycles without constructing the APSP graph, which requires N^2 space [CLR 1990]. Furthermore, the Bellman-Ford SSSP algorithm runs in O(NE) time [CLR 1990].

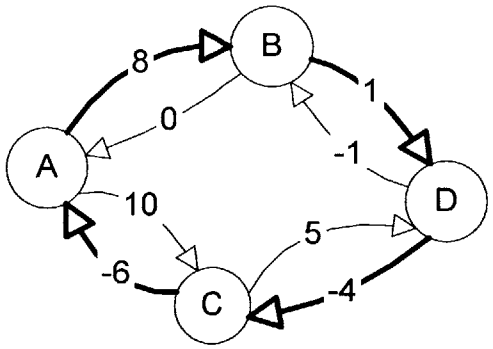


Figure 2-5 The distance graph is inconsistent because there is a negative cycle ABDCA.

Two good resources that formally cover other properties of STN include [Dechter 1991] and [Hunsberger 2002].

2.4 Dynamic Execution of TPNs

In this section we describe the process of dynamically scheduling a TPN. A TPN contains temporal flexibility and in order to exploit this flexibility, the timepoints are scheduled online. Executing a TPN can be broken down into two phases, an offline reformulation phase, followed by an online dispatching phase as shown in Figure 2-6. The offline reformulation phase compiles the temporal constraints in to a *minimal dispatchable network*. This enables the dispatcher to efficiently execute the plan using a small number of local propagations during execution.

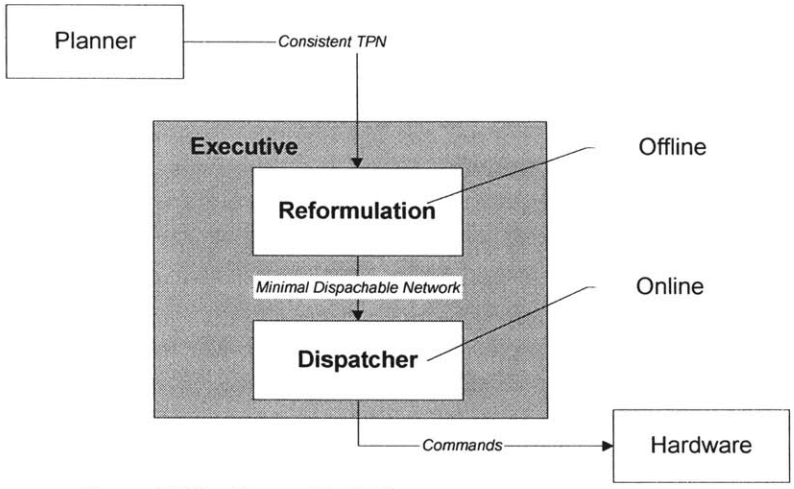


Figure 2-6 Plan Runner Block Diagram

The goal of the dispatcher is to dynamically generate a consistent schedule for timepoints in the plan. The dispatcher uses a least commitment execution strategy, where the timepoints are scheduled and tasks associated with the time point are executed, simultaneously. Specifically, the dispatcher starts any activity whose start time is associated with the timepoint being executed or stops any activity whose end time is associated with the time point being executed. The dispatcher determines the schedule for future timepoints by using the execution times of the past. In doing so, it makes scheduling decisions only after as much uncertainty about the past has been resolved.

The dispatcher dynamically schedules the timepoints by switching between executing timepoints and *locally* propagating the execution time to future timepoints. These propagations are used to update the *execution window* for unexecuted timepoints. [Dechter 1991] showed that the initial execution window can be computed by using two SSSP computations. An execution window consists of lower and upper bounds, which represents the range of feasible execution times for each timepoint. The dispatcher is free to choose any time within the timepoint's execution window as long as the timepoint is

both *alive* and *enabled.* The next example will help us to define what we mean by alive and enabled and the STN_DISPATCHING algorithm shown in Figure 2-7.

```
function STN_DISPATCHING(G)
Input: a dispatchable distance graph G
Effects: dynamically schedules each timepoint in G
1. Let
        A= {start_time_point}
        current_time = 0
        S= { }

2. Arbitrarily pick a time point TP in A such that current_time
belongs to TP's time bound;

3. Set TP's execution time to current_time and add TP to S;

4. Propagate the time of execution to its IMMEDIATE NEIGHBORS
in the distance graph;

5. Put in all time points TPx such that all negative edges starting from
TPx have a destination that is already in S;

6. Wait until current_time has advanced to some time between
        min{lower_bound (TP) : TP in A} and
        min{upper_bound} (TP) : TP in A};

7. Go to 2 until every time point is in S.
```

Figure 2-7 Pseudo-Code for the STN_DISPACHING Algorithm

Example 2-1:

Figure 2-8 shows a sample execution of a simple STN. The initial feasible execution windows for each timepoint are computed by considering the edges in the distance graph. A is the start of the plan and is assumed to be executed at T = 0. This gives us fix reference point. We can then compute the maximum feasible execution time for each

timepoint by computing the shortest paths from A to all other timepoints. Similarly, the smallest feasible lower bound is computed by computing the SDSP from all timepoints to A.

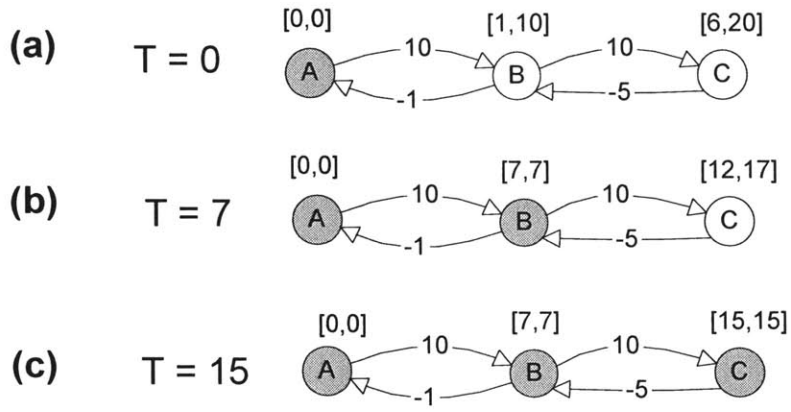


Figure 2-8 The network is properly executed by using local propagation

Timepoint A is executed at T = 0, as shown in Figure 2-8 (a). This execution time needs to be propagated to all neighboring timepoints through the constraints. The upper bounds of the execution windows are updated by forward propagation and lower bounds are updated by backward propagation. Consider the forward edge AB. It imposes an upper bound constraint on B with respect to A. Specifically, $T_B - T_A \leq 10$. Therefore, once we know that $T_A = 0$, we can deduce $T_B \leq 10$. This is an example of forward propagation. Similarly, the edge BA imposes the constraint that $T_A - T_B \leq -1$. Therefore, when we know that $T_A = 0$, we can derived the constraint $T_B \geq 1$. This is an example of backward propagation along a negative edge. However, this propagation provides no new information because we already knew that $T_A = 0$ when we computed the initial execution windows. In general, propagation does not always impose more restrictive bounds on the execution windows.

In general, say we execute timepoint A and there exists a forward (outgoing) edge AB. This edge imposes a constraint $T_B - T_A \leq b(A,B)$. Once we know, T_A, we can deduce an upper bound on B of $T_B \leq T_A + b(A,B)$. Similarly, a backward (incoming) edge BA, imposes a constraint $T_A - T_B \leq b(B,A)$. Once, T_A is known, we can derive a new lower bound on B of $T_B \geq T_A - b(B,A)$.

The dispatcher now has the freedom to choose any execution time for B that falls between its execution window of [1,10]. However, recall that the dispatcher executes and schedules at the same time. Therefore, it must wait until the current time falls within the execution window. We say the timepoint is *alive* if the current time is between the timepoints execution window.

Suppose the dispatcher waits until T = 7 until it executes B, as shown in Figure 2-8 (b). Once it executes B, the dispatcher must propagate this execution time to its neighbors. It does not need to backward propagate the time through the positive edge AB nor forward propagate the execution time through the negative edge BA because it already used those constraints to select the time for B. However, it does need to propagate its execution time to C. Specifically, it needs to propagate through edges BC and CB in order to update the execution window of C. Specifically, C's new upper bound is $T_B + b(B,C)$ or $7 + 10 = 17$, and C's new lower bound is $T_B - b(C,B)$ or $7 + 5 = 12$. The dispatcher is now free to choose any execution time between [12,17]. Figure 2-8 (c) shows the case where the dispatcher waits until T = 15 to execute C. □

Example 2-2:

In the previous example, we executed B before C. However, if we executed C before B, this would cause a problem. For example, consider the execution sequence shown in **Error! Reference source not found.** Timepoint A is executed at T = 0 as before. However, the dispatcher waits until T = 8. At which point both B and C are alive. The dispatcher mistakenly chooses timepoint C for execution. This propagates a new upper bound to B, via the edge CB, and a new lower bound to B, via BC. The new execution window for B is now [2,3]. Choosing any time between [2,3] is consistent; however, it is in the past! In order to prevent the dispatcher from making this mistake we impose an enablement condition on each timepoint. The enablement conditions are derived by considering the negative edges in the distance graph. Consider the edge BA. This impose a constraint that $T_A - T_B \leq -1$ or $T_B - T_A \leq 1$. Hence B must always be executed at least 1 time unit after A.

In general, a timepoint B is *enabled* only if all timepoints that must precede B have been executed. The set of *enablement timepoints* are all timepoints A for which there exist an outgoing negative edge BA. A timepoint can only be executed if it is both alive and enabled.

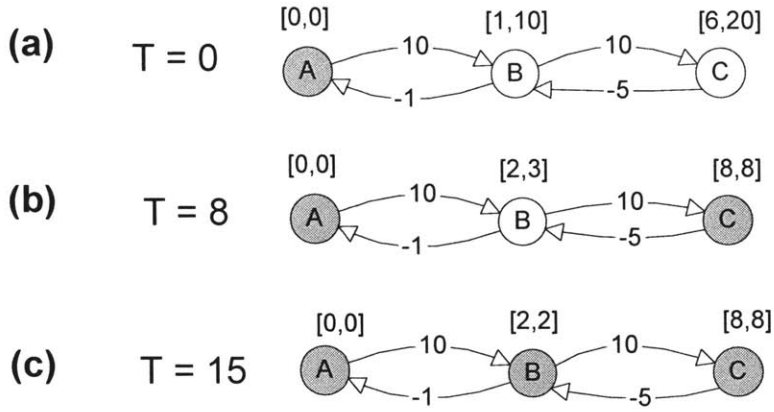
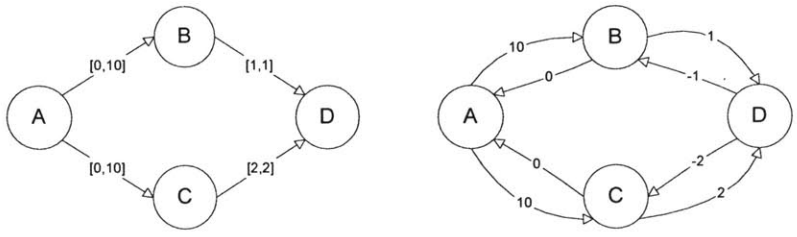


Figure 2-9 This network is improperly executed because the dispatcher did not respect the enablement conditions.

Example 2-3:

The dispatcher may still run into problems, even when the dispatcher only executes enabled and alive timepoints. The temporal constraints in the distance graph explicitly express the pair wise ordering constraints. However, there may exist implicit ordering constraints that need to be exposed before the dispatcher can properly execute the network. Consider the following STN and associated distance graph shown in Figure 2-10. After executing A at T = 0, both timepoint B and timepoint C are alive and enabled. However, timepoint C must be executed before B. Specifically, C must be executed exactly 2 time units before D and B must be executed exactly 1 time unit before D, therefore, C must be executed 1 time unit before B. If the dispatcher chooses to execute B before C, then the dispatcher will fail. Note that these implicit ordering constraints exist for networks even it they do not contain rigid constraints.



(a) Simple Temporal Network **(b)** Associated Distance Graph

Figure 2-10 There is an implicit temporal ordering between B and C. Specifically, B must be??? C must be executed exactly 1 time unit before B.

[Muscettola 1998a] showed that any consistent STN it can be transformed into an *equivalent dispatchable graph.* If the graph is dispatchable, it can always generate a consistent schedule by using local propagation to future timepoints. [Muscettola 1998a] showed that an STN can be converted into a equivalent dispatchable graph by first computing the associated distance graph then computing the All-Pair Shortest-Path graph (APSP-graph) of this distance graph. Note that APSP-graph is called a d-graph by [Dechter 1991]. Furthermore, given a dispatchable graph, upper bounds only need to be forward propagated along outgoing non-negative edges and lower bound only need to be backward propagated along incoming negative edges. Computing the All-Pairs Shortest Path (APSP) graph performs two tasks: 1) it compiles the temporal constraints, such that all implicit constraints are exposed, and 2) is exposes the enablement conditions of the distance graph.

The APSP graph is dispatchable; however, the dispatcher must perform N propagation every time it executes a timepoint. For large graphs this can be computationally expensive and may prohibit real-time execution. In order to reduce the propagation cost at execution time, the APSP is trimmed of all redundant edges. An edge is redundant if in all possible executions, there exists another edge that always propagates a tighter bound. The two cases are shown in Figure 2-11. [Muscettola 1998a] showed that the dominated (redundant) edges can be removed without adversely affecting the ability of the dispatcher to dynamically execute the network.

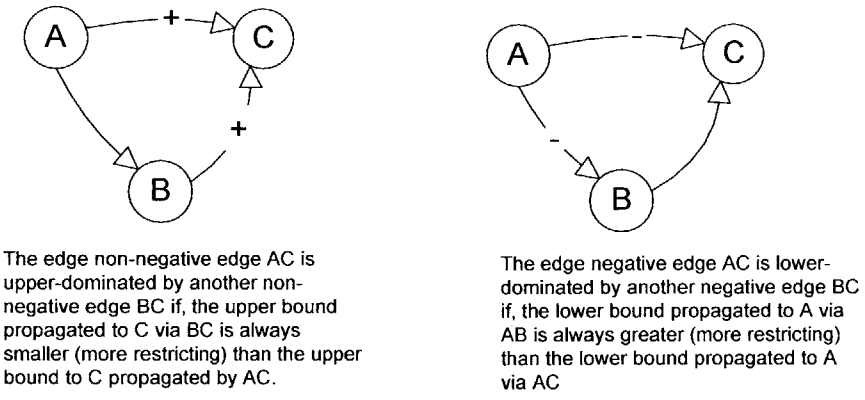


Figure 2-11 Definitions of upper dominance and lower-dominance

[Muscettola 1998a] showed that an edge is only dominated edge satisfies the triangle rule. We use notation |AB| is the shortest path distance as constructed by the APSP-graph.

Triangle Rule [Muscettola 1998a] *Consider a consistent STN where the associated distance graph???*

(1) A non-negative edge AC is upper-dominated by another non-negative edge BC if and only if $|AB| + |BC| = |AC|$
(2) A negative edge AC is lower-dominated by another negative edge AB if and only if $|AB| + |BC| = |AC|$

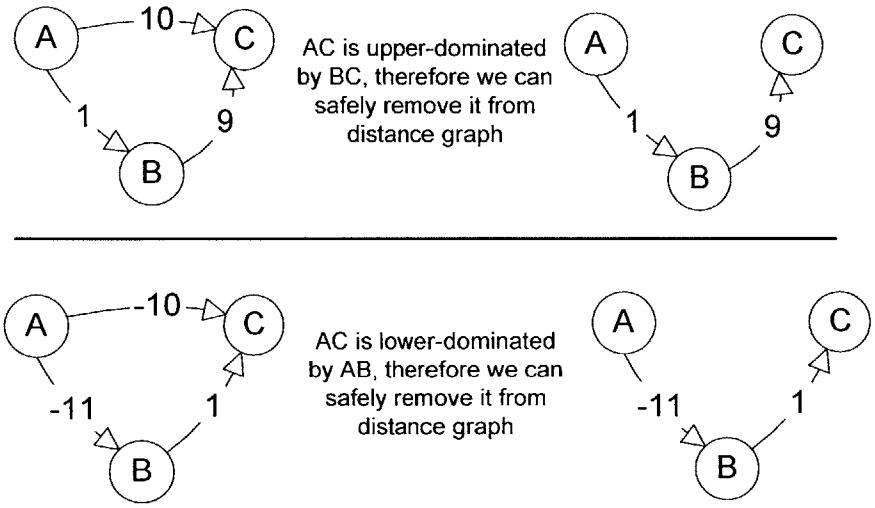


Figure 2-12 (top) An example of upper-dominated edge AC (bottom) An example of a lower-dominated edge AC.

The APSP-graph can be converted into a *minimal dispatchable graph* by applying the filtering algorithm introduced by [Muscettola 1998b]. A minimal dispatchable graph is a dispatchable graph that contains the fewest number of edges. The minimal dispatchable graph enables the dispatcher to perform efficient execution. The pseudo-code for the filtering algorithm is given in Figure 2-14. Note that one edge can dominate one another as shown in Figure 2-13. This case only occurs when the intersecting edges are related by a rigid set of edges.

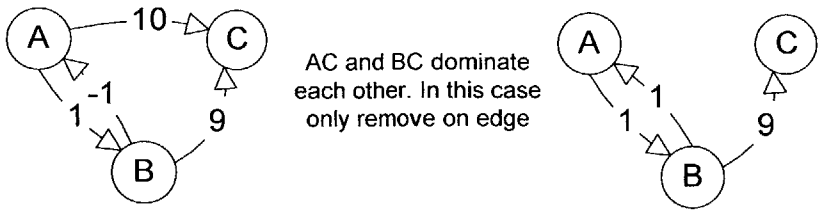


Figure 2-13 Mutual Dominance Example

```
function FILTERING_ALGORITHM ( G )
Input: A dispatchable APSP-graph G
Output: A minimal dispatchable graph
1  for each pair of intersecting edges in G
2      if both dominate each other
3          if neither is marked
4              arbitrarily mark one for elimination
5          end if
6      else if one dominates the other
7          mark dominated edge for elimination
8      end if
9  end for
10 remove all marked edges from graph
11 return G
```

Figure 2-14 Pseudo-Code for Filtering Algorithm

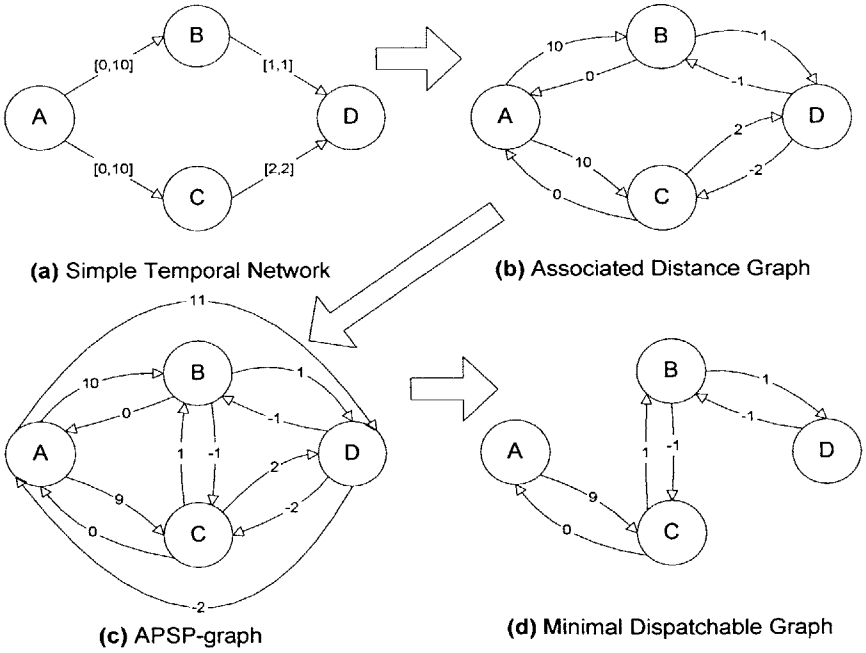


Figure 2-15 Basic Steps to STN Reformulation

The pseudo-code for the basic reformulation algorithm, which converts an arbitrary STN into a minimal dispatchable graph, is given in Figure 2-16 [Muscettola 1998a]. The steps of the reformulation algorithm are illustrated in Figure 2-15. First the STN is converted into the distance graph. Then APSP-graph is computed from the distance graph. Recall, the APSP-graph is dispatchable; however, typically contains many redundant edges. These redundant edges are removed by applying the filtering algorithm. Recall that the filtering algorithm is based on the triangle rule.

```
function BASIC_STN_REFORMULATION(G)
Input: A STN G
Output: A minimal dispatchable distance graph M
1. Convert the STN, G in to a distance graph D
2. Compute the of APSP-graph, A, from D via Floyd-Warshall
3. Apply Filtering Algorithm to A, compute minimal dispatchable
graph M.
4. return M
```

Figure 2-16 Pseudo-Code for Basic STN Reformulation Algorithm

The problem with the basic reformulation algorithm is that it causes an intermediate graph explosion. Specifically, the APSP graph requires $O(N^2)$ space. Furthermore, the basic STN reformulation algorithm requires $\Theta(N^3)$ time to run the filtering algorithm [Tsarmardinos 1998] presented a more sophisticated ``fast" reformulation algorithm that alleviates both these problems. The fast algorithm interleaves the APSP computation with the edge trimming elimination, such that the full APSP never needs to be built. Furthermore, it only requires linear space and has a time complexity comparable to Johnson's algorithm. This is referred to as the fast reformulation algorithm. The six main steps of the fast algorithm are shown in Figure 2-17. See [Tsarmardinos 1998] for details on the fast STN reformulation algorithm.

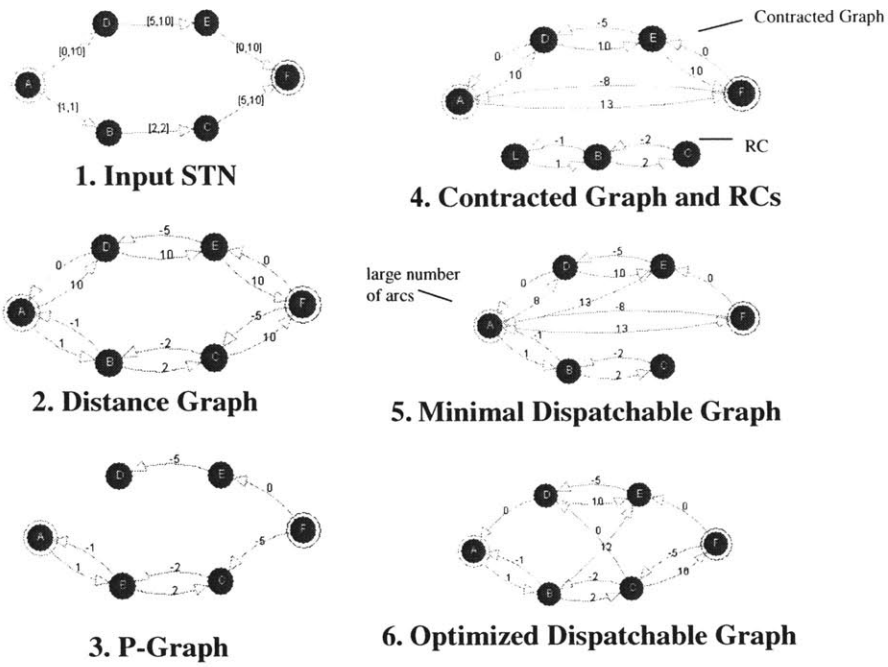


Figure 2-17 Example of the Fast STN Reformulation Algorithm

2.5 Simple Temporal Networks with Uncertainty

A STNU is a temporal constraint graph similar to a STN that contains an explicit model of uncertainty [Vidal 1996]. The edges of the STNU fall into one of two categories: the first are those representing temporal constraints that specify the allowable time that events are permitted to occur, and are called *requirement links*. These are exactly the same as STN constraints. The second category of edges represents uncontrollable activity durations, and are called *contingent links*. The timepoints that terminate contingent activities are controlled by nature and are called *contingent timepoints;* all other nodes are considered *executable timepoints.* The key point is that only a subset of nodes can be controlled by the executive and we call this network partially controllable.

The formal definition is of an STNU is provided below.

Definition 2-3 (STNU)*: A STNU is a 5-tuple < N, E, l, u, C >, similar to an STN, where N is a set of timepoints, E is a set of edges and* $l : E \rightarrow \Re \cup \{-\infty\}$ *and* $u : E \rightarrow \Re \cup \{+\infty\}$ *are functions mapping the edges to lower and upper bound temporal constraints. The STNU also contains C, which is a subset of the edges that specify the contingent links, the others being requirement links. We assume* $0 < l(e) < u(e)$ *for each contingent link.*

A Temporal Plan Network with Uncertainty (TPNU) is a plan similar to a TPN except the activities are constrained by an STNU. For example consider the TPNU shown in Figure 2-18. Rover2 is able to control when it stops charging its battery; however, rover1 cannot control the exact duration of its drive-to(rock1) activity.

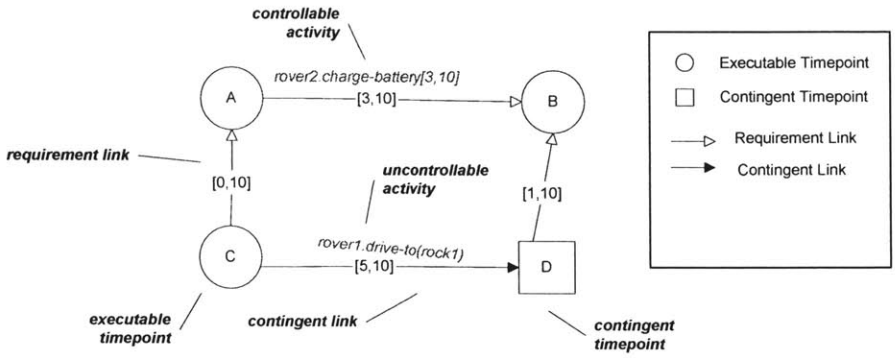


Figure 2-18 Anatomy of a TPNU

Every STNU has an associated Distance Graph with Uncertainty (DGU). The conversion of a STNU into a DGU is similar to the process of converting a STN into a distance graph. The DGU is formed by replacing each requirement link with two requirement edges and each contingent link is replaced by two contingent edges.

2.6 Summary

In this chapter we showed how to reformulate plans constrained by an STN for dynamic execution and introduced STNUs. In the next chapter, we introduce our Hierarchical Reformulation algorithm.

3 Hierarchical Reformulation Algorithm

3.1 Introduction

This chapter is the first of two technical chapters that serve to describe a coordinated, distributed executive. This distributed executive is novel in its ability to dynamically schedule tasks for multi-agent plans that contain both temporal uncertainty and communication constraints. Recall that an executive is composed of two components: a centralized reformulator and a distributed dispatcher. This chapter describes the reformulator, which prepares the multi-agent plan for execution by the dispatcher. The reformulator partitions a multi-agent plan into a set of decoupled group plans such that the agents that participate in each group plan can properly synchronize their activities, without communication outside their group. Within each group, the agents are allowed to communicate in order to adapt to runtime execution uncertainty.

In general, a distributed dispatcher requires communication to dynamically execute a plan. Recall that a dispatcher is an online dynamic scheduling algorithm that exploits the temporal flexibility of the plan, by waiting to schedule events until the last possible moment. In this least commitment execution strategy, the dispatcher schedules and dispatches the tasks simultaneously, rather than scheduling the tasks prior to execution. This dynamic execution strategy allows the agents to adapt to runtime uncertainty at the cost of some online computation. In the centralized case, the dispatcher must propagate the execution times of each event towards future events, every time an event is executed. This propagation enables the dispatcher to select consistent execution times for these future events. In the distributed case, this propagation translates into communication. If the agents are unable to keep in constant communication with one another, the agents may fail to properly execute the plan.

The simplest, although most restrictive, way to deal with communication limitation is to completely fix the schedule prior to execution. Pre-scheduling the activities removes the need for propagation, and hence removes the need for communication. However, in order to be robust to the uncertainty of uncontrollable events, the agent may need to be overly conservative about scheduling the time of execution of the activities, thus degrading the performance of the overall system. In other words, fixing the schedule prior to execution may require the agents to wait around when they could be doing something useful instead. Even beyond the issue of performance, there exists the question of viability. If a plan requires tight synchronization with respect to uncertain outcomes, then fixing the schedule will not work. In this case, a dynamic execution strategy is required in order to enable the agents to adapt to the uncertain outcomes. For example, consider a scenario where you plan on getting on a bus. This plan consists of two activities, going to the bus stop and boarding the bus. It may be possible to pre-schedule the time when you starting going to the bus stop; however, in order to successfully board the bus, you need to adapt the uncertainty of the bus' arrival time. Simply walking forward to board the bus at some pre-scheduled time is not a wise

execution strategy. In order use a dynamic execution strategy, the plan needs to retain some temporal flexibility.

This chapter presents a novel hierarchical reformulation algorithm that mitigates the need for communication, while retaining much of the temporal flexibility inherent in the plan, in order to dynamically adapt to uncertainty. It preserves the flexibility in places where tight coordination is required and fixes the schedule in places where the agents only require loose coordination.

In order to resolve which events should be scheduled dynamically and which events should be scheduled statically, we need to consider two factors: (1) when is a dynamic execution strategy required and (2) when is it possible. In general, a dynamic execution strategy is required when the agents have tight timing constraints between coordinated activities and a dynamic execution strategy is typically possible when agents can communicate. Fortunately, a dynamic execution strategy is possible when it is needed. We make the observation that communication availability tends to be synergistic with need. For example, robots tend to require tight coordination when they are close together and communication tends to be available when the robots are in close proximity.

In this thesis, we divide the full multi-agent plan into a set of tightly coordinated clusters. We assume that the agents that participate in these plan clusters are free to communicate with one another. We refer to these clusters as *communication clusters*. Note that we make no assumptions on the ability of the agents to communicate outside of their communication cluster. Given a clustering we organize the multi-agent plan into two layers, as shown in Figure 3-1. This two-layer plan consists of a *mission plan* and a set of *group plans*. The group plans specify tight coordination between a set of agents (one corresponding to each communication cluster) and the mission plan specifies loose coordination between the group plans. The mission plan uses a simplified abstraction for each group plan that hides the details of the group plan This encapsulation enables the reformulator to reason about the overall mission without getting into the details of the group plans.

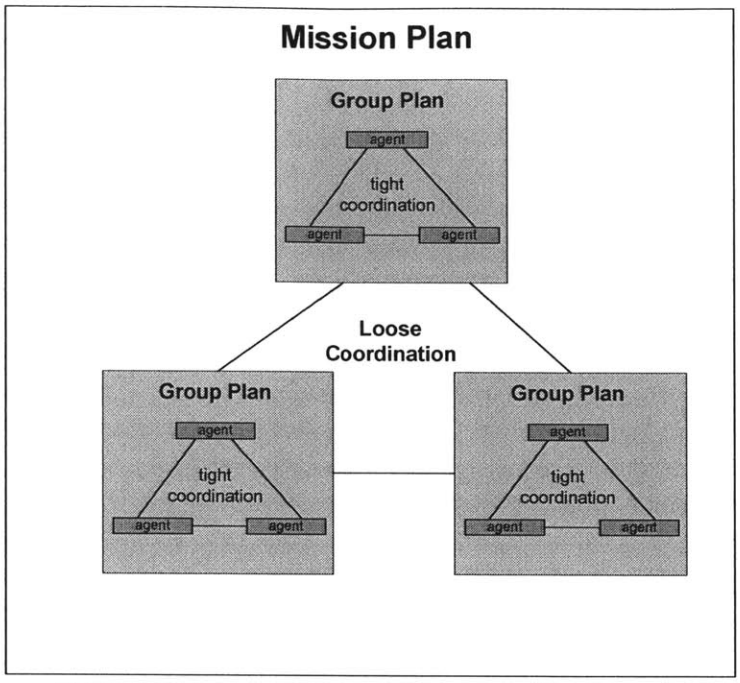


Figure 3-1 A two-layer multi-agent plan consists of a mission plan and set of group plans. The mission plan specifies loose coordination between a set of tightly coordinating group plans.

We call the set of agents that participate in each group plan simply a *group*. Each agent can only participate in one group at a time, but an agent may participate in multiple groups (group plans) over the lifetime of the mission. For example, in a Mars exploration scenario, a robot specializing in moving heavy objects may participate in some science gathering activity by turning over rocks early in the mission and then be used to clear a path in some exploration activity later in the mission.

The goal of the reformulator is to transform the two-layer plan into a form such that each group plan can be dynamically scheduled, without requiring the agents to communicate outside their group. This enables the groups to dynamically execute their plans in cases where inter-group communication is unavailable, unreliable or costly.

For example, consider the autonomous Mars exploration scenario illustrated in Figure 3-2. The robots are organized into two groups, a science group, and an exploration group. The science group consists of a set of science rovers specializing in data acquisition, along with a tethered blimp, which provides aerial reconnaissance for the science rovers. The science group's job is to take samples at the science site. The exploration group consists of a set of agile rovers used to explore future science sites. The groups will need to coordinate their intra-group activities more tightly than their inter-group activities. The two groups may need to periodically rendezvous to share data; however, we can expect the plan to be fairly flexible about when this occurs. Furthermore, it is reasonable to expect reliable communication within each group, because of the group member's proximity; however, communication between the groups

may be costly or unavailable, as the science team explores regions far from the current science site.

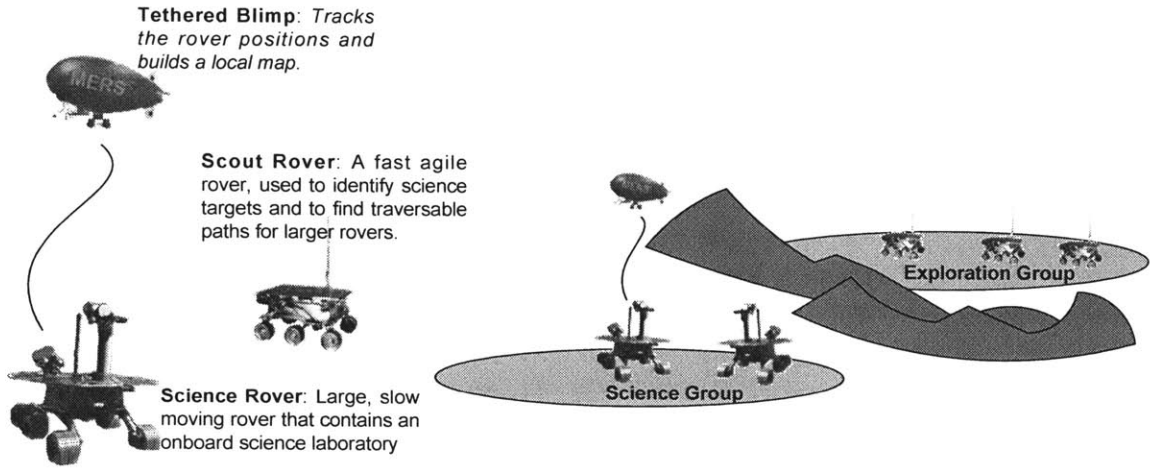


Figure 3-2 Heterogeneous Robotic Group Scenario

The science group and exploration group must cooperate to complete the mission. The goal of the reformulator is to enable each group to work independently in order to be robust to inter-group communication limitations.

The high level overview of the reformulator is shown in Figure 3-3. First it transforms the multi-agent plan into a two layer multi-agent plan (Figure 1-3a). Then the reformulator applies the Hierarchical Reformulation (HR) algorithm, which converts the two-layer plan into a set of decoupled dispatchable group plans. The HR algorithm operates on both layers of the multi-agent plan in this reformulation. The HR algorithm decouples each group plan by applying a variant of the strong controllability algorithm [Vidal 2000] to the mission plan (Figure 3-3b) and applies the fast dynamic controllability algorithm to each group plan (Figure 3-3c). The dynamic controllability algorithm prepares the group plans for the dispatcher. Finally, the reformulator applies an edge trimming algorithm to the dispatchable group plans (Figure 3-3d). The edge trimming algorithm removes the redundant constraints from the dispatchable plan in order to improve dynamic scheduling. The final form of each group plan is called a *minimal dispatchable group plan.*

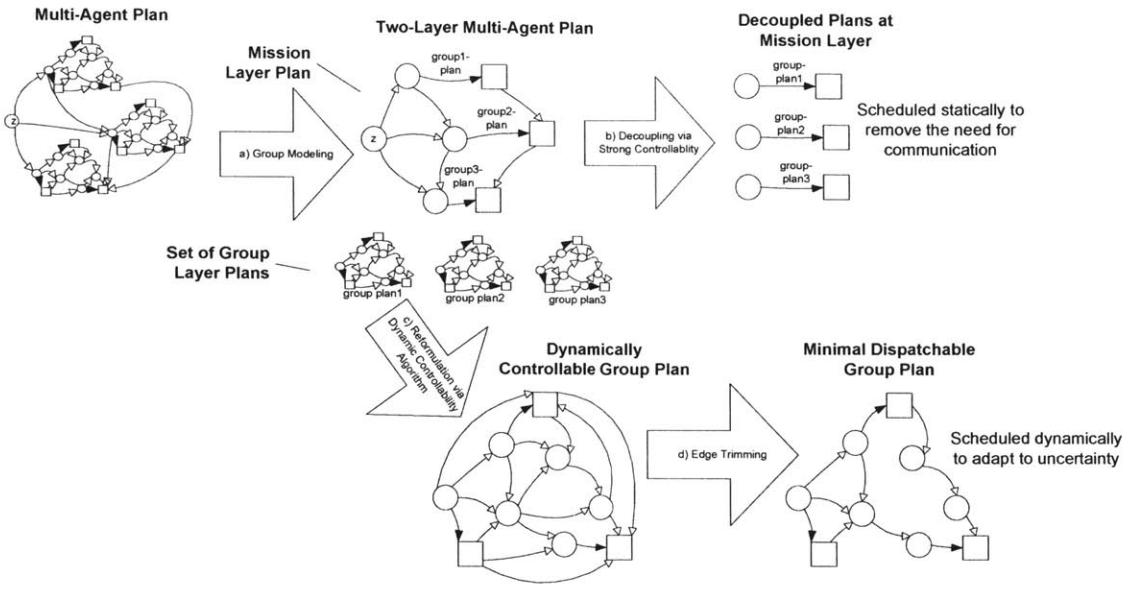


Figure 3-3 Overview of the Hierarchical Reformulation Algorithm

The outline for this chapter is as follows. Section 3.2 discusses the communication assumptions used in this thesis. Section 3.3 reviews the existing controllability theory and discusses the general problem of finding a viable dynamic multi-agent execution strategy for plans that contain both uncertainty and communication constraints. This new problem is called *communication controllability*. In Section 1.4 we formally introduce the two-layer Multi-agent Temporal Plan Networks with Uncertainty (two-layer MTPNU) and show how to specify these plans using a variant of the Reactive Model-Based Programming Language (RMPL). In Section 1.5 we review the strong controllability algorithm presented by [Vidal 2000] and present the decoupling algorithm based on this strong controllability algorithm. Finally, in Section 1.6 we present our novel Hierarchical Reformulation (HR) algorithm. Some of the details of the HR algorithm are deferred to Chapter 4. Specifically, Chapter 4 presents a new fast dynamic controllability and the edge trimming algorithm.

By the end of the chapter the reader should: 1) understand the definition of communication controllability, 2) know how to model multi-agent plans that involve temporal uncertainty and communication limitation as a two-layer MTPNU, and 3) know how to reformulate a two-layer MTPNU into a set of decoupled, minimally dispatchable plans using the HR algorithm.

3.2 Communication Assumption

In general, agents require communication if they participate in cooperative activities. If communication is always both available and reliable, programming these distributed systems would be a lot less complex; however, most real distributed systems must cope

with communication limitations. Specifically, real distributed systems must deal with hardware failures, and lost or dropped messages. Furthermore, mobile robots that use radio communication must cope with a limited communication range and physical obstacles that occlude communication pathways. Most of us have experienced these limitations first hand, when using cell phones. In this chapter, we address the problem of dynamically scheduling tasks in the presence of communication limitations.

This chapter makes the important assumption; when communication is available, it is reliable. We assume that the agents can achieve reliable communication by using some existing communication protocol, such as TCP/IP. Our focus is on multi-agent systems where communication is limited, yet predictable. For example, consider two rovers exploring Mars. In this domain, the rovers will be able to reliably communicate when in close proximity; however, when the rovers are separated by far distances or by some obstruction, the rovers will lose this reliable communication. Thus, depending on the relative position of each rover, it is possible to determine if communication is available.

3.3 Communication Controllability

This section formally defines the problem of verifying if there exists a successful multi-agent execution strategy for plans that contain both an explicit model of uncertainty and communication limitations. In doing so, this section develops several supporting definitions used throughout this thesis.

Previous work [Vidal 2000] has defined a set of controllability properties for STNUs. This controllability work has been concerned with the ability to generate an execution strategy for plans that contain uncertainty. Of particular interests is the ability to dynamically schedule these uncertain plans [Morris 2001]. In this section, we extend the notion of dynamic controllability to include the ability to cope with communication limitations as well as uncertainty. Plans for which it is possible to dynamically schedule by a set of agents that contain communication limitations between one another are called *communicationally controllable.*

First we will quickly review the four types of controllability that were previously introduced by [Vidal 2000 and Morris 2001]. This will help place the derivation of communication controllability in the context of previous work.

3.3.1 Primary Types of Controllability

Recall that a partially controllable plan is constrained by a Simple Temporal Network with Uncertainty (STNU). A STNU is a temporal constraint network that contains a set of timepoints and a set of temporal constraints, specifying the valid execution times of the timepoints. The timepoints of an STNU fall into two types: executable timepoints and contingent timepoints.[1] *Executable timepoints* are scheduled by the agent, while the times

[1] For a more thorough discussion of STNUs, refer to Chapter 2.

of the *contingent timepoints* are controlled externally rather than by the agent. Thus the execution times of the contingent timepoints are observed, rather than scheduled. The links[2] that connect the timepoints of the STNU also fall into two categories: contingent links and requirement links. A *contingent link* models the uncontrollable duration associated with an uncertain activity, and a *requirement link* represents a constraint on the duration between two timepoints.

Example 3-1:

Consider the contingent link associated with the *drive_to* activity on the left of Figure 3-4. The contingent link specifies that the duration of the *drive_to* activity is uncertain and will last between 5 and 10 time units. Thus, the contingent timepoint, e1, will be executed between 5 and 10 time units after the agent executes the executable timepoints, s1. A requirement link imposes a temporal constraint on the duration between two timepoints. Consider the requirement link associated with the *spectrometer_reading* activity at the right of Figure 3-4. The rover must perform the spectrometer_reading activity between 10 and 20 minutes.

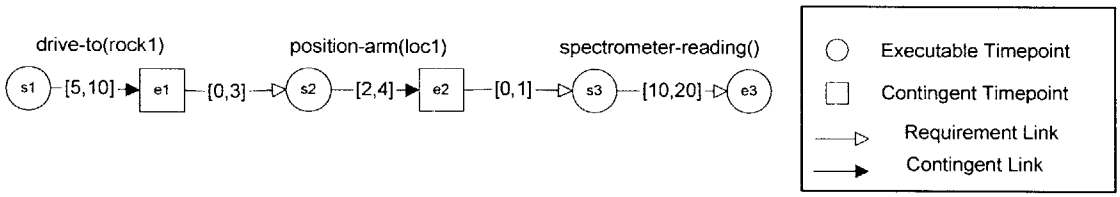


Figure 3-4 The duration of drive_to(rock1) and position_arm(loc1) activities are uncertain; however, the duration of spectrometer_reading() is determined by the agent.

Overall, Figure 3-4 denotes a partially controllable plan that contains three activities. The duration of both the *drive-to* and *position-arm* activities is uncertain; however, the duration of the *spectrometer-reading* activity is determined by the rover. The rover's job is to execute each timepoint in the plan in a fashion that is consistent with the temporal constraints of the STNU. The rover has two choices. It may either schedule the timepoints offline, hence fixing the schedule prior to execution, or it may dynamically schedule the plan. We are interested in the latter.

Informally, an *execution strategy* is a policy for scheduling the timepoints in the plan. In the case of an STNU, an execution strategy only specifies a policy for scheduling the executable timepoints. Recall that the contingent timepoints are not under the control of the agent. Furthermore, an execution strategy is called *viable* if the schedule it produces is temporally consistent for all possible *situations*. Here we use the term situation to

[2] Links contain both a lower and upper bounds, whereas, edges only contain a single upper bound constraint.

mean some outcome of uncertain events. Furthermore, we call an execution strategy *dynamic* if the execution and scheduling of the timepoints occurs simultaneously. In this section we are interested in viable dynamic execution strategies. From now on we will use the term dynamic execution strategy for a viable dynamic execution strategy, unless otherwise stated.

Designing a dynamic execution strategy for the plan in Figure 3-4 is straight forward. One possible viable dynamic execution strategy is as follows:

> 1. Execute the start of drive_to(rock1) activity at Time = 0. 2. After observing that the rover reached *rock1*, immediately start positioning the arm. This satisfies the constraint that the position arm activity must start between 0 and 3 time units after getting to the rock. 3. Immediately after observing that arm is positioned at *loc1*, start taking the spectrometer reading. This satisfies the constraint that the spectrometer reading must occur between 0 and 1 time units after positioning the arm. 4. Stop the spectrometer reading activity after 15 time units have passed. This satisfies the constraint that the spectrometer_reading activity lasts between 10 and 20 time units.

For plans constrained by more complex STNU topologies, designing a viable dynamic execution strategy becomes much more complex. In these cases we need a principled method to generate an execution strategy. This is why controllability algorithms have been developed. In our discussion of controllability the STNU is the important part of the plan, as such, we refer to the controllability of the STNU to be short for the controllability of the plan.

Informally, a STNU is controllable if there is a viable execution strategy for scheduling the executable timepoints. Controllability refers to an agent's ability to "control" the consistency of the schedule, despite the uncertainty in the plan. In general, the more temporal flexibility a plan contains, the more likely that the plan is controllable. Controllability is a battle between flexibility against uncertainty. However, the flexibility must be in a place where an execution strategy can use it.

Abstractly, there are three fundamental problems related to executing plans with uncertainty, however, in practice they are all related. First there is a problem of determining if there exists a viable execution strategy (controllability verification). After verifying the controllability of a plan, there still remains the problem of actually generating an execution strategy. Generating an execution strategy boils down to compiling the temporal constraints (reformulation) into a form that a dispatcher can readily use. If this reformulation succeeds, the plan is said to be dispatchable. The last problem is to execute the plan using a dispatcher (dispatching). Verification, reformulation, and dispatching are all interrelated. Verification is done by reformulating the plan, meaning that if the reformulation succeeds, then the plan is controllable and

dispatchable. Furthermore, the reformulation algorithm is done with a specific dispatching algorithm in mind.

Controllability was first introduced by [Vidal et. al. 1996]. There are three levels of controllability: strong, dynamic, and weak. Algorithms for checking weak and strong controllability were developed soon there after. A dynamic controllability checking algorithm took a few more years. [Morris 1999] first introduced a new controllability property, called waypoint controllability, which generalizes strong and weak controllability, as a first attempt to make a dynamic controllability checking algorithm. Then [Morris 2000] presented a algorithm to verify, reformulate and dynamically execute an STNU.

The three primary levels of controllability differ in that they make different assumptions on when the uncertain durations in the plan are observed. For strong controllability we assume no uncertain durations are known when the scheduling decisions are made. In dynamic controllability, we assume that the agent knows the outcomes of uncertain durations as they are completed. Therefore, agents can only use uncertain durations that happened in the past to make its scheduling decisions. In weak controllability we assume that all uncertain durations are known at the time of scheduling.

The amount of information known to the agent increases from strong to dynamic to weak control. Therefore it should come as no surprise that the three primary forms of controllability follow an implication rule. Specifically, [Morris and Vidal] have formally shown that, if a network is strongly controllable, then it is dynamically controllable, and if a network is dynamically controllable, then it is weakly controllable. Also note that both strong and dynamic controllability of a STNU can be checked in polynomial time; however, weak controllability is a co-NP-complete problem [Vidal, 1999].

Waypoint controllability, introduced by [Morris, 1999], combines the properties of strong and weak controllability. Waypoint controllability applies to plans for which a subset of the timepoints are designated as *waypoints*. Waypoint controllability refers to an execution strategy that schedules the waypoints using a strong control strategy and the remaining timepoints in a weakly controllable fashion. In other words, a waypoint controllable execution strategy schedules the waypoints prior to knowing the uncertain outcomes, then once all of the uncertainty is resolved, the remaining timepoints are scheduled. Waypoint controllability is of limited utility because it makes no guarantees on the ability to dynamically execute that plan. Furthermore, the waypoint controllability checking algorithm runs in exponential time [Morris 2000].

Waypoint controllability is discussed because there is an interesting relation between the property that we seek in our two-layered approach and waypoint controllability. While waypoint controllability combines strong and weak controllability, the two layered approach presented in this thesis combines strong and dynamic controllability. Specifically, we show that a portion of the plan can be scheduled offline without knowing

the uncertain durations, while the remaining portion is scheduled dynamically. Recall, that we schedule a portion offline in order to compensate for lack of communication.

3.3.2 Formal Definition of Communication Controllability

In this section, we extend the definition of dynamic controllability to plans executed by a set of distributed agents that may not always be in constant communication. Here we are interested in the ability or inability of the agents to communicate the dynamically scheduled execution times, as required by a dynamic execution strategy, between one another. If a plan is dynamically executed on a set of distributed agents, inter-agent communication limitations may preclude the agents from propagating execution times. For example, consider a Mars exploration scenario where two rovers need to cooperate, yet one rover moves out of communication range from another rover, during the execution of the plan. In such a scenario, we are interested in knowing whether there exists a successful dynamic execution strategy to enable the rovers to cooperatively execute the plan. In this case, if we can devise a multi-agent execution strategy that is guaranteed to succeed, then we say that the cooperative multi-agent plan is *communication controllable.*

If the plan is strongly controllable, then the agents can cope with this information deficiency by fixing the schedule prior to execution; hence, eliminating the need for any communication at execution time. However, we are interested in the more general case in which the plans are not strongly controllable, and the agents must use a dynamic execution strategy. In this case, a plan is communication controllable only if the agents can make scheduling decisions with partial knowledge of the execution history.

In order to formally define communication controllability, we first describe how the plan is distributed among the agents. Distributed dynamic scheduling is a process in which multiple agents collaborate in order to dynamically schedule the timepoints of a multi-agent plan. Each agent takes ownership of a portion of the timepoints. An agent must schedule each timepoint it owns. The ownership of timepoints is defined by a *distribution.*

Definition 3-1 (Distribution): *Given a STNU, Γ=<N,E,l,u,C>, and a set of agents, A; a distribution, D, is a mapping, D: $N \rightarrow A$, such that each timepoint $x \in N$ is uniquely assigned to an agent $a \in A$. D(x), also written D_x, is the agent that owns timepoint x. Furthermore, the set of timepoints assigned to an agent, a, in distribution, D, is written N^a.*

Figure 3-5 shows a distribution of a STNU over two rovers. For simplicity, the time bounds are not shown on the links of the STNU. The gray timepoints are mapped to rover1, and the white timepoints are mapped to rover2.

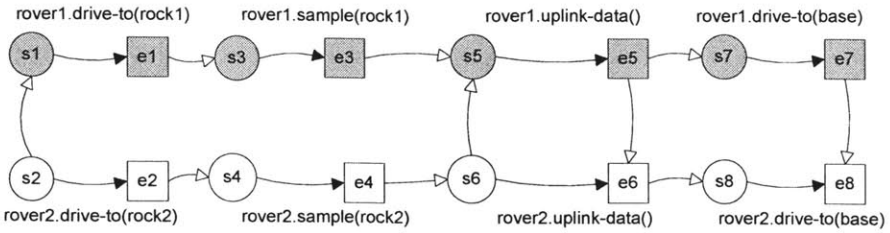


Figure 3-5 Distribution of a STNU. The timepoints of the STNU are uniquely assigned to an agent.

Next, for a distributed STNU, we need to know when the agents can communicate with one another. We introduce a *communication availability graph (CAG)* in order to model the communication availability between the agents. The communication availability graph combines a finite state automaton with a set of communication constraints. The CAG is composed of a set of agent states, a set of directed edges representing the criteria for state transition, and a set of undirected, *communication availability* edges. If a communication availability edge connects two states, then reliable communication is available between those states. For simplicity, the state of an agent is determined solely determined by monitoring the execution status of the timepoints in the associated STNU. Communication between two timepoints is then available, if a communication availability edge exists between the locations of the activities of the two timepoints.

For example, consider the communication availability graph shown in Figure 3-6, which is associated with the STNU given in Figure 3-5. The CAG contains two states for each rover. In this case, the states of the rovers represent their location. Rover1 (gray states) is either in the at-base state or at-rock1 state. Similarly, rover2 (white states) is either in the at-base state or at-rock2 state. A rover is always able to communicate with itself; therefore, there is a communication availability edge between states. Furthermore, we assume that the rovers are free to communicate with each other when they are both at base as indicated by the communication availability edge AC. However, when rover1 reaches rock1 it is no longer able to communicate with rover2(i.e. no communication availability edge BC nor BD) Similarly, rover2 is unable to communicate with rover1 when it reaches rock2 (i.e. no communication availability edge DA nor DB). Rover1 transitions from state A to state B when it executes timepoint e2 (in the TPNU), and transitions back to state A when it executes timepoint e7. Similarly, rover 2 transitions from state C to D when it executes timepoint e3, and transitions back to C when it executes timepoint e8.

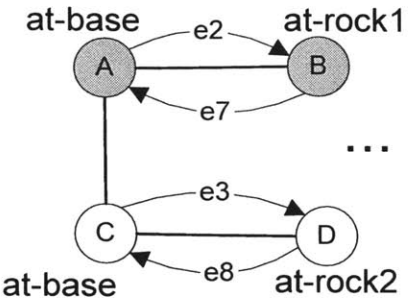


Figure 3-6 Communication Availability Graph The directed edges represent state transition criteria, and the undirected edges represent communication availability.

Definition 3-2 (Communication Availability Graph): *Given an STNU, $\Gamma = \langle N,E,l,e, C\rangle$, and a set of agents Q, a communication availability graph, $\Pi = \langle S,T,U\rangle$ of Γ and Q, is a graph where S is a set of states where each state is mapped to an agent A. Furthermore, T is a set of directed edges representing state transitions and U is a set of undirected communication availability edges. We say that reliable communication exists between states S_i and S_j if there exists an edge $U_{ij} \in U$.*

The purpose of introducing a communication availability graph is not to cover all the details of modeling communication between a set of distributed agent but rather just to put forth a rational model of communication.

Controllability is a property of a STNU that states whether there exists a policy for consistently scheduling the executable timepoints in any *situation*. Communication controllability is a property of a STNU that specifies whether there exists a multi-agent execution strategy for STNU that is constrained by a communication availability graph. In order to develop this concept, we first define the standard concept of a schedule.

Definition 3-3 (Schedule [Morris 2000]): *A schedule, T, for a set of timepoints, N, is a mapping $T: N \rightarrow \mathfrak{R}$, which maps each timepoint $x \in N$ to a scheduled time, where T(x), also written T_x, is the scheduled time of timepoint x.*

The dispatcher's job is to consistently schedule all of the timepoints contained in a STN. We call a schedule feasible if the assignments do not violate the simple temporal constraints in the STN.

Definition 3-4 (Feasible Schedule of a STN): *Given an STN, $\Gamma = \langle N,E,l,u\rangle$, a schedule of Γ, $T(\Gamma)$, is a schedule for all of the timepoints $n \in N$. This schedule is feasible if $l(XY) \leq T(x)-T(y) \leq u(XY)$ holds for each pair of timepoints $x,y \in N$, and for each link $XY \in E$.*

Recall that for STNUs, only a subset of the timepoints are scheduled by the dispatcher. Specifically, the agent only schedules the executable timepoints N_e, whereas, the agent observes the outcomes of the contingent timepoints, N_c. In order to explicitly define the uncertain outcomes we introduce a set of variables corresponding to the uncontrollable durations. These uncontrollable durations are constrained by the contingent links in the STNU. Here we make a distinction between the contingent link, which specifies the constraint on the uncontrollable duration and the variable that is assigned a value upon observing the uncontrollable duration. For example, see Figure 3-7.

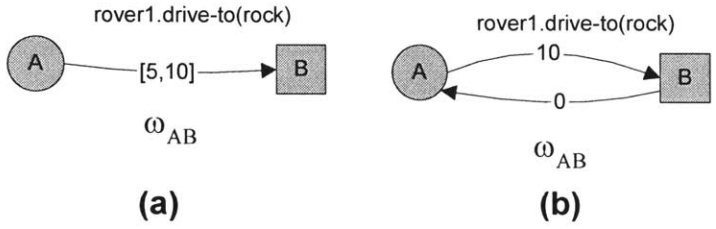


Figure 3-7 (a) The STNU contains a contingent link AB with bounds [5,10] constraining the uncontrollable duration ω_{AB} **(b)** The associated distance graph also contains the uncontrollable duration even though there is no contingent link.

Definition 3-5 (Uncontrollable Duration [Vidal 2000]): *Given an STNU, Γ =<N,E,l,u,C>, for each contingent link AB in Γ there exists an uncontrollable duration $\omega_{AB} \in [l(AB),u(AB)]$ which is a variable representing the duration between the timepoints A and B, whose outcome is uncertain.*

In order to talk about the possible outcomes of the uncontrollable durations, we define a *situation.* A *partial situation* is an assignment to the subset of the uncontrollable durations and a *complete situation*, also called a situation, is an assignment to all the uncontrollable durations. The assignments to the uncontrollable durations are done such that they respect the constraints imposed by the contingent links.

Definition 3-6 (Complete/Partial Situation [Vidal 2000]): *A situation is an assignment to the uncontrollable durations, i.e. $\omega_{XY} = d \in [\ l(XY), u(XY)]$. A partial situation is a partial assignment to the uncontrollable durations, and a complete situation, also called a situation, is a full assignment to the uncontrollable durations.*

In order to define a feasible schedule for STNUs, we use the concept of a *projection,* introduced by [Vidal 2000]. A projection is another way to represent a complete

situation. Given a complete situation, a projection transforms a STNU into a STN by transforming the contingent links into a set of rigid STN links whose lower and upper bound equals the value of the corresponding duration in the situation. Note that each uncontrollable duration, ω, may take on any value between the lower and upper bound specified by the corresponding contingent link. Therefore, a STNU defines a set of projections.

Definition 3-7 (Projection [Vidal 2000]): *Given an STNU, Γ = <N,E,,l,u,C>, a projection, Γ' = <N,E,l,u>', of Γ, is a Simple Temporal Network (STN), where each requirement link is replaced by an identical STN link, and each contingent link $e \in E$ is replaced by a rigid STN link, where the lower and upper bound is $[\omega, \omega]$, for some ω such that $l(e) \leq \omega \leq u(e)$. The projection of Γ is written $P(\Gamma)$.*

For example, consider the projection shown in Figure 3-8(b) of the STNU given in Figure 3-8(a). This projection is just one of many possible projections of the STNU. The projection shown in Figure 3-8(b) corresponds to a situation where the uncontrollable duration, ω_1, corresponding to the *rover1.drive-to(rock1)* activity is 7 time units, and the uncontrollable duration, ω_2, corresponding to the *rover2.drive-to(rock2)* activity is 10 time units.

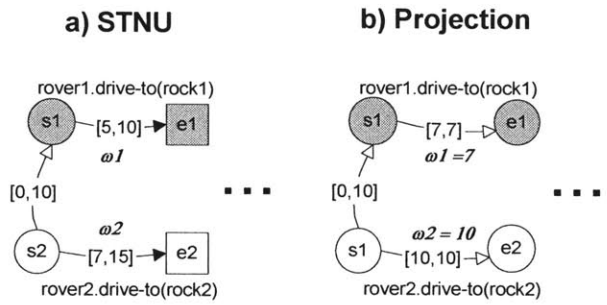


Figure 3-8 Example of a Projection of a STNU

The concept of projection is useful because it transforms the idea of a set of uncontrollable outcomes into data structure we are familiar with working with, i.e. an STN. The *feasible schedule of an STNU* is defined in terms of its possible projections. Specifically, a schedule of an STNU is feasible if the schedule corresponds to some feasible STN schedule of one of its projections.

Definition 3-8 (Feasible Schedule of an STNU): *Given an STNU, Γ = <N,E,,l,u,C>, a STNU schedule T of Γ, is a schedule for all of the timepoints $n \in N$. A feasible STNU schedule of Γ is a feasible STN schedule of one of Γ's projections, written $T(P(\Gamma))$.*

Note that the schedule includes both the executable and contingent timepoints.

Now we turn our attention to the process of generating a schedule of an STNU. An execution strategy specifies a policy for scheduling the controllable timepoints of the STNU for all possible projections (or situations). Furthermore, we call the execution strategy *viable* if the execution strategy generates a feasible schedule for all possible projections. By definition, a viable execution strategy will succeed, given it knows the projection prior to generating the schedule; therefore, the existence of a viable execution strategy is equivalent to saying that the STNU is weakly controllable [Morris 2000].

Definition 3-9 (Execution Strategy [Morris 2000]): *Given an STNU, Γ, an execution strategy $S: P \rightarrow T$ of Γ, is a mapping of projections, P, to schedules, T, for the timepoints of Γ. An execution strategy, S, is* ***viable*** *if the schedule, S(p), is consistent with Γ, for every projection $p \in P$.*

Communication controllability is inherently a multi-agent concept; hence, we will generalize our notion of schedules and histories to multi-agent systems. For a distributed multi-agent system, each agent is in charge of its portion of the plan. Specifically, each agent must schedule the timepoints that it owns, as specified by a distribution D. Therefore, we define *agent schedule* as the set of timepoint assignments made by that individual agent.

Definition 3-10 (Agent Schedule): *Given an STNU, $\Gamma = \langle N,E,l,u,C \rangle$, a set of agents A, and distribution D; an agent schedule, T^a, is a schedule for all timepoints owned by agent a. Furthermore, $T^a(x)$, also written T^a_x, is the scheduled time of individual timepoint x, owned by the agent a in a schedule T^a. Given an agent schedule for each agent $a \in A$, the schedule of Γ, $T(\Gamma)$, is the union of each agent schedule T^a, for all agents $a \in A$.*

Now we extend the concept of feasibility to agent schedules. The definition of a feasible agent schedule is analogous to the corresponding centralized concept.. An agent schedule is feasible if the agent's schedule is consistent with the temporal constraints of the full STNU.

Definition 3-11 (Feasible Agent Schedule): *Given an STNU, $\Gamma = \langle N,E,l,u,C \rangle$, a set of agents A, a distribution D, and an agent schedule, T^a; For each $a \in A$ the schedule T^a is feasible if and only if each timepoint $n \in N^a$ is consistent with the full set of temporal constraints in Γ. Furthermore, if the union of all agent schedules are feasible, then $T(\Gamma)$ is feasible.*

To define a communication limited, multi-agent strategy, we first extend the notions of execution strategies to a distributed multi-agent system. For a distributed multi-agent system, the schedule for the entire plan is generated by using a set of agent execution strategies, where each agent schedules the timepoints assigned to it by some distribution.

Definition 3-12 (Agent Execution Strategy and Multi-Agent Execution Strategy): *Given an STNU, Γ, a set of agents, A, and a distribution, D, an* ***agent execution strategy,*** *$S^a: P \rightarrow T^a$, for agent $a \in A$ maps the projections, P, to an agent schedule T^a. An agent*

execution strategy, S^a, is viable if the resulting agent schedule is feasible for all projections $p \in P$. A ***multi-agent execution strategy*** *uses a set of agent execution strategies, M, for all agents $a \in A$ to map projections, P, to schedules, $T = \bigcup_{S^a \in M} S^a(p)$. Furthermore, a multi-agent strategy, M, is viable if each agent strategy, $\bigcup_{S^a \in M} S^a(p)$ is viable.*

Note that a complete schedule, T, of the STNU, is the union of all agent schedules, T^a, generated by the corresponding agent execution strategy, S^a.

Generalizing from dynamic controllability, communication controllability pertains to a dynamic scheduling process in which each agent makes scheduling decision solely based on past outcomes. In general, each agent may use outcomes it directly observes or outcomes observed from other agents to make scheduling decisions. In the centralized case, a plan is dynamically controllable if an agent can create an execution strategy that schedules each timepoint x, knowing only the uncertain outcomes that happened prior to x. However, as we will see in Chapter 4, a viable dynamic execution strategy depends on the ability to propagate both scheduling decisions and uncertain outcomes to unscheduled timepoints. It is useful to define the portion of the full schedule that each agent can observe when it makes each scheduling decision and to separate this set into controllable and uncontrollable timepoints.

In the centralized case, when the agent dynamically schedules a timepoint x, the agent only has access to the uncertain durations that happened prior to scheduling the timepoint x. and scheduling decisions made prior to scheduling x. Collectively, this information is called the scheduling *history of timepoint x,* referred to as simply the history at timepoint x. This history is composed of both a *contingent history*, which is the set of contingent timepoint assignments made prior to x, and a *scheduled history*, which is the set of executable timepoint assignments made prior to x. To summarize, the history of x is just the portion of the schedule fixed prior to scheduling timepoint x.

Definition 3-13 (History): *Given an STNU, $\Gamma = \langle N,E,l,u,C \rangle$ and a schedule T for Γ,*

(a) *The* ***history*** *at timepoint x, H(T,x), is the assignment in T to timepoints $n \in N$, scheduled prior to timepoint $x \in N$.*

(b) The ***contingent history*** *at timepoint x, $H_c(T,x)$, is the assignment in T to contingent timepoints $n \in N_c$, scheduled prior to $x \in N$.*

(c) The ***scheduled history*** *at timepoint x, $H_s(T,x)$, is the assignment in T to executable timepoints $n \in N_e$, scheduled prior to timepoint $x \in N$.*

Communication controllability deals with agents that that contain communication limitations. The agents must in addition cope with the communication limitations imposed by the communication availability graph, Π. Due to communication limitations, one agent may not communicate its schedule to another agent at execution time. Hence, at any given time, each agent may only be able to know part of the history. Note that each agent's knowledge of the full schedule is dependent on both the schedule and on the

communication availability graph. Depending on when an agent schedules a timepoint, the value of the assignment may change, as well as its ability to communicate this scheduling decision to other agents. We introduce a concept of an agent's knowable history at one of its owned timepoints x, as the portion of the schedule that is knowable to the agent at the time of scheduling x. This is called the agent's knowable history at x. We break the knowable history into a knowable contingent history and a knowable scheduled history.

Definition 3-14 (Agent's Knowable History): *Given an STNU, Γ = <N,E,l,u,C>, a CAG, Π, a schedule T(Γ), a set of agents A, and distribution, D,*

(a) The agent a's ***knowable history*** *at timepoint x, written $H^a(T,\Pi,x)$, is the schedule of all timepoints $n \in N$, scheduled prior to timepoint $x \in N^a$ that could have been communicated given the communication availability Π.*

(b) The agent a's ***knowable contingent history*** *at timepoint x, written $H_c^a(T, \Pi,x)$, is the schedule of the contingent timepoints of $c \in H(T,\Pi,x)$.*

(c) The agent a's ***knowable scheduled history*** *at timepoint x, written $H_s^a(T,\Pi,x)$, is the schedule of executable timepoints $n \in H(T,\Pi,x)$.*

We are finally ready to formally define the *communication controllability* property, that specifies whether or not there exists a viable, dynamic, multi-agent execution strategy, for a distributed, partially controllable STNU, in the presence of communication availability imposed by a communication availability graph, Π. Recall that the complete multi-agent execution strategy is the union of each agent execution strategy. For this communication controllable there exists a viable dynamic multi-agent execution strategy, such that the schedule it generates is independent of future or unobservable events, where the implication in the definition below captures this independence.

Definition 3-15 (communication controllability): *Consider an STNU, Γ = <N,E,l,u,C>, distributed over a set of agents A, according to a distribution D, and constrained by a communication availability graph, Π. Then Γ is communication controllable, if there is a viable, dynamic, multi-agent execution strategy, given the communication availability specified by Π. That is there exists a viable agent execution strategy, S^a, where S(p) is defined as the union $S^a(p)$ over all agents, for each timepoint $x \in N$, such that:*

$$H^a(S(p1),\Pi,x) = H^a(S(p2),\Pi,x) \Rightarrow [S^a(p1)]_x = [S^a(p2)]_x,$$

for each agent $a \in A$, and for each pair of projections, p1 and p2, of Γ.

In other words, a STNU is communication controllable if there exists a viable, dynamic, multi-agent execution strategy for scheduling each executable timepoint in the presence of limited communication availability. This viable dynamic multi-agent execution strategy is composed of a set of viable, dynamic, agent execution strategies, where each agent's execution strategy only depends on a subset of the past to make a scheduling decision for each timepoint x. Recall that this subset of the past is called the

agent's observable history at timepoint x (as defined in Definition 3-14). The equation in Definition 3-15 states that if the agent's knowable history at timepoint x is the same in two separate projections (outcomes of uncertain durations), then there exists a viable agent execution strategy to generate the same schedule for timepoint x. Hence, the agent's execution strategy for timepoint x is only dependent on the agent's observable history at timepoint x.

One major problem is that for an arbitrary communication availability graph, there is no easy way to compute the agent's observable history for each timepoint. This is left for future work. In this thesis we simplify the problem by restricting the form of the communication availability graph. Specifically, we only consider plans that are partitioned into a set fully communicating clusters, where each cluster has a fully connected communication availability graph. Thus the knowable history for each agent is a total history of the communication cluster.

The problem is reduced to two simpler problems: 1) finding the schedule for the start of the group plan, only knowing the time of the start of the mission, and 2) generating a schedule for the timepoints in each group for which each agent knows the group plan's full history and the start time of the mission. These problems are still coupled, hence we further simplify the problem by imposing a two layer structure on the plan and solve the two problems with only very limited coupling between them. In the next section, we will precisely describe these two layer multi-agent plans, which enable us to simplify the communication controllability problem.

3.4 Two-Layer Multi-Agent Plans

This section precisely describes the two-layer, partially controllable, multi-agent plans that contain communication constraints. For simplicity, these plans are referred to as *two-layer plans*. This section also describes how to construct these two-layer plans. The two-layer plans are generated by either 1) *clustering* a fully elaborated plan, or 2) specifying the two layer plan using a variant of the Reactive Model-Based Programming Language (RMPL) [Williams 2003], [Kim 2001], called the Group Planning Language (GPL).

The two-layer plan consists of a mission plan and a set of group plans. The mission plan describes the high level structure of the mission. It specifies a set of constraints on a set of abstract group activities. The group activities are abstract, meaning the group activities are not executable primitives, but rather represent a place holder for a detailed group plan. Each group plan is executed by a set of agents, called a group. The constraints in the mission plan include both temporal and communication constraints. The temporal constraints serve two purposes, first they model the uncertain duration of each group activity and second, they constrain the valid execution times of the start and end of each group activity. The communication constraints specify when the groups can communicate with each other. The group plans specify the details of each group activity. Specifically, each group plan contains a set of activities to be executed by a set of agents. Each group plan also contains a set of internal temporal and communication constraints.

The temporal constraints of a group plan are used to model uncertain duration and place temporal constraints to restrict the feasible execution times of the group's agent activities. The communication constraints specify when the agents can communicate with other members within the group.

The two-layer plan encapsulates each group plan within a simple abstraction, called a *group plan macro,* or macro for short. A macro consists of an abstract group activity, an executable start timepoint, a contingent end timepoint, and a contingent link connecting the start and end timepoint, which specifies a range of possible durations of the group activity. The contingent link in the macro models the possible duration of the group activity. For each macro in the mission plan there is an associated group plan and vice versa. Given a macro, the corresponding group plan is returned by calling a GET_GROUP_PLAN function and similarly, given a group plan, the corresponding macro in the mission plan is returned by calling a GET_MACRO function.

The macro assumes that each group plan is executed in a distributed fashion. This means that the agents that participate in each group plan are in charge of executing their own activities, as compared to some centralized dispatcher, which makes execution decisions. Therefore, with respect to the mission plan, the duration of all group activities are uncertain (i.e. controlled by the groups). This is true even in the case where all the activities in a group plan are controllable. The macros enable terse representation of the mission, an executive summary of sorts, in which the details of the group activities are hidden. This allows the executive to reason about scheduling the group activities without dealing with the specifics of how each activity in the group is accomplished. The macros achieve this simplicity at the cost of loss of information. The macros simply model the feasible duration of each group plan as an uncertain duration. This is the simplest way to model the group plans at the mission level. In Chapter 5, we discuss an alternative macro representation that preserves more information about the group plan.

The two layer, partially controllable, multi-agent plans with communication constraints are formalized as a two-layer Multi-Agent Temporal Plan Network with Uncertainty (MTPNU). Recall that a TPN is a set of activities to be performed, each of which includes a start and end time, together with a set of temporal constraints that specify the valid activity start and end times for each activity, specified as a simple temporal constraint. Hence a TPN is a generalization of a STN consisting of a set of activities A, and a mappings, T^+: A→ N, and T^-:A→N, mapping the start and end times to the timepoints in the STN. A TPN under uncertainty (TPNU) is analogous, where the temporal constraints are expressed as a STNU. A multi-agent TPNU (MTPNU) extends the TPNU in two fundamental ways. First it introduces a set of agents, Q, and a distribution, D: N→Q, mapping the timepoints, N, to an agent, Q. Thus, each timepoint is owned by a specific agent. Second, the MTPNU introduces a set of communication constraints, formalized as a communication availability graph (CAG), which specifies when the agents are able to communicate with one another.

A two-layer MTPNU extends the definition of the MTPNU. The two-layer MPTNU is the tuple <M,G,B>, where M is a mission layer MTPNU (mission plan) and G is a set

of group layer MTPNUs (group plans), and B is a function mapping the macros in the mission plan to the group plans. The mission plan, macro, group plans are formally described below.

The mission plan, M = <Γ, A, T, Q, D, Π, Ψ >, where:

- *Γ*: is the STNU = <N, E, l, u, C> that specifies the temporal constraints of the plan.
- *A*: is a set of abstract group activities. These group activities are an abstraction of the group plans. Each group activity is associated with a group plan.
- *T*: consists of two functions T^+: A→ N, and T^-:A→ N, which map the start time and end time respectively of each group activity to a timepoint.
- *Q*: is a set of groups which consists of a set of agents. There is one special group which contains no agents, called the mission group. This mission group is associated with all timepoints that are not explicitly part of a macro.
- *D*: is a group distribution function, D: N→ Q, mapping each timepoint in the mission plan to a group.
- Π: is a communication availability graph that specifies when the groups can communicate with one another.
- Ψ is a set of macros associated with each group in Q.

A macro Ψ = <γ,a,t> of the mission plan M, where:

- γ is a two timepoint STNU = <*N,E,l,u,C*> containing one executable start timepoint, *s*, and a contingent end timepoint, *e*, and a contingent link between *s* and *e* where γ ⊆ Γ[M].
- *a* is group activity where *a* ∈ *A*[*M*]
- *t* ⊆ T[M] is a function mapping the start time of *a* to *s* and the end time of *a* to *e*.

A group plan g ∈ G is a tuple < Γ, A, T, Q, D, Π >, where:

- Γ is a STNU = <N,E,l,u,C> that specifies the temporal constraints of the group plan.
- *A* is a set of activities executed by the agents in the group.
- *T* is a mapping T^+: A→ N, and T^-:A→ N, which map the start time and end time of each activity to a timepoint in Γ.
- *Q* is a set of agents.
- *D* is a distribution which is a mapping D: N→ Q that maps each timepoint in the group plan to an agent in Q.
- *Π* is a communication availability graph (CAG) specifying when each agent can communicate with one another.

In this thesis, we only consider group plans that have fully connected communication availability graphs. The communication availability graph in the group plan is fully connected. This means that the group members of each group plan are allowed to communicate throughout the group plan. In the mission plan, we only assume that the

communication availability graph is specified such that each group is able to communicate with the agent that executes the start of the plan. Thus, during execution, each group knows when the mission starts.

Consider the two-layer MTPNU shown in Figure 3-9. This two-layer plan encodes a mission where four rovers, R1, R2, R3, and R4, participate in three group activities: search, sample, and send-data, to achieve the mission. In the mission plan the search and sample activities occur concurrently, and are followed by the send-data activity. In the search group activity, R1 and R2 search for new science targets. In the sample group activity, R3 and R4 sample a rock at the current science site. In send data group activity, R1 and R2 send R3 what they discovered exploring, while R4 sends R3 the science data it collected. Then R4 relays this data back to Earth.

The mission plan, shown in Figure 3-9 contains three group activities: search, sample and transmit_receive. Each group activity is contained in group macro. Recall that a macro consists of an executable start timepoint, a contingent end timepoint, the group activity, and the contingent link connecting the start and end timepoint. Each timepoint in the mission plan is mapped to a group. The start and end timepoints of each macro are associated with their corresponding group plan. In particular, timepoints A and B are associated with the search group (light gray), timepoints C and D are associated with the sample group (dark gray), and timepoints F and G, are associated with the transmit_receive group (black). The mission plan also contains two special mission timepoints (white), namely, timepoint Z, which is the start of the mission, and timepoint E which serves as a connector timepoint. Furthermore, the mission plan contains a set of requirement links constraining the possible execution times of each timepoint in the mission plan.

The group plans are also shown in Figure 3-9. Each timepoint of a group plan is mapped to an agent. The white timepoints are associated with R1, light gray timepoints are associated with R2, dark gray timepoints are associated with R3, and black timepoints are associated with R4.

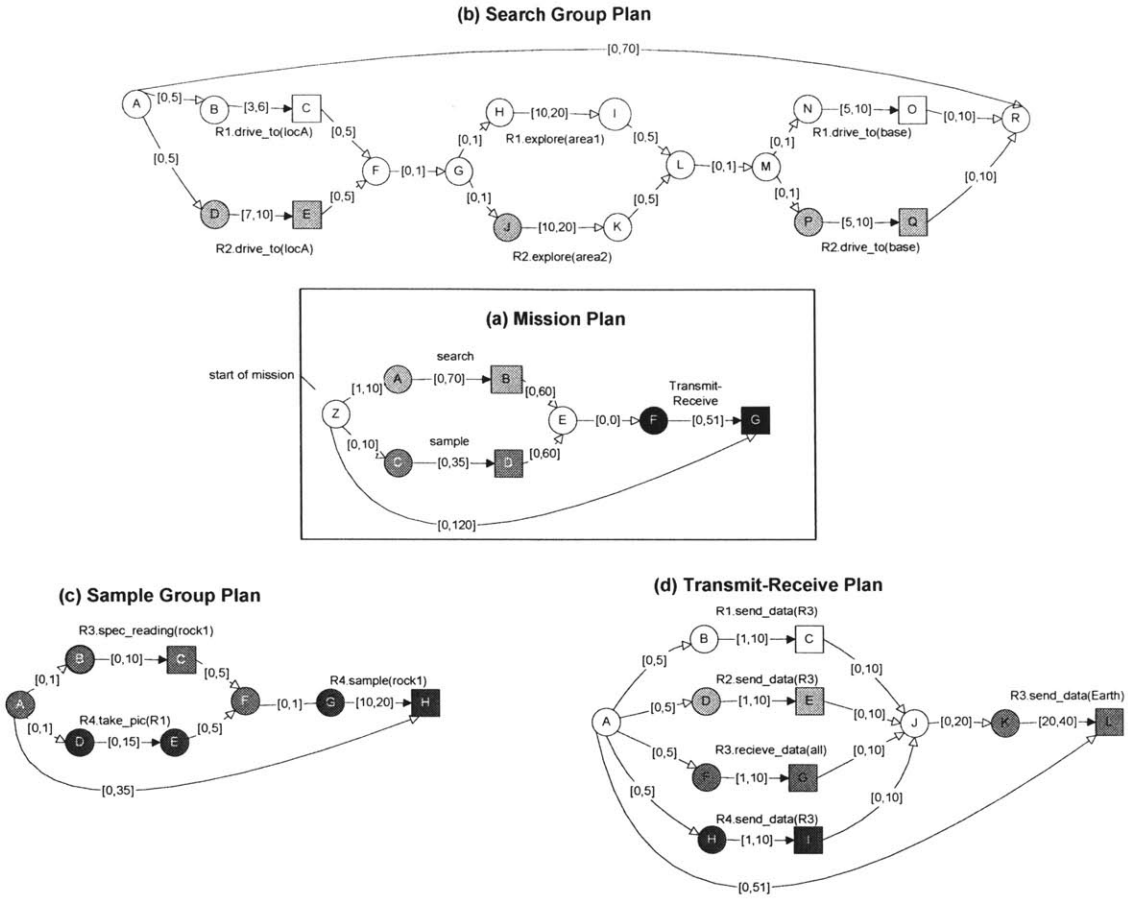


Figure 3-9 Two-Layer Multi-Agent Temporal Plan Network with Uncertainty

In the next three sections we discuss two methods used for constructing the two-layer plans. First, we describe how to use a variant of the Reactive Model-Based Programming Language (RMPL) in order to specify a two-layer plan. Second, we describe how a fully elaborated plan is converted into a two-layer plan, by clustering the plan into a set of tightly coordinating sub-plans

3.4.1 Group Programming Language (GPL)

The group programming language (GPL) is a programming language that enables a programmer to specify the two-layer MTPNUs. The GPL is a variant of the RMPL language [Williams 2003, Kim 2001]. Similar to RMPL, GPL explicitly allows for: concurrency, serialization, temporal constraints. However, GPL does not include many advanced features of the RMPL. The GPL language is a high level, object orientated programming language similar to C++. It uses a set of classes and a set of methods to define the plan. GPL provides a means to distinguish between requirement constraints and contingent constraints in the plan. It also introduces a special *wait* command that encodes a temporal constraint without imposing an activity. In order to specify two-layer plans, GPL, uses two special classes called the *mission* class and the *group* class. Using

these classes explicitly defines the distinction between the set of constraints that are a part of the mission plan and set of constraint that are a part of each group plan. GPL assumes that each agent can communicate within other agents in the same group; however, the groups may not be able to communicate with one another. Adding explicit communication constraints to GPL is left for future work.

The following describes the primitives of GPL and its mapping to a MTPNU.

Controllable Activity: *agent.activity (params) [lb, ub];*
A controllable activity contains an agent assignment, activity name, and a set of parameters for the activity. It also contains lower and upper time bounds on the duration of the activity. The controllable activity is converted into one executable start timepoint and one executable end timepoint, with a contingent link connecting the start and end timepoints. Both start and end timepoints are associated with an agent specified in the activity. The lower and upper bounds are used to create a requirement link between the start and end timepoints. The start and end of the activity is associated with the start and end timepoint. Figure 3-10 illustrates the mapping between the GPL and MPTNU for a controllable activity.

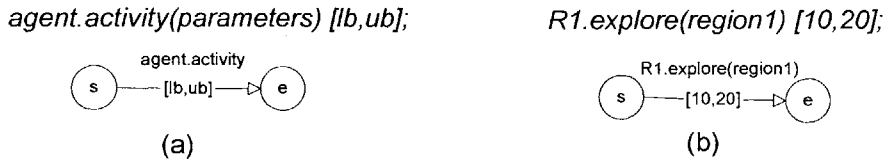


Figure 3-10 (a) This shows the GPL to MTPNU mapping for a controllable activity in general (b) This is specific example of a controllable activity mapping.

Wait: wait [lb, ub];
There is one special controllable activity called a wait, which encodes a purely temporal constraint. The wait does not contain an agent association. When the Wait is converted into a MTPNU it looses its activity status. It is used to place a temporal requirement in the plan without an activity. For example, consider the mapping between the GPL wait and the resulting MTPN is shown in Figure 3-11.

Figure 3-11 (a) general wait mapping (b) specific example of wait mapping

Uncontrollable Activity: *agent.activity (params) <lb, ub>;*
The GPL specification for defining an uncontrollable activity is similar to that of a controllable activity except that timebounds are used to create a contingent link between the start and end timepoints. Note that the brackets of the timebounds are different. The GPL to MTPNU mapping for an uncontrollable activity is shown in Figure 3-12.

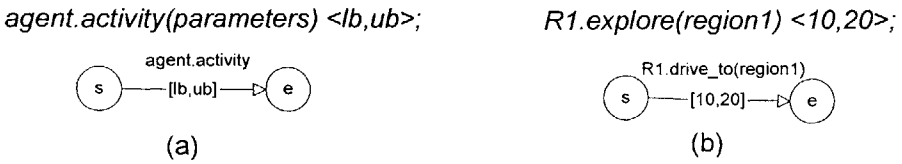


Figure 3-12 (a) **General mapping between a GPL uncontrollable activity and MTPNU.** (b) **A specific example.**

GPL contains two combinators: sequence and parallel, which encode serialization and concurrency, respectively. The combinators are applied to either primitives or a combination of primitives. The sequence and parallel combinators are defined as:

Sequential Structure: *sequence {statement1; statement2,...} [lb, ub];*
The sequential structure specifies that the two or more statements in the expression must be applied in sequence. The lower and upper bound place a requirement constraint between the start and end timepoint of the whole sequence. The sequence combinator inserts dummy timepoints into the plan, in order to connect the statements that are not associated with any specific agent. Figure 3-11 shows the mapping between sequence GPL combinator and an equivalent MTPNU.

Parallel Structure: parallel {statment1, stament2,...} [lb, ub];
The parallel combinator specifies that the two or more statements in the expression must be performed concurrently. Furthermore, the statements must start and end at the same time. The lower and upper bound specify a temporal requirement between the start and end of the parallel structure. Similar to the sequential combinator, the parallel combinator applies inserts dummy timepoints into the plan that are not associated with any specific agent. For example see Figure 3-9.

The rover MTPNU, shown in Figure 3-9, is specified in GPL in Figures 3-14 and 3-15. The GPL mission plan is shown in Figure 3-14 and the three group plans are shown in Figure 3-15.

```
Class mission
{
    rover_mission() {
    sequence {
        parallel {
            sequence { wait() [0,10]; search(); wait() [0,60] };
            sequence { wait() [0,10]; sample(); wait() [0,60] };
        }
        wait() [0,10];
        send_data();
    } [0,120]
}
```

Figure 3-13 Mission Plan in GPL

```
Class group
{
    Rovers R1, R2, R3, R4;   // define rover objects

    sample() {
        sequence {
            parallel {
                sequence { wait() [0,1]; R3.spec_reading(rock1) <0,10>; wait() [0,5] }
                sequence { wait() [0,1]; R4.take_image(rock1) <0,15>; wait() [0,5] }
            }
        wait() [0,1];
        R4.sample_rock(rock1) <10,20 >
        } [0,35]
    }

    send_data() {
        sequence {
            parallel {
                sequence { wait() [0,5]; R1.send_data(R3) <1,10>; wait() [0,10] }
                sequence { wait() [0,5]; R2.send_data(R3) <1,10>; wait() [0,10] }
                sequence { wait() [0,5]; R3.recieve_data() <1,10>; wait() [0,10] }
                sequence { wait() [0,5]; R4.send_data(R3) <1,10>; wait() [0,10] }
            }
            wait() [0,20];
            R4.send_data(Earth) <20,40 >;
        } [0,51]
    }

    search()
    sequence {
        parallel {
            sequence{ wait() [0,5]; R1.drive_to(locA) <3,6>; wait()[0,5] };
            sequence{ wait() [0,5]; R2.drive_to(locA) <7,10>; wait()[0,5] };
        }
        wait() [0,1];
        parallel {
            sequence{ wait() [0,1]; R1.explore(region1) <10,20>; wait()[0,5] };
            sequence{ wait() [0,1]; R2.explroe(region2) <10,20>; wait()[0,5] };
        }
        wait()[0,1];
        parallel {
            sequence{ wait() [0,1]; R1.drive_to(base) <5,10>; wait()[0,5] };
            sequence{ wait() [0,1]; R2.drive_to(base) <5,10>; wait()[0,5] };
        }
    }[0,70];
}
```

Figure 3-14 The Three Group Plans in GPL

3.4.2 Converting Multiagent Plans to Two-Layer MTPNUs

In this subsection we describe the process of converting a MTPNU into a two-layer MTPNU. This conversion is done by clustering the tightly coordinated group plans within the original plan into a set of clusters. Next, each cluster is extracted from the original plan to become a group plan. Then we compute the feasible timebounds for the group plan. These feasible timebounds are used to model the duration of each group plan in the mission plan. Finally, we replace the cluster with a macro in order to form the mission plan.

Recall that a macro is a simple abstraction of the group plan, where the macro consists of an executable start timepoint, a contingent end timepoint, and a contingent link, connecting the start and end timepoint with a lower an upper bound. This simple abstraction hides the details of each group plan within the mission plan.

Also, recall that the duration of the macro is modeled as an uncontrollable duration, because the mission plan has no control over how each group decides to execute their plan - each group plan is executed independently. Using this simple representation makes it easy to compute the macro and preserves the maximum flexibility in each group plan.

We assume that the planner (human or automated) can identify the portions of the plan that require tight coordination. These portions are called *clusters of tight coordination.* Automatically determining how to cluster an arbitrary plan is left for future work. Furthermore, we assume that within each cluster of tight coordination, the agents are able to communicate with one another. In other words, we assume that tight coordination is conjunct with communication. The following describes a communication cluster which corresponds to a cluster of tight coordination.

Definition 3-16 (Communication Cluster): *Given an STNU $G = <N,E,l,u, C>$ a communication cluster is a set of timepoints, $B \subseteq N$, where each agent, $a \in A$, that owns a timepoint in B, maintains communication with all other agents A, during the entire execution duration of all timepoints $n \in N$.*

Furthermore, we say that a plan is *completely clustered* if all timepoints associated with activities are a member of a cluster, and the start and end of each activity is in the same cluster. This ensures the each activity will be a member of a group plan, when the plan is converted into a two layer plan.

We assume that there exists one timepoint that precedes all others and one timepoint that finishes after all others. If these timepoint do not exist, then they can be added to the cluster. These are referred to as the start and end cluster points, respectively. Furthermore, we assume that all edges connecting timepoints within the cluster to timepoints outside the cluster are connected through the start and end timepoints. If this condition is not satisfied, then the complete clustering is invalid. Consider the complete clustering shown in Figure 3-15. Cluster 1 and 3 are valid; however, cluster 2 contains a timepoint G, which is neither a start or end timepoint of the cluster, yet it contains an edge GI, connecting it to another cluster.

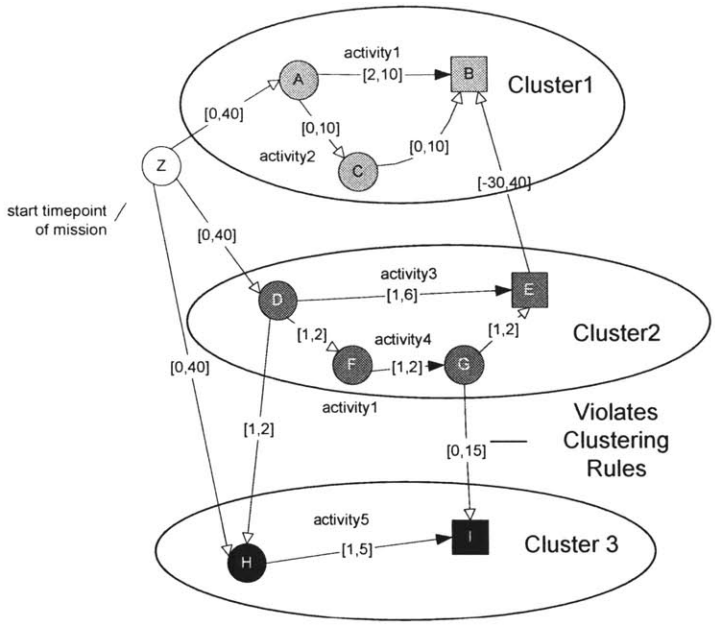


Figure 3-15 Clustering Example

After completely clustering the plan, the next step is to determine the range of feasible execution times for each cluster. These feasible execution times are specified as the lower and upper bound on the macro's contingent link. [Dechter et. al. 1990] showed that the feasible durations of each timepoint in a distance graph can be computed using two Single-Source Shortest-Path (SSSP) computations.

The macro's upper bound is set to the shortest distance from the clusters start timepoint to the clusters end timepoint. Similarly, the macros lower bound is set to the negation of the shortest path from the clusters end timepoint to the clusters start timepoint. These two shortest paths are computed via the Bellman-Ford SSSP algorithm [CLR 1990].

After computing the feasible duration for each cluster, the next steps involve separating the clusters from the multi-agent plan, forming group plans from these clusters, and substituting each cluster with its group plan macro activity with the macro. The pseudo-code for the entire conversion from multi-agent plan to MTPNU is given in Figure 3-16.

```
CONSTUCT_TWO_LAYER_PLAN (F, C)
Input: MTPNU F and valid set of clusters C which completely cluster F
Output: Two-Layer MPTN, T, with a mission plan M and a set of group plans G
1  for each cluster c ∈ C
2      s ← start timepoint of cluster c
3      e ← end timepoint of cluster c
4      g ← CREATE_GROUP_PLAN(F,c)
5      add g to G
6      if ¬BELLMAN_FORD_SSSP(g, s ) return NIL
7      ub ← d[e]
8      BELLMAN-FORD_SSSP ( g, e )
9      lb ← d[s]
10     macro ← CREATE_MACRO(s,e,lb,ub, g)
11     substitute macro for cluster c in the MTPNU F
12 end for
13 M[T] ← G
14 G[T] ← F
15 return T
```

Figure 3-16 Pseudo Code for CONSTRUCT_TWO_LAYER_PLAN

3.5 The Decoupling Algorithm

In this section we describe the mission plan decoupling algorithm, which decouples each group macro in the mission plan. The decoupling of the macros in the mission plan allows each group plan to be scheduled independently. The simplest method to perform this decoupling is to use a slight variation of the strong controllability algorithm introduced by [Vidal 2000]. Figure 3-17 shows the decoupling procedure. First, the strong controllability algorithm decouples the executable timepoint from the contingent timepoints, by making all requirement edges that connect contingent timepoints dominated (redundant). Then the decoupling algorithm selects a consistent assignment to the executable timepoints in the mission plan.

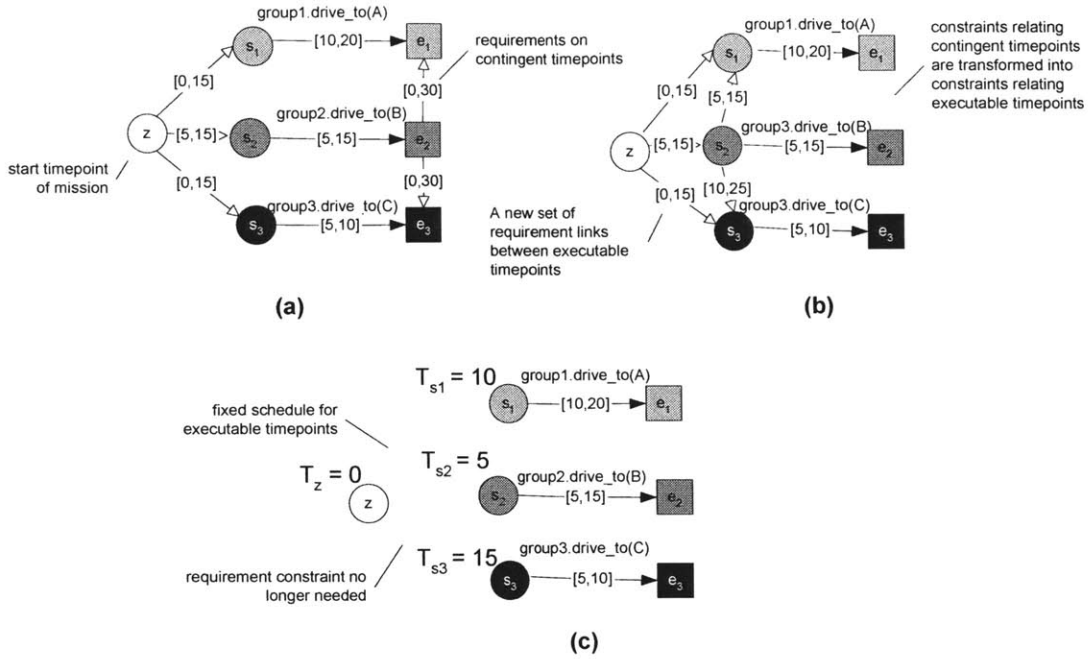


Figure 3-17 (a) The original mission plan containing requirement edges connecting contingent timepoints (b) The mission plan after the contingent timepoints are decoupled by the strong controllability algorithm. Note, all requirement edge connecting contingent timepoints are removed. (c) The decoupling algorithm fixes the start time for each executable timepoints. This eliminates the need to propagate scheduling times during execution.

In the future work section of Chapter 5, we explore an improved decoupling algorithm that combines the STN decoupling algorithm, introduced by [Berger 2003], with the strong controllability algorithm to enable the start time of each group plan to retain some flexibility, with respect to the start time of the mission.

First we explore strong controllability with some simple examples. Then we present the strong controllability checking algorithm introduced by [Vidal 2000]. Finally, we present the decoupling algorithm.

3.5.1 Strong Controllability

First let's clarify the definition of strong controllability. Recall that a plan is strongly controllable, if there exists a viable execution strategy that does not depend on knowing the outcomes of the uncontrollable durations. Therefore, the schedule of the executable timepoints of a strongly controllable plan can be generated offline. Recall, that the viability of an execution strategy simply means that the schedule for the executable timepoints generated by the execution strategy is consistent for all situations. However, often there are many schedules of the executable timepoints that are consistent in all situations. In these cases, rather than committing to a schedule prior to execution, it is possible to dynamically generate the schedule. This process is exactly the same as

dynamically scheduling an STN. Strong controllability does not mean that the plan must be executed statically, it just means it can.

In this subsection, we explore some simple examples in order to demonstrate strong controllability. These examples also help set the stage for the strong controllability and decoupling algorithms to follow.

Example 3-2

Consider a scenario in which a rover must drive to a location on limited battery power. Assume that we can reliably determine that the battery will last between 5 and 10 minutes. Consider the following two cases. In the first case, the rover must drive to location A, which will take the rover between 1 to 2 minutes, and in the second case, the rover must drive to location B, which will take between 1 to 8 minutes. The duration of these drive-to activities is uncertain. In both cases, let's assume that the rover may start driving at anytime; however, the rover must reach the location before the battery dies. The plan for driving to location A is shown in Figure 3-18(a) and the plan for driving to location B is shown in Figure 3-18(b).

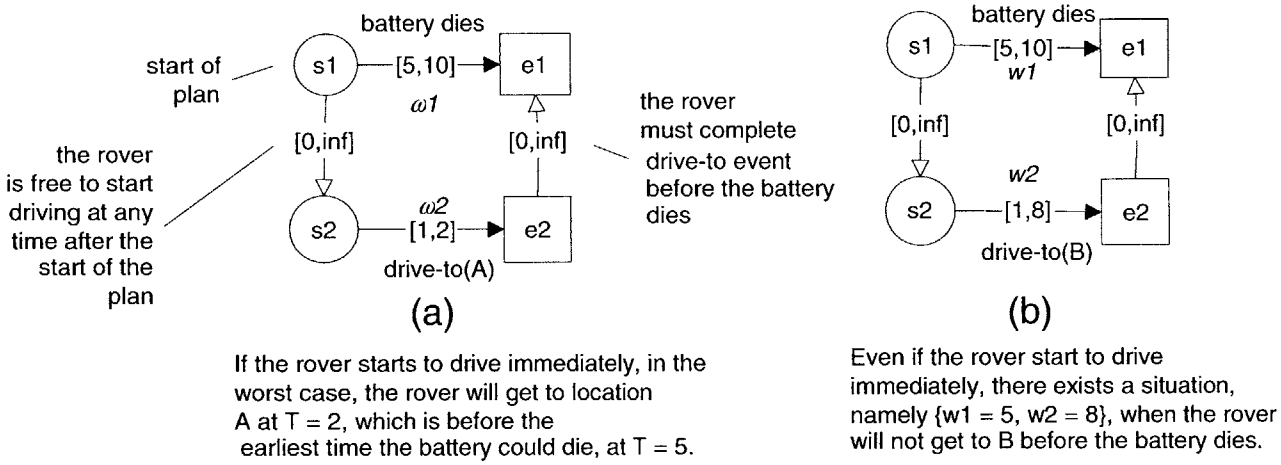


Figure 3-18 (a) A strongly controllable plan. (b) An example of a plan that is not strongly controllable.

Let's consider whether the plans shown in Figure 3-18 are strongly controllable. For simplicity, let's assume that the start time of each plan, timepoint s_1, is *a priori* fixed at zero, thus, $T(s_1) = 0$. In order for the plan to be strongly controllable, we need to be able to schedule the start of the drive-to activity, timepoint s_2, such that the assignment $T(s_2)$ is consistent in all situations. Recall, that a situation is defined as one valid assignment of

the uncertain durations. In this example, a situation, ω, consists of an assignment to both the duration of the lifetime of the battery, ω_1, and the duration of the drive-to activity ω_2. The only non-trivial temporal constraint plan is a lower bound constraint between timepoints e2 and e1, which specifies that the drive-to activity must occur before the battery dies.

The plan for driving to location A, shown in Figure 3-18(a), is strongly controllable. If the rover starts driving immediately (scheduling $T(s_2) = 0$) the latest time that the rover could arrive at location A is at T = 2, corresponding to a situation where $\omega_2 = 2$. This corresponds to an arrival time, $T(e_2) = 2$. This arrival time is before the earliest possible time the battery could die at $T(e_2) = 5$, which occurs in a situation where, $\omega_1 = 5$.

The plan for driving to location B, shown in Figure 3-18(b), is not strongly controllable. Even if we start the drive-to(B) activity immediately, hence, scheduling T(s1) = 0. Then there exists a situation, namely $\{\omega_1 = 5, \omega_2 = 8\}$, when the rover arrives at location B, after the battery dies. In this case, $T(e_1) = 5$ and $T(e_2) = 8$ and the constraint between e2 and e1 is violated.

Example 3-3:

Now consider a scenario in which a rover must performing Entry, Descent, and Landing (EDL) on to the Martian surface, and then communicate its status back to Earth. Suppose we know that the EDL activity will take between 10 and 20 minutes and communicating to Earth will take between 10 and 20 minutes. Again, let's consider two cases. In the first case, the scientists are in charge of scheduling the rover activities, and require that the rovers must start communicating with Earth between 0 to 5 minutes after EDL. In the second case, the engineers are in charge and require that the rover starts reporting back some time between 10 and 30 minutes after landing. The plan for each case is shown in Figure 3-19. In order for the plan to be strongly controllable, we need to be able to schedule the start of the communicate activity, s2, offline. Again, we assume that the start of the mission is at T = 0.

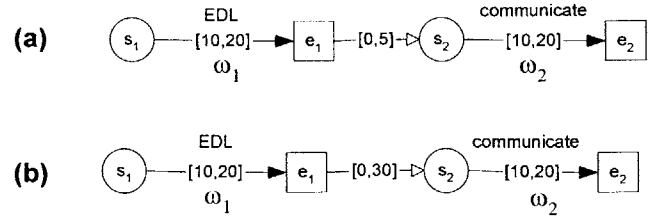


Figure 3-19 (a) **This plan in not strongly controllable.** (b) **This plan is not strongly controllable.**

Let's first consider the plan in Figure 3-19(a). In order to determine if the plan is strongly controllable, let's consider the valid execution time for timepoint s_2 in various situations. Specifically, let's consider how the valid execution time of s_2 is affected by the duration of the EDL activity. In the earliest possible situation, $\omega_1 = 10$, and s_2 must occur between [10, 15] minutes. In the latest possible situation, $\omega_1 = 20$, and s_2 must occur between [20, 25] minutes. There is no overlap in the execution windows for s2 in these two situations; therefore, it is impossible to schedule s_2 such that it satisfies the constraints both situations. Hence, the plan is not strongly controllable.

Now let's consider the plan in Figure 3-19(b). In the earliest situation $\omega_1 = 10$ and s2 must occur between [30, 50] minutes. In the latest situation $\omega_1 = 20$, and s2 must occur between [30, 50] minutes. In this case there is an overlap between the valid execution windows of s2. This intersection is precisely the execution times for s_2 that are consistent for all possible situations. Specifically, if s_2 is scheduled anytime between [30, 40] minutes, it will be consistent for all situations. Therefore, the plan is strongly controllable.

In the EDL scenario, the flexibility between the end of the EDL activity and the start of the communicate activity as compared to the uncertainty in the EDL activity, determined whether or not the plan was strongly controllable. Also, note that we only needed to consider the extreme values of the EDL uncertainty in order to determine the valued execution times for s2.

In the previous examples we used a type of worst situation analysis to determine if the plans were strongly controllable. [Vidal 2000] generalized this type of worst situation in a strong controllability checking algorithm. This algorithm is presented in the next subsection, which is followed by the decoupling algorithm.

3.5.2 Strong Controllability Checking Algorithm

The strong controllability algorithm introduced by [Vidal 2000] uses a type of worst situation analysis to reduce the problem of checking strong controllability into a problem of checking temporal consistency. The algorithm is composed of two steps: a transformation step, followed by a consistent checking step. In this section we operate on the associated distance graph (DGU) of an STNU. In the transformation step, the algorithm decouples the executable timepoints from the contingent timepoints, meaning that the allowed execution times for the executable timepoints are no longer a function of the times of the contingent timepoints. The algorithm achieves this type of decoupling by transforming the requirement edges of the DGU that relate contingent timepoints into requirement edges that only relate executable timepoints. For example, the transformation uses a type of worst situation analysis to ensure that the transformed constraint entails the original temporal constraints in all possible situations. The transformed edges are placed in a new distance graph called the *transformed graph.* For example, the transformed graph does not contain any contingent timepoints; therefore, the schedule of the

transformed graph is not dependent on the uncontrollable durations. In the temporal consistency checking step, the algorithm checks if the transformed graph is temporally consistent. If the transformed graph is temporally consistent then the original graph is strongly controllable. The temporal consistency checking step is done by checking for negative cycles in the transformed graph. This check is done using the Bellman-Ford SSSP [CLR], although any negative cycle checking algorithm works.

In the transformation step, we need to consider four possible types of requirement edges. Specifically, a requirement edge may start on either an executable or a contingent timepoint, and similarly, it may end on either an executable or a contingent timepoint.

The distance graph in Figure 3-20 shows the four types of requirement edges we need to consider. It contains an (executable/executable) edge CA, a (contingent/executable) edge BC, an (executable/contingent) requirement edge CB, and a (contingent/contingent) requirement edge DB. Each edge has a distance of a, b, c and d, respectively.

The distance graph in Figure 3-20 contains two uncontrollable durations: $\omega_1 \in [l_1,u_1]$ and $\omega_2 \in [l_2,u_2]$. These uncontrollable durations are associated with the contingent timepoints B and D, respectively. The first uncontrollable duration, ω_1, starts at the executable timepoint A, and finishes on the contingent timepoint B. The second uncontrollable duration, ω_2, starts on timepoint C, and finishes on timepoint D. Recall that upper and lower bounds of the uncontrollable durations is precisely the same information contained in the contingent edges. In the derivation of the transformation rules we will use the uncontrollable durations for convenience.

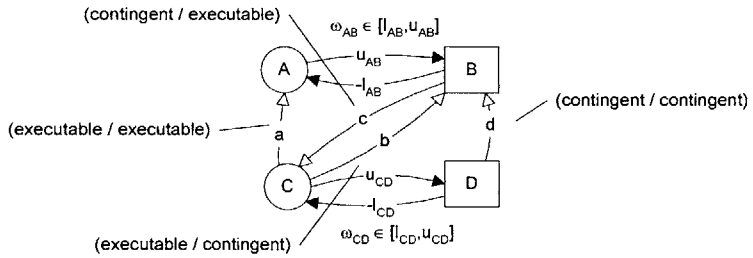


Figure 3-20 The requirement edges fall one of four types depending on the type of start timepoint and type of end timepoint. They timepoints are either executable or contingent.

The strong controllability transformation rule for each type of requirement edge is now presented. The transformation rules describe how to convert the requirement edges

of a DGU into a new requirement edge that only relates executable timepoints. For each transformation rule we present both an algebraic and a graphical derivation of the rule, except for rule 1 which is trivial. The transformation rules reference the DGU shown in Figure 3-20.

Case 1 (executable/executable)

Requirement edges that start and finish on an executable timepoint are precisely in the format that we seek; therefore, these constraints remain unchanged in the transformation. For example, the edge CA in Figure 3-20 remains unchanged in the transformation. Note however, that this edge may be tightened as a result of some other transformation.

(Executable/Executable) Transformation Rule: *Any requirement edge CA, constraining two executable timepoints is unchanged in the strong controllability transformation step.*

Case 2 executable/contingent

Consider requirement edge CB with d(CB) = b, as shown in Figure 3-20. This edge constrains the execution time of the contingent timepoint B w.r.t the executable timepoint C. Our goal is to derive a new constraint CA such that, no matter how long the uncontrollable duration, $\omega_{AB} \in [l_{AB},u_{AB}]$, takes, the original requirement constraint, CB, is satisfied. In other words, we seek a new constraint CA, which makes the edge CB dominated in all situations. The algebraic derivation of this new constraint CA is given below.

(1) $T_B - T_C \leq b$: original constraint BC
(2) $T_B = T_A + \omega_{AB}$: execution of B in terms of the uncontrollable duration
(3) $T_A - T_C \leq b - \omega_{AB}$: sub (2) into (1) and rearrange
(4) $T_A - T_C \leq b - u_{AB}$: tightest constraint for all possible situations

The original constraint imposed by CB is expressed in (1). Equation (2) represents the execution time of B in terms of the execution time of A and the uncontrollable duration $\omega_{AB} \in [l_{AB},u_{AB}]$. Substituting (2) into (1) and rearranging results in equation (3), which relates execution times of C and A, and explicitly contains the uncontrollable duration ω_{AB}. Notice that Equation (3) no longer contains the execution time of contingent timepoint B. Equation (3) corresponds to an edge CA in the DGU. In order for this constraint to be satisfied in all situations it is sufficient to consider the worst situations. Equation (3) imposes the most restrictive constraint when $\omega_{AB} = u_{AB}$. We call this the worst situation because it imposes the tightest constraint on the timepoints of the distance graph. Applying the constraint in equation (4) ensures that the original constraint BC is satisfied (i.e. dominated) in all situations. This constraint corresponds to an edge CA with distance $b - u_{AB}$. After applying this new constraint CA to the distance graph, the original constraint CB can be removed from the graph. The strong controllability transformation rule for a contingent/executable requirement edge is given below.

(Executable/Contingent) Transformation Rule *Given an uncertain duration $\omega_{AB} \in [l_{AB},u_{AB}]$, any requirement edge CB with $d(CB) = b$ between an executable timepoint C and contingent timepoint B is transformed into a new requirement edge CA with $d(CA) = b - u_{AB}$ in the strong controllability transformation step.*

The graphical derivation of this transformation rule is illustrated in Figure 3-21. The derivation relies on a shortest path argument in the projection of the DGU. The DGU shown in Figure 3-21(a) contains the executable/contingent edge CB. The arbitrary projection of this DGU is shown in Figure 3-21(b). The new constraint CA is derived by computing the shortest path CBA in this arbitrary projection. The worst possible situation (which imposes the tightest constraint) occurs when $\omega_{AB} = u_{AB}$. This tightest constraint is shown in Figure 3-21(c). After applying this new constraint CA, the path CAB is always shorter than the direct path CB; therefore, the edge CB is dominated in all situations and can be removed.

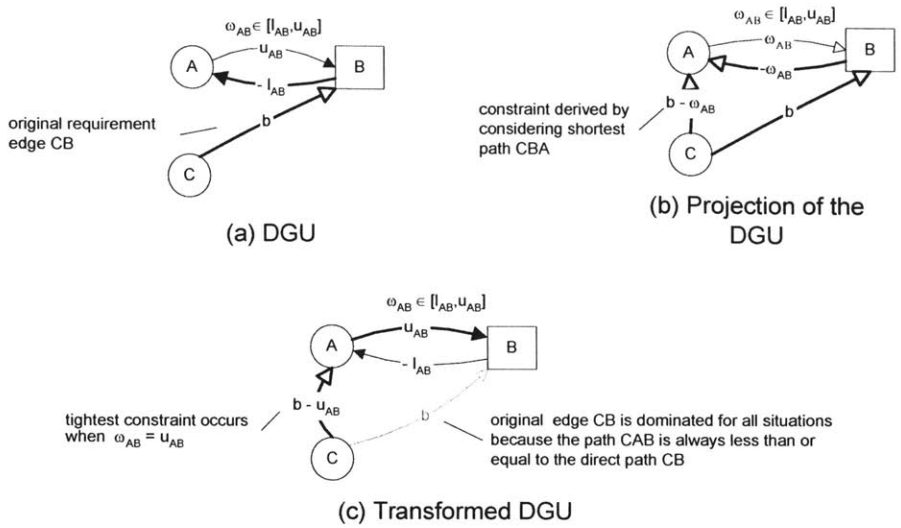


Figure 3-21 (a) The DGU containing an executable/contingent requirement edge CB. **(b)** A new constraint CA is derived by computing shortest path CBA through the arbitrary projection of the DGU. **(c)** This edge CA is tightest when $w_{AB} = u_{AB}$ and dominates the edge CB in all situations.

Case 3: contingent/executable

Consider the contingent/executable requirement edge BC with d(BC) = c, as shown in Figure 3-20. For this edge we seek a new constraint AC, which dominates BC for all situations. The algebraic derivation of this new constraint AC is given below.

(5) $T_C - T_B \leq c$: original constraint BC
(6) $T_B = T_A + \omega_{AB}$: execution time of B in terms of uncontrollable duration
(7) $T_C - T_A \leq \omega_{AB} + c$: sub (6) into (3) and rearrange
(8) $T_C - T_A \leq l_{AB} + c$: tightest in all possible situations

The original requirement BC is given in (5). The execution time of B in terms of the uncontrollable duration $\omega_{AB} \in [l_{AB}, u_{AB}]$ is given in (6). Combining (5) and (6), results in a new equation that relates execution times of A and C in terms of the uncontrollable duration ω_{AB}. The worst situation (which imposes the tightest constraint) occurs when $\omega_{AB} = l_{AB}$. Equation (8), gives this tightest constraint, which corresponds to an edge CA with $d(CA) = l_{AB} + c$. After applying this new constraint CA, the original constraint BC is dominated for all possible situations, and therefore, removed from the distance graph.

(Contingent/Executable) Transformation Rule: Given an uncontrollable duration $\omega_{AB} \in [l_{AB}, u_{AB}]$, *any requirement edge BC with d(BC) = c, starting on an contingent timepoint B and ending on an executable timepoint C, is transformed into a new requirement edge AC with* $d(AC) = l_{AB} + c$ *in the strong controllability transformation step.*

The graphical derivation of this transformation rule is illustrated in Figure 3-22. The DGU containing the contingent/executable edge BC is shown in Figure 3-22(a). The arbitrary projection of the DGU is shown in Figure 3-22(b). A new edge AC can be derived by computing the shortest path ABC in this arbitrary projection. This constraint is the same as the constraint in equation (7). The worst situation (which results in the tightest edge AC) occurs when $\omega_{AB} = l_{AB}$. The tightest constraint AC is shown in Figure 3-22(c). After applying this tightest constraint AC, the path BAC is always shorter than the direct path of BC; therefore, BC is dominated and can be removed from the graph.

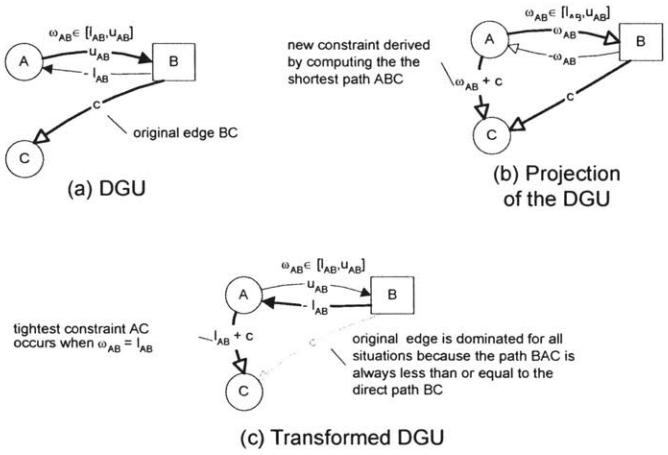


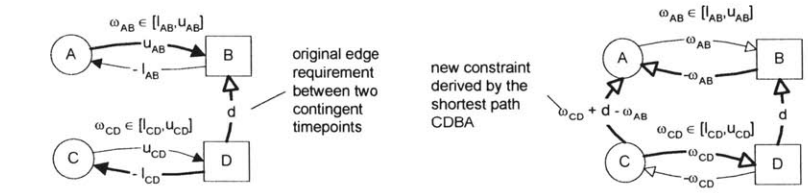
Figure 3-22: (a) The DGU containing the contingent/executable edge BC. (b) A new edge AC is derived by computing the shortest path BAC in the projection of the DGU. (c) The tightest constraint AC occurs in the situation when $\omega_{AB} = l_{AB}$ and dominates the edge BC in all situations.

Case 4 (contingent/contingent)

Consider the contingent/contingent requirement edge DB with d(DB) = d, as shown in Figure 3-20. In this case we seek a requirement edge DA to dominate DB for all situations. This edge is derived by using sequential applications of the executable/contingent transformation rule and contingent/executable transformation rule. If we ignore the fact that D is a contingent timepoint, we can apply the executable/contingent transformation rule to derive a new edge DA. This new edge DA has distance of $d + u_{AB}$. The edge DB then is transformed by the contingent/executable transformation rule into a new edge CA with a distance of $d + l_{CD} - u_{AB}$. The contingent/contingent transformation rule is given below.

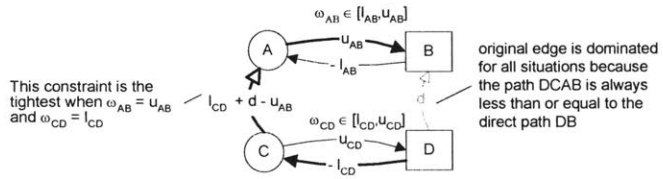
(Contingent/Contingent Transformation Rule 4) Given the uncontrollable durations $\omega_{AB} \in [l_{AB}, u_{AB}]$, *and* $\omega_{CD} \in [l_{CD}, u_{CD}]$, *any requirement edge DB with* $d(DB) = d$, *starting on a contingent timepoint D and ending on a contingent timepoint B, is transformed into a new edge CA with* $d(CA) = l_{CD} + d + u_{AB}$ *in the strong controllability transformation step.*

The graphical derivation of the contingent/contingent transformation rule is similar to the previous two graphical derivations and is illustrated in Figure 3-22. The DGU containing the contingent/contingent edge DB is shown in Figure 3-22(a). A new constraint CA is derived by considering the shortest path CDBA in the arbitrary projection of the DGU, as shown in Figure 3-22(b). The worst case situation occurs when $\omega_{AB} = u_{AB}$, *and* $\omega_{CD} = l_{AB}$. The resulting constraint is shown in Figure 3-22(c). After applying this constraint, the path DCAB is always less than or equal to the path DB; therefore, the edge DB is dominated in all situations and can be removed from the distance graph.



(a) DGU

(b) Projection of DGU



(c) Transformed DGU

Definition 3-17 (a) The DGU contains a contingent/contingent edge DB. (b) A new edge CA is computed by considering the shortest path CDBA in the projection of the DGU. (c) This tightest constraint CA occurs the situation when $\omega_{AB} = u_{AB}$, *and* $\omega_{CD} = l_{AB}$ and dominates DB for all situations.

The strong controllability checking algorithm [Vidal 2000] simply applies transformation rules defined above to create a transformed distance graph, then checks if this transformed graph is temporally consistent.

The pseudo code for the strong controllability checking algorithm is given in Figure 1-19. In Line 1, the executable timepoints of the input distance graph, G, are copied to a new transformed distance graph, T. After this, every executable timepoint in G has a corresponding executable timepoint in T. In Line 2, all of the edges of T are initialized to NIL. Lines 3-6 compute the transformed distance graph T by looping through each

requirement edge in the input distance graph G. Line 4 applies the strong controllability transformation rule to each requirement edge (u,v) in G and generates a transformed edge (u',v') with d(u',v') = x. In Line 5, the corresponding edge (u',v') in T is updated by calling the UPDATE_EDGE function. The pseudo-code for the UPDATE_EDGE function is shown in Figure 3-24. If the edge (u',v') does not exists in T, then the function adds a new edge (u',v') to T with distance x; otherwise, it updates the distance of the edge (u',v') to x, if the new value is smaller than the existing distance.

After the algorithm completes the transformation process, the transformed distance graph, T, contains a set of executable timepoints corresponding to the set of executable timepoints in G, and a set of requirement edges constraining these executable timepoints. The question of strong controllability is resolved in Lines 7-8. The algorithm calls the Bellman-Ford Single-Source Shortest-Path (SSSP) algorithm [CLR] to check if the transformed distance graph, T, contains any negative cycles. Note that any negative-cycle detecting graph-algorithm could be used in place of the Bellman-Ford SSSP algorithm. If the transformed distance graph, T, does not contain any negative cycles, then T is temporally consistent, and the strong controllability algorithm returns true. If the transformed graph is temporally inconsistent, then the algorithm returns false.

```
IS_STRONGLY_CONTROLLABLE (G)
Input: a distance graph with uncertainty G
Returns: TRUE if G is strongly controllable, otherwise FALSE
1  copy all executable timepoints of G to T
2  initialized all edges of T to NIL
3  for each requirement edge (u,v) ∈ E[G]
4      transform the edge (u,v) using SC transformation rules to an edge (u',v') with d(u',v') = x
5      UPDATE_EDGE( T, u', v', x )
6  end for
7  s ← start timepoint in T
8  consistent ← BELLMAN_FORD_SSSP( T, s )
9  return consistent
```

Figure3-23 Pseudo-Code for IS_STRONGLY_CONTROLLABLE

```
function UPDATE_EDGE(T, u ,v, x )
Input: a distance graph T, start timepoint u, end timepoint v, and distance x
Effects: updates the distance d(u,v) if it exists, otherwise adds a new edge (u,v) with d(u,v) = x
1  if T.Get_Edge(u,v) = NIL
2      T.Add_Edge(u,v)
3      T.d(u,v) ← x
4  else
5      T.d(u,v) ← MIN( T.d(u,v), x )
6  end if
```

Figure 3-24 Pseudo Code for UPDATE_EDGE

Example 3-4:

Consider the distance graph shown Figure 3-25(a). The strong controllability algorithm first copies the timepoints A, B and D to the transformed graph, T, and initializes the edges of T. The executable/executable edges AB, BA, AD and DA are simply copied over to T. The contingent/executable edge CA with d(CA) = 5 is converted into a new edge BA with d(BA) = -5. The contingent/contingent edge EC with d(EC) = 12 is converted into an edge DB with d(DB) = 5. The executable/contingent edge AE with d(AE) = 15 is converted into an edge AD with d(AD) = 2. The resulting transformed distance graph is shown in Figure 3-25(b). The algorithm then runs the Bellman-Ford SSSP from the source timepoint A on the transformed distance graph T. The transformed distance graph T contains no negative cycles; therefore, the algorithm returns true, indicating that the original distance graph is strongly controllable.

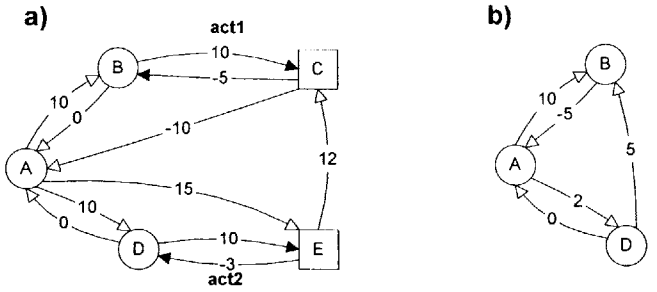


Figure 3-25 (a) The original DGU, G. (b) The transformed distance graph T used in the strong controllability algorithm.

The strong controllability checking algorithm runs in polynomial time. Copying the executable timepoints runs in O(N) time. Each transformation runs in constant time; therefore, the transformation of the requirement constraints runs in O(E) time. The running time of the Bellman-Ford algorithm runs in O(NE) time [CLR]. The running time of the strong controllability checking algorithm is dominated by this computation; therefore, the strong controllability algorithm runs in O(NE) time.

3.5.3 The Decoupling Algorithm

In this section we present a decoupling algorithm that is used to decouple the group plan. This decoupling algorithm operates on the mission plan in order to generate a fixed schedule for the start timepoint of each macro. These fixed start times are then passed to their respective group plans. The resulting group plans can be scheduled independently. The decoupling builds upon the strong controllability checking algorithm presented in the last subsection. The decoupling algorithm transforms the distance graph of the mission plan using the strong controllability transformation rules. If this transformed graph is consistent, the decoupling algorithm generates a schedule for the timepoints of the transformed graph. Note that any consistent schedule would work; however, we elect to schedule the group activities as early as possible. This schedule is used to fix the time of the corresponding group plans.

The pseudo-code for the decoupling algorithm is shown in **Figure 3-26**. The algorithm takes in a two-layer plan, consisting of a mission plan, M and a set of group plans, G. In Line 1 the decoupling algorithm gets the distance graph of the mission plan, G_m. Lines 2-12 are similar to the strong controllability checking algorithm. These lines compute the transformed graph T from the mission plan's distance graph and returns false if the transformed graph is inconsistent. The only difference between the decoupling algorithm and the strong controllability algorithm up to this point is that the decoupling algorithm runs a Single-Destination Shortest-Path (SDSP) algorithm instead of the Single-Source Shortest-Path (SSSP) algorithm on the transformed distance graph, in order to detect negative cycles. Both the SSSP and SDSP algorithms detect negative

cycles; however, the SDSP algorithm computes the earliest feasible time for each timepoint instead of the latest feasible time.

In Lines 13-22 the decoupling algorithm sets the schedule of the group plans. It loops through each executable timepoint n in the transformed graph T. In Line 15, it gets the macro associated with the timepoint by calling GET_MACRO function. If the timepoint is not part of a macro, then the GET_MACRO function returns NIL. If the timepoint is part of a macro, then the algorithm finds the corresponding group plan g ∈ G by calling the GET_GROUP_PLAN function in Line 17. In Line 18 the decoupling algorithm fixes the start time of the group plan associated with timepoint n. The start time of each group is set to the negation of the SDSP distance of n as computed in Line 7. Finally, in Line 22, the decoupling algorithm returns true.

```
Decouple ( M, G )
Input: A mission plan M and a set of group plans G
Effects: Decouples the group plans by generating a fixed schedule for the start of each group plan
Returns: TRUE if decoupling algorithm succeeds, otherwise FALSE
1  G_m ← get distance graph of mission plan.
2  copy all executable timepoints of G_m to T
3  initialized all edges of T to NIL
5   for each requirement edge (u,v) ∈ E[G_m]
6      transform the edge (u,v) using SC transformation rules to an edge (u',v')  with d(u',v') = x
7      UPDATE_EDGE( T, u', v', x )
8   end for
9  s ← start timepoint of T
10 consistent ← BELLMAN-FORD-SDSP( T, s )
11 if ¬ consistent
12     return FALSE
13 else
14     for each timepoint n ∈ N[T]
15         macro ← GET_MACRO(n)
16         if macro ¬ NIL
17             g ← GET_GROUP_PLAN(macro)
18             fix the start time of g to –d[n] computed by BELLMAN-FORD-SDSP algorithm
19         end if
20     end for
21 end if
22 return TRUE
```

Figure 3-26 Pseudo Code for Decoupling Algorithm

Example 3-5

This example extends Example 3-4. Consider the mission plan and set of group plans shown in Figure 3-27(a-b). This mission plan's distance graph is exactly the same as the distance graph used in Example 3-4. In Lines 1-8 the decoupling algorithm computes the

transformed distance graph, T, of the mission plan. The transformed graph T shown in Figure 3-27c. In Line 10 the decoupling algorithm computes the Single-Destination Shortest-Path (SDSP) distances (as shown in Figure 3-27c.) from timepoint A to all timepoints in the plan These distances are used in Line 18 to fix the start time of the group plans, as shown in Figure 3-27d. The start time of group plan1 is fixed at T = 5 and the start time of the group plan2 is fixed T = 0. Finally, in Line 22 the decoupling algorithm returns true, indicating that the decoupling algorithm succeeded.

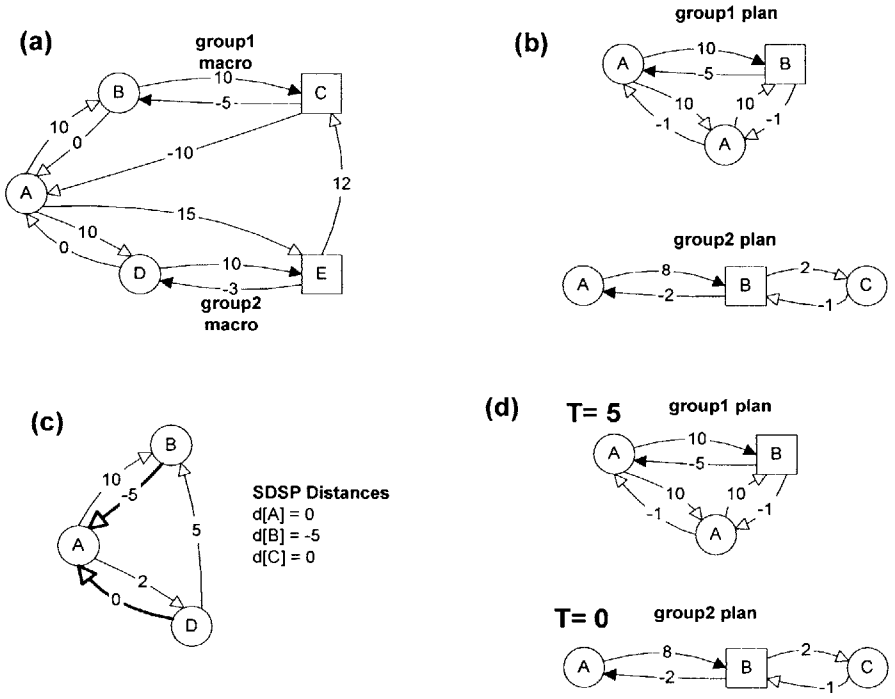


Figure 3-27 (a) The input mission plan (b) The input group plans (c) The transformed graph with SDSP distances (d) The group plans with fixed start times.

The decoupling algorithm runs in polynomial time. Lines 1-13 run in the same time as the strong controllability algorithm (i.e. O(NE)). In Lines 14-20 the decoupling algorithm loops through each timepoint and fixes the start time of each group plan. Using a simple lookup the GET_MACRO and GET_GROUP_PLAN run in linear time in the number of group plans. Therefore, Lines 14-20 run in O(NG), where G is the number of group plans. The number of group plans G is less than the number of edges in the distance graph; therefore, the decoupling algorithm is dominated by the Bellman-Ford SDSP computation. The running time of the decoupling algorithm is O(NE).

3.6 The Hierarchical Reformulation Algorithm

In this section, we present our novel Hierarchical Reformulation (HR) algorithm. The HR algorithm is a centralized reformulation algorithm that transforms a two-layer plan into a set of decoupled, minimally dispatchable group plans. The HR algorithm combines the decoupling algorithm, presented in the Section 1.5, with our novel fast dynamic controllability algorithm, presented in Chapter 4. The HR algorithm is used to enable the each group plan to be dynamically executed independently.

3.6.1 HR Algorithm Pseudo-Code

The HR algorithm operates on both layers of the two-layer plan. The fast dynamic controllability algorithm operates on the group plans, whereas, the decoupling algorithm primarily operates on the mission plan. Recall that the information contained in the group plans and mission plan are related. Specifically, the timebounds of the macros (contained in the mission plan) represent the feasible duration of the group plans. The HR algorithm must keep the information of the macros and group plans in sync. When the timebounds on the macros change, the HR algorithm updates the corresponding group plans via the UPDATE_GROUP_PLANS function. Similarly, when the feasible durations of the group plans are modified, the HR algorithm updates the timebounds of the corresponding macros using the UPDATE_MACROS function.

The pseudo-code for the HR algorithm is shown in Figure 2-23. The algorithm takes in a two-layer MTPNU, P = <M,G>, consisting of a mission plan, M, and a set of group plans, G, and generates a set of decoupled dispatchable MTPNUs with fixed start times. The algorithm returns true if the reformulation succeeds; otherwise, it returns false.

The HR algorithm may fail for several reasons. The HR algorithm fails if either the mission plan or group plans are temporally inconsistent. Furthermore, the HR algorithm fails if the group plans are not dynamically controllable or if the mission plan is not strongly controllable. The HR interleaves these failure checks throughout the algorithm.

Recall that a two-layer plan can be created in one of two ways: 1) it can be specified using a Group Plan Language (GPL) file, or 2) it can be created by calling CONSTRUCT_TWO_LAYER_PLAN function, given a fully elaborated plan, and associated clustering.

The HR algorithm is described using the two-layer plan, illustrated in Figure 3-28. This is referred to as the simple two-layer plan hence forth. The mission plan (shown in Figure 3-28a) contains two group activities: group act1 and group act2. The corresponding group plans, called group plan1 and group plan2, are shown in Figure 3-28b,c. Both of the group plans consist of three timepoints, with one contingent activity.

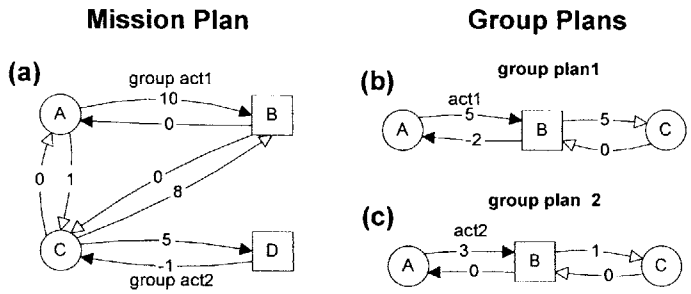


Figure 3-28 (a) The simple two-layer mission plan, (b) group plan1 (c) group plan2.

Lines 1-3 of the HR algorithm, Figure 3-28, calls the UPDATE_MACROS function and returns false if the update detects a temporal inconsistency in the group plans. The UPDATE_MACROS function first computes the feasible duration of each group plan. Then it updates the timebounds of the corresponding macros in the mission plan. This function is called at the beginning of the HR algorithm, in order to sync the macros within the mission plan with the constraints of the group plans. The pseudo code for the UPDATE_MACROS is shown in Figure 3-30.

```
function Hierachical_Reformulation (P)
Input: A two-layer MTPNU P = <M,G> consisting of a mission plan M and set of
group plans G
Effects: A set of decoupled dispatchable group plans G
Returns: TRUE if the reformulation succeeds, otherwise FALSE
1  consistent ← UPDATE_MACROS( G, M )
2  if (¬ consistent )
3      return FALSE
4  consistent ← COMPUTE_APSP_GRAPH( M )
5  if (¬ consistent )
6      return FALSE
7  UPDATE_GROUP_PLANS( G, M )
8  for each g ∈ G
9      controllable ← FAST_DC( g)
10     if(¬ controllable )
11         return FALSE
12     end if
13 end for
14 UPDATE_MACROS( G, M )
15 success ← DECOUPLE( M, G )
16 return success
```

Figure 3-28 Pseudo Code for HR algorithm

```
function UPDATE_MACROS( M, G )
Input: A mission plan M and set of group plans G
Effects: Computes the feasible duration of each group plan and updates the
corresponding timebounds of the macros in the mission plan.
Returns: True if the Group
1  for each group plan g ∈ G
2      s ← get start timepoint of g
3      consistent ← BELLMAN_FORD_SSSP( g, s )
4      if ¬consistent
5          return FALSE
6      ub ← max( d[n]  for each n ∈ N[g] )
7      BELLMAN_FORD_SDSP( g ,s)
8      lb ← -min( d[n]  for each n ∈ N[g] )
9      macro ← GET_MACRO(g)
10     UPDATE_EDGE( M, GET_START(macro), GET_END(macro), ub )
11     UPDATE_EDGE( M, GET_END(macro), GET_START(macro), -lb )
12 end for
13 return TRUE
```

Figure 3-30 Pseudo-code for UPDATE_MACRO

The UPDATE_MACROS function loops through each group plan (Lines 1-12). For each loop, the algorithm performs two shortest path computations on the group plan's distance graph. These shortest path computations compute the lower and upper bound on the feasible duration of each group plan. After computing the range of feasible durations, the algorithm updates the timebounds of the corresponding macro contained in the mission plan.

In Lines 2-6, the UPDATE_MACROS function computes the largest feasible duration of the group plan. This is done by computing the largest single source shortest path (SSSP) from the start timepoint to all other timepoints. This computation is done using the Bellman-Ford SSSP algorithm. If the Bellman-Ford algorithm detects any negative cycles (i.e. detects any temporal inconsistency) then the UPDATE_MACROS function returns false in Line 5. The Bellman-Ford SSSP algorithm places the shortest path distances for each timepoint into an array d. The largest feasible duration of the group plan is equal to the maximum SSSP distance as computed in Line 6.

In Lines 7-8, the UPDATE_MACROS function computes the smallest feasible duration of the group plan. This is done by computing the smallest single destination shortest path (SDSP) from all the timepoints in the plan to the start timepoint of the group plan using the Bellman-Ford Single-Destination Shortest-Path (SDSP) algorithm. The SDSP distances are placed in the array d. Note that SDSP distances are all less than or equal to zero, reflecting the fact that all the timepoints must occur at or after the start

timepoint of the plan. The smallest feasible duration of the group plan is equal to the negation of the minimum SDSP distance contained in the array d.

In Lines 9-11, the UPDATE_MACROS function updates the timebounds of the corresponding macro in the mission plan. In Line 9, it gets the macro corresponding to the group plan g using the GET_MACRO function. In Lines 10-11 it updates the edges of the mission plan's distance graph associated with the macro. The start timepoint of the macro is found using the GET_START function and the end timepoint is found using the GET_END function. Recall that the UPDATE_EDGE function only updates the edge if the new value is less than the current value. If the UPDATE_MACROS function reaches Line 13, then it returns true.

Let's continue with our simple two-layer example. The two layer plan shown in Figure 3-28 contains two group plans. For group plan1, the maximum SSSP distance is 10 for the path ABC, and the minimum SDSP is -2 for the path CBA. The UPDATE_MACROS function leaves the distance of the contingent edge AB in the mission plan at 10; however, the distance of the contingent edge BA is updated to -2. For group plan2, the maximum SSSP distance is 4, for the path ABC, and the minimum SDSP is 0, for the path CBA. The UPDATE_MACROS function updates the distance of the mission plan's contingent edge CD to 4, and the contingent edge DC remains at -1. Both group plans are temporally consistent; therefore, the UPDATE_MACROS function returns true. The mission plan after calling UPDATE_MACROS is shown in Figure 3-31.

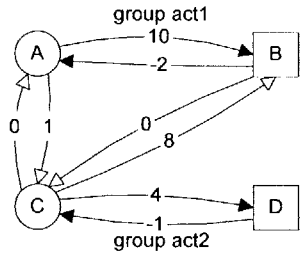


Figure 3-31 The UPDATE_MACRO function updates the edges associated with the macros in the mission plan. AB is updated to 9 and CD is updated to 4.

Lines 4-7 of the HR algorithm computes the All-Pairs Shortest-Path graph (APSP-graph) of the mission plan's distance graph (returning false if the mission plan is temporally inconsistent). Then the HR algorithm updates the timebounds of the group plans if the edges associated with the macros are tightened by the APSP-graph. The COMPUTE_APSP_GRAPH function in Line 4 computes the APSP-graph given the mission plan's distance graph. This APSP-graph is maintained separately from the mission plan's distance graph. The APSP computation is done by either Johnson's algorithm or Floyd-Warshall algorithm [CLR]. Johnson's algorithm is used if the edges the edges are stored in an adjacency list and The Floyd-Warshall algorithm is used if the edges are stored in an adjacency matrix.

The APSP-graph is computed for two purposes. First, it checks if the mission plan is temporally consistent (if the mission plan is inconsistent then the algorithm returns false in Line 6). Second, the APSP-graph is used to deduce any tightenings on the macros induced by the constraints in the mission plans distance graph. If the edges in the APSP-graph corresponding to the macro edges are tightened, then the HR algorithm updates the corresponding group plan. The group plans are updated by calling the UPDATE_GROUP_PLANS function in Line 7 of the HR algorithm.

```
function UPDATE_GROUP_PLANS( M, G,)
Input: A mission plan M and set of group plans G
Effects: Uses the mission plan's APSP-graph to update the macros in the mission plan
and timebounds of the group plans.
1  for each macro ∈ Macros[M]
2      s ← GET_START(macro)
3      e ← GET_END(macro)
4      ub ← apsp_graph[s,e ]
5      lb ← -apsp_graph[e, s ]
6      UPDATE_EDGE( M, (s,e), ub )
7      UPDATE_EDGE( M, (e,s), -lb )
8      g ← GET_GROUP(macro)
9      s ← GET_START(g)
10     e ← GET_END(g)
11     if ub < GET_UB( macro )
12         UPDATE_EDGE( g, s, e, ub )
13     end if
14     if lb < GET_LB( macro )
15         UPDATE_EDGE( g, e, s, -lb )
16     end if
17 end for
14 return TRUE
```

Figure 3-32 Pseudo Code for UPDATE_ GROUP_PLANS

The pseudo code for the UPDATE GROUP_PLANS function is shown in Figure 1-24. This function loops through each macro in the mission plan. In Lines 2-5, it finds the corresponding distances in the APSP-graph for each macro. In Lines 6-7, it uses these distances to update the corresponding edges in the mission plan's distance graph. Then in Lines 8-10, the UPDATE_GROUP_PLANS function finds the group plan, g, associated with the macro and the start timepoint, s, and, end timepoint, e, of that group plan. In Lines 11-16, it uses the lower bound (lb) and upper bound (ub) as computed in Lines 6-7 to update the feasible duration of the group plan. If the ub is smaller than the group plan's maximum feasible duration, then the group plan' edge from the start of the group plan to the end of the group plan is updated. Similarly, if the lb is smaller than the

group plan's minimum feasible duration, then the group plan's edge from the end timepoint to the start timepoint is updated. After calling the UPDATE_GROUP_PLANS function, the data in the macros and group plans are synchronized.

Let's continue with our simple two-layer plan. The mission plan's APSP-graph is shown in Figure 3-33(a). The APSP-graph edge AB is smaller than the edge AB in the mission plan's distance graph. This edge AB is associated with the upper bound on the group act1. The edge AB is tightened from 10 to 9. The UPDATE_GROUP_PLANS function updates the mission plan's distance graph accordingly as shown in Figure 3-33(b). The UPDATE_GROUP _PLAN function then updates the group plans. The updated group plans are shown in Figure 3-33(c-d). The UPDATE_GROUP_PLANS function adds the edge AC = 9 to group plan1, corresponding to the edge AB = 9 in the mission plan, and adds the edge CA = -1 to group plan2, corresponding to the edge DC = -1 in the mission plan. Note that the edge DC = -1 was present in the original mission plan, whereas the edge AB = 9 was derived by the APSP-graph.

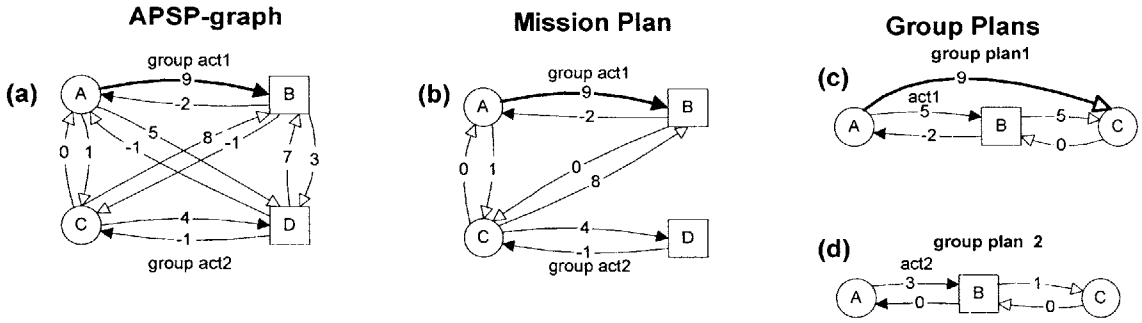


Figure 3-33 (a) APSP-graph (b) Updated mission plan (c) Updated Group Plan 1 (d) Updated group plan 2.

After the group plan's timebounds are updated, the HR algorithm calls the fast dynamic controllability (FAST_DC) algorithm to reformulate each group plan into a dispatchable group plan on Line 9. This FAST_DC algorithm is presented in Chapter 4. If this reformulation succeeds, then the group plan is dynamically controllable and the HR algorithm continues. However, if the FAST_DC algorithm fails (for any group plan) then the HR algorithm terminates and returns FALSE.

The complete description of the FAST_DC algorithm is presented in Chapter 4. For now the reader only needs to understand that the FAST_DC algorithm is a reformulation algorithm that either adds or tightens the constraints of the group plan. These additional constraints may alter the range of feasible durations of the group plan. If the range of feasible durations of a group plan is tightened (the lower bound is increased or the upper bound is decreased), then the HR updates the edges of the corresponding macro by once again calling UPDATE_MACROS. This is done in Line 14. There is no need to check for temporal consistency of the group plans, because if the group plans are dynamically controllable, then they are temporally consistent. Note that tightening the constraints of

the macros only serves to remove uncertainty from the mission plan. Thus, the update in Line 14 only serves to make the decoupling algorithm more likely to succeed.

In our simple two-layer plan example, both of the group plans are dynamically controllable. Furthermore, feasible durations of the group plans are unchanged by the FAST_DC algorithm; therefore, the UPDATE_MACROS call in Line 14 of the HR algorithm does not change the mission plan.

In Line 15, the HR algorithm calls the decoupling algorithm on the mission plan. Recall that the decoupling algorithm operates on the mission plan in order to assign a fixed schedule for the start of each group plan. If the decoupling algorithm succeeds, then the HR algorithm returns true, otherwise the HR algorithm returns false.

For our simple two-layer plan, the decoupling algorithm succeeds. Recall that the decoupling algorithm uses a series of transformations build a constraint network that only contains a set of executable timepoints. The decoupling algorithm is applied to the updated mission plan as shown in Figure 3-33(b). The distance graph of the mission plan is converted in the transformed STN as shown in Figure 3-34(a). The decoupling algorithm first copies over the executable timepoints, A and C, then it transforms the edges using the strong controllability transformation rules. The decoupling algorithm copies over the requirement edges AC = 1 and CA = 0 from the mission plans distance graph. The edge CB = 8 is transformed in to an edge CA = -1 which relaxes the edge CA in the transformed STN. The edge BC = 0 is transformed into an edge AC = 2, which is greater than the existing edge, so there is no change in the transformed STN. Finally, the decoupling algorithm computes the earliest execution time for each timepoint using a SDSP computation. The earliest execution time for A = 0 and B = 1; therefore, the start timepoint associated with group plan1 is fixed at 0, and the start timepoint for group plan2 is fixed at 1. The decoupled group plans are shown in Figure 3-34(b)-(c).

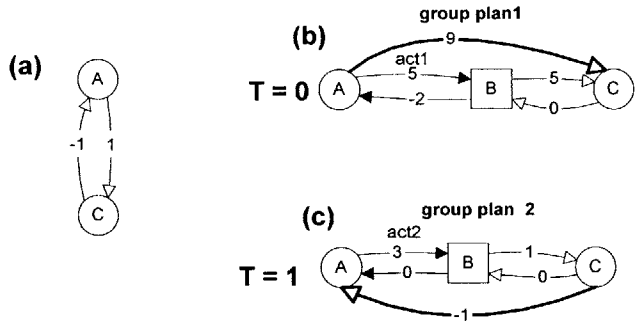


Figure 3-34 (a) The transformed STN (b) The start time of group plan1 is fixed at T = 0 (c) The start time of group plan2 is fixed at T = 1.

The HR algorithm is a polynomial time algorithm. It gains efficiency by dividing the reformulation problem into a set of smaller sub-problems. Let's analyze the runtime complexity of the HR algorithm. In this discussion we use the following notation.

- G = number of group plans, also equal to the number of macros in the mission plan.
- N_m = number of timepoints in the mission plan.
- E_m = number of edges in the mission plan.
- N_g = maximum number of timepoints in any group plan.
- E_g = maximum number of edges in any group plan.

In Line 1 HR calls the UPDATE_MACROS function. The UPDATE_MACROS function loops through each group plan and the time of each loop is dominated by the Bellman-Ford algorithm. Therefore, the UPDATE_MACROS runs in $O(G*N_g*E_g)$. Lines 2-3 of the HR algorithm run in constant time. In Line 4 the HR algorithm calls COMPUTE_APSP_GRAPH. The Floyd-Warshall algorithm is used, which runs in $\Theta(N_m^3)$. Lines 5-6 run in constant time. Line 7 calls the UPDATE_GROUP_PLANS function. The UPDATE_GROUP_PLANS function loops through each macro and in each loop is performed in constant time. Therefore, the UPDATE_GROUP_PLANS runs in $\Theta(G)$ time. Lines 8-13 of the HR algorithm loop through each group plan and calls the FAST_DC algorithm. The time complexity of the FAST_DC algorithm has yet to be formally proven; however, experimental results has shown the running time to be $O(N_g^3)$. Therefore, the running time of Lines 8-13 is experimentally shown to be $O(G*N_g^3)$. Line 14 calls the UPDATE_MACROS function. Finally, in Line 15 the HR algorithm calls DECOUPLE, which we showed is Section 3.5, to run in $O(G*N_g)$ time.

Adding the terms together we get an expression for the running time of the HR algorithm as $O(G*N_g*E_g) + O(N_m^3) + O(G) + O(G*N_g^3) + O(G*N_g) + O(1)$, which can be simplified to $O(G*N_g^3 + N_m^3)$. As we shall see in Chapter 4, the N_g^3 term is derived by an All-Pairs Shortest-Path (APSP) computation applied to the group plan used in the FAST_DC algorithm and the N_m^3 term is due to the APSP computation on the mission plan.

3.7 Summary

In this chapter we, 1) formally defined communication controllability, 2) formally introduce the two-layer MTPNU, 3) introduce a novel Hierarchical Reformulation algorithm that enables teams of agents to coordinate their activities without communication while allowing the agent within each team to dynamically adapt to uncertainty. In the next chapter we discuss our new Fast-Dynamic controllability algorithm which is used to reformulate each group plan.

4 Fast Dynamic Controllability Algorithm

4.1 Introduction

This chapter finishes the explanation of the Hierarchical Reformulation Algorithm introduced in Chapter 3, by describing how to reformulate each group plan into a dispatchable group plan. Specifically, this chapter introduces a novel efficient, centralized, dynamic controllability algorithm, called a *fast dynamic controllability algorithm* that transforms a plan constrained by an STNU into a dispatchable form. This chapter also describes a new edge filtering algorithm that transforms the dispatchable group plan into an efficiently dispatchable plan, called *a minimal dispatchable plan.* Together, the fast dynamic controllability algorithm and edge filtering algorithm perform *group plan reformulation.*

The goal of group plan reformulation is to enable the dispatcher to efficiently, dynamically, and consistently execute the group plan. The reformulation algorithms presented in this chapter are analogous to the reformulation algorithm, described in Chapter 2. Recall that in Chapter 2 we considered plans constrained by a STN; however, here we consider plans constrained by a STNU. We need to deal with uncertainty. Recall that in Chapter 3 we dealt with this uncertainty of the activities in the mission plan by decoupling the activities. This decoupling procedure enabled the activities in the mission plan to be executed independently. However, in this chapter we seek to precompile the temporal constraints of the group plan such that the agents can react to the uncertainty at execution time, in order to exploit the fact that the agents can communicate within the group plans. We seek to preserve some flexibility in the group plans so they can react to their situation at execution time, rather than simply preparing for the worst.

After the reformulation, the agents of the group must cooperate in order to execute the group plan. In the simplest approach, each group plan is executed using a leader-follower architecture. In this approach, a single leader is commissioned to make all scheduling decisions. The leader manages the execution process by sending commands to and receiving execution updates from the other agents in the group. All information passes through the leader. In Chapter 5, we present an alternative approach that distributes the execution process such that all the agents take part in the scheduling process. No matter which approach (leader-follower or distributed) is used, the fast dynamic controllability algorithm is still applicable.

This chapter builds on the concepts presented in Chapter 2 and Chapter 3. Specifically, the fast dynamic controllability algorithm expresses the temporal constraints of the group plan as a distance graph, then uses a set of local shortest path computations to reformulate the plan. The shortest path computations are a type of constraint propagation. Recall that constraint propagation is the process of deriving the feasible assignments on one set of variables given a set of constraints on different sets of variables. The fast dynamic controllability algorithm generalizes the strong controllability rules used in Chapter 3.

This chapter introduces one fundamentally new concept, the idea of a conditional constraint, which was originally introduced by [Morris 2001]. A conditional constraint (or wait constraint) is a ternary constraint (i.e. relates three timepoints) that is satisfied either by the passage of a minimum amount of time or through the notification of an event, which ever is sooner. It is similar to a lower bound simple temporal constraint, except that its enforcement is conditioned on the outcome of some other event. We use these types of constraints in everyday life. For example, consider a scenario where you plan on meeting a friend for lunch; however, you are running late. In such a scenario, you may call your friend and tell him you are running late. You would like to eat together, but, you do not want to be inconsiderate, so you ask your friend to wait for at least 20 minutes before ordering. However, if you get there before 20 minutes, there is no need to wait any longer. If your friend agrees, he has agreed on a conditional wait constraint. Your friend will wait for at least 20 minutes or until you arrive, which ever is sooner.

In this chapter, we show how to use these types of conditional wait constraints to preserve flexibility in partially controllable plans so that they can be dynamically executed. Beyond being desirable, the completeness of the fast dynamic controllability algorithm depends on using these conditional wait constraints.

The conditional wait constraint is a fundamental departure from the constraints we have been using, because they are ternary constraints (relate three timepoints), rather than binary constraints (relate two timepoints). Fortunately, [Morris 2001] showed that these conditional constraints can be propagated through the constraint network similar to simple temporal constraints. Furthermore, the introduction of conditional constraints only requires a small change to the STN dynamic dispatching algorithm, as presented in Chapter 2.

The dynamic controllability problem is solved by iteratively applying a set of local constraint propagation rules. Our fast algorithm builds on the basic structure of the dynamic controllability algorithm introduced by [Morris 2001]; however, it removes the need to perform repeated calls to an $O(N^3)$ All-Pairs Shortest-Path (APSP) algorithm.

The speed of the fast dynamic controllability algorithm is derived by two main innovations. First, our new dynamic controllability algorithm filters redundant constraints from the distance graph, up front, which reduces the number of propagations required. Second, we show that after performing a single APSP computation, the temporal constraints are placed in a *pseudo-dispatchable* form. Given this pseudo-dispatchable form, each constraint only needs to be resolved with the constraints that involve timepoints that occur earlier in the plan. Therefore, the constraints *back-propagate* through the distance graph. Applying these back-propagation rules allows the fast dynamic controllability algorithm to incrementally build up the reformulated distance graph, starting from constraints that relate timepoints that occur at the end of the plan, to constraints that relate timepoints that occur early in the plan.

The outline for this chapter is as follows. First we introduce some definitions and concepts related to dynamic controllability. Next, we review the dynamic controllably algorithm introduced by [Morris 2001]. Then we introduce our novel fast dynamic controllability algorithm. Finally, we introduced the new edge trimming algorithm. The empirical results for the new fast dynamic controllability algorithm are presented in Chapter 6.

4.2 Overview

This section reviews some key concepts and introduces several definitions regarding dynamic execution of plans that contain uncertainty. These definitions will be useful in subsequent sections.

A plan is dynamically controllable if there is a viable, dynamic execution strategy to schedule the timepoints in the plan. Recall that an execution strategy is viable if it generates a consistent schedule in all situations, and an execution strategy is dynamic if each scheduling decision is based only on the past.

The goal of dynamic controllability algorithm presented in this chapter is to compile the temporal constraints of the plan into a form such that a dispatcher can use to dynamically execute the plan. This reformulation enables the dispatcher to adapt to the plan's uncertainty at execution time.

Recall that dynamic execution is a scheduling process in which the timepoints of the plan are scheduled in real-time (timepoints are executed and scheduled simultaneously). In order to understand how to do this dynamic execution for plans that contain uncertainty, let's review the general job of the dispatcher and how to dynamically execute plans that do not contain any uncertainty (i.e. plans constrain by STNs rather than STNUs)

The dispatcher, whether applied to plans that contain uncertainty or not, is constantly making two related decisions: 1) what timepoint to execute next, and 2) when to schedule each timepoint. The reformulation algorithm compiles the temporal constraints of the plan in order to enable the dispatcher to make these decisions properly and quickly. This compilation is composed of two tasks: 1) it computes a set of enablement conditions for each timepoint, and 2) it exposes the set of implicit constraints inherent in the original explicit temporal constraints.

In Chapter 2 we presented two reformulation algorithms (a basic version and fast version) along with a compatible dispatching algorithm for plans constrained by an STN [Muscettola 1998a, Muscettola 1998b]. Recall that the basic reformulation algorithm first computes the All-Pair Shortest-Path (APSP) graph of the plan's distance graph, which exposes the implicit constraints, then trims the redundant (dominated) edges. The resulting graph is called the minimal dispatchable graph. Recall that in the fast version, the APSP computation and edge trimming are interleaved. For plans constrained by an

STN, the enablement condition is simply a list of timepoints that must be executed. For each timepoint, after the set of enablement timepoints have been executed, then that timepoint becomes enabled. The set of enablement timepoints for a timepoint X is computed by compiling all timepoints that are related to X by outgoing non-positive edges.

During execution, the dispatcher is free to select any timepoint for execution that is both enabled and alive. A timepoint is enabled if the all of the enablement timepoints have been executed and a timepoint is alive if the current time falls between the timepoints execution window. Every time the dispatcher executes a timepoint it performs two updates. First, it sends a set of enablement messages to all timepoints waiting on that timepoint's execution, and second it uses the constraints in the reformulated distance graph to update the execution windows of neighboring timepoints. [Muscettola 1998a] showed that upper bound updates are propagated via outgoing positive edges and lower bound updates are propagated via incoming non-positive edges.

The reformulation and dispatching algorithms need to be modified to support plans constrained by STNUs. The dispatcher only has partial control over the execution of the timepoints. A plan is only dynamically controllable if the plan does not further constraint the uncontrollable durations. Both the reformulation algorithm and the dispatcher must respect the timebounds of the contingent links. Recall that a contingent link specifies a lower and upper bound on an uncontrollable duration. If the temporal constraints of the plan imply strictly tighter bounds on the uncontrollable duration, then the uncontrollable duration is *squeezed*. Specifically, an uncontrollable duration is *squeezed* if its lower bound is increased or its upper bound is decreased, as illustrated in **Figure 4-1**. If the uncontrollable duration is squeezed, then there exists a situation where the outcome of the uncontrollable duration falls outside of the specified timebounds; therefore, consistency of the execution is dependent on the outcome of some uncertain event.

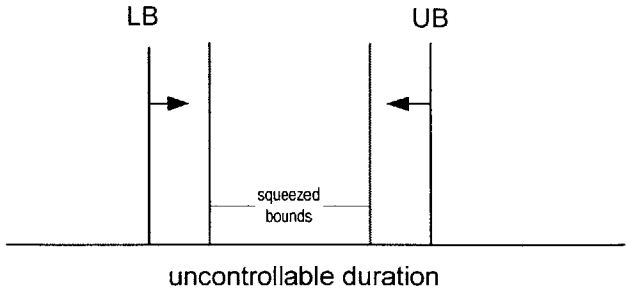


Figure 4-1 Each uncertain duration contains a lower and upper bounds as specified by the associated contingent link. The uncontrollable duration is squeezed if its lower bound is increased or its upper bound of the decreased

[Morris 2001] introduced the concept of *pseudo-controllability*, which provides a first check on the dynamic controllability of a plan. A plan in pseudo-controllable if it is temporally consistent and none of its uncontrollable durations are squeezed. The pseudo-controllability of a plan can be checked by computing the All-Pairs Shortest-Path graph (APSP-graph) of the plan's distance graph (ignoring the distinction between contingent and requirement edges). If the APSP-graph does not contain any negative cycles, and contingent edges remain unchanged in the APSP-graph, then the plan is pseudo-controllable. Therefore, if a plan is pseudo-controllable, then the contingent edges in the plan's distance graph are the shortest paths.

Example 4-1:

Consider the Distance Graph with Uncertainty (DGU) shown in **Figure 4-2**(a). The contingent edges represent the time bounds of the uncontrollable duration AB. The uncontrollable duration will last between [5,10] time units. The path ACB = 9 is shorter than the direct path AB = 10; therefore, the other constraints imply a tighter value on the upper bound of the uncontrollable duration. The APSP-graph shown in **Figure 4-2**(b) exposes this tightening. The uncontrollable duration is squeezed; therefore, the plan is not pseudo-controllable.

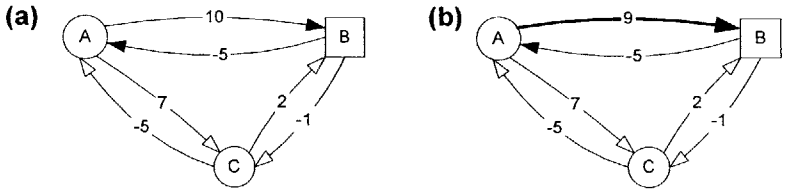


Figure 4-2 (a) The DGU with a uncontrollable duration between timepoints A and B (b) The APSP-graph exposes the temporal constraints imply a tighter upper bound on the uncontrollable duration; therefore, the uncontrollable duration is squeezed.

Example 4-2:

Consider the DGU shown in **Figure 4-3**(a). The contingent edges remain unchanged in the APSP-graph, shown in **Figure 4-3**(b). Furthermore, the plan is temporally consistent; therefore, the plan is pseudo-controllable.

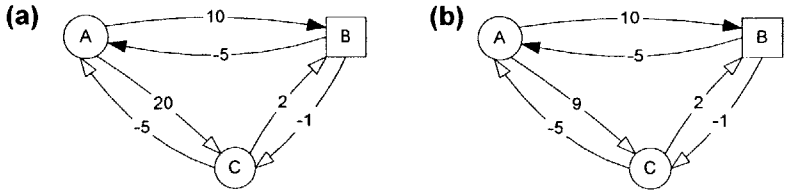


Figure 4-3 (a) A DGU with uncontrollable duration AB. (b) The APSP-graph does not further constraint the contingent edges.

Even if a plan is pseudo-controllable, the uncontrollable durations may be squeezed at execution time. When the dispatcher executes a timepoint, it effectively imposes a rigid constraint between the start of the plan and the timepoint being executed. If the dispatcher were to resolve this new constraint with the other constraints (by computing the APSP-graph) it may tighten a contingent edge, thus squeezing an uncontrollable duration.

Recall that the dispatcher does not need to recompute the APSP-graph every time it executes a timepoint. It updates the execution windows of the timepoints via a set of local propagations rather than updating the constraints of the plan. This is precisely the reason that the dispatcher is able to schedule the network in real-time. Therefore, when we talk about squeezing an uncontrollable duration during execution, it is more natural to express it in terms of the execution windows, rather than in terms of the temporal constraints of the plan.

Given a distance graph with uncertainty (DGU) with a positive upper bound contingent edge, AB, and corresponding contingent lower bound edge, BA, the execution window of the contingent timepoint B is squeezed if the execution window [x, y], resulting from propagation through edges AB and BA is tightened by any other propagation. Specifically, if the propagation through an incoming positive edge CB, where $C \neq A$, results in an upper bound, y', where $y' < y$, then the contingent execution window is upper bound squeezed. Similarly, if a propagation through some outgoing negative edge BC, where $C \neq A$, produces lower bound x', where $x' > x$, then the contingent execution window is lower bound squeezed. If the contingent execution window is squeezed during execution then the uncontrollable duration is also squeezed.

Example 4-3:

Consider the DGU show in Figure 4-4(a). The DGU is pseudo-controllable; however, it the dispatcher chooses execution time for B such that it squeezes the execution window of the contingent timepoint C. Timepoint A is the start of the plan and is executed at time = 0. After executing A, the dispatcher propagates the execution time through the plan. The execution window for C is [1,7] and the execution time for B is [5,10]. The dispatcher is free to choose any execution time for C between [1,7]. Figure 4-4(b) shows a case when the dispatcher chooses an execution time of 5 for B. After executing C, the dispatcher propagates the execution time of C through the edges CB and BC. These propagations result in an execution window for the contingent timepoint B of [8,10]. This squeezes the execution window of B. If the uncertain duration takes any time between 5 and 7 time units, then the execution is inconsistent. Note that if the dispatcher executed B at time 1 or 2, then the contingent execution window would not have been squeezed.

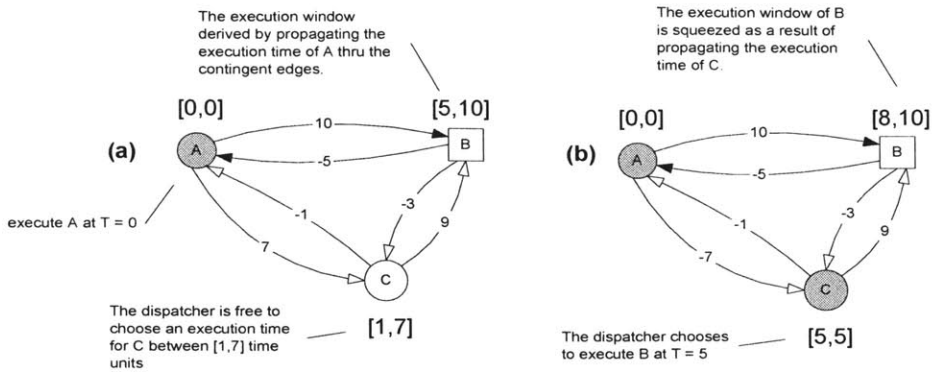


Figure 4-4 (a) The execution windows for the plan are shown after executing A at time = 0. (b) The execution window of the contingent timepoint B is squeezed from [5,10] to [8,10] after executing the timepoint C at time = 5.

The goal of the dynamic controllability algorithm is to add additional constraints to the plan in order to enable the dispatcher to consistently schedule the plan without squeezing consistent timepoints at execution time.

4.3 The Dynamic Controllability Algorithm

This section describes the dynamic controllability (DC) algorithm introduced by [Morris 2001]. The dynamic controllability algorithm transforms an STNU into a dispatchable graph. [Morris 2001] also showed that this algorithm is both sound and complete. If algorithm successfully reformulates the STNU, then the STNU is dynamically controllable; however, it the algorithm fails to reformulate the STNU, then the STNU is not dynamically controllable. In this section, we first present the overall structure of the DC algorithm, we will then describe the details of each step, and finally present the pseudo-code for the DC algorithm along with a brief analysis of the time complexity of the algorithm. In the next section, we present a new, faster, dynamic controllability algorithm, which is used in the Hierarchical Reformulation algorithm.

The dynamic controllability algorithm iteratively applies a set of *reductions* in order to prevent the dispatcher from squeezing the plan at execution time. These reductions are a set of rules that add (or tighten) the constraints to the plan. These reductions are similar to the strong controllability transformation rules presented in Chapter 3. The dynamic controllability algorithm uses a constraint processing loop that iterates between applying the reductions, propagating the effects of the reductions to the other constraints in the plan, and checking if the plan is pseudo-controllable. The DC algorithm loops until either 1) it determines that the plan is not dynamically controllable, by detecting an inconsistency or determining that the plan is not pseudo-controllable, or 2) it converges on a dispatchable graph.

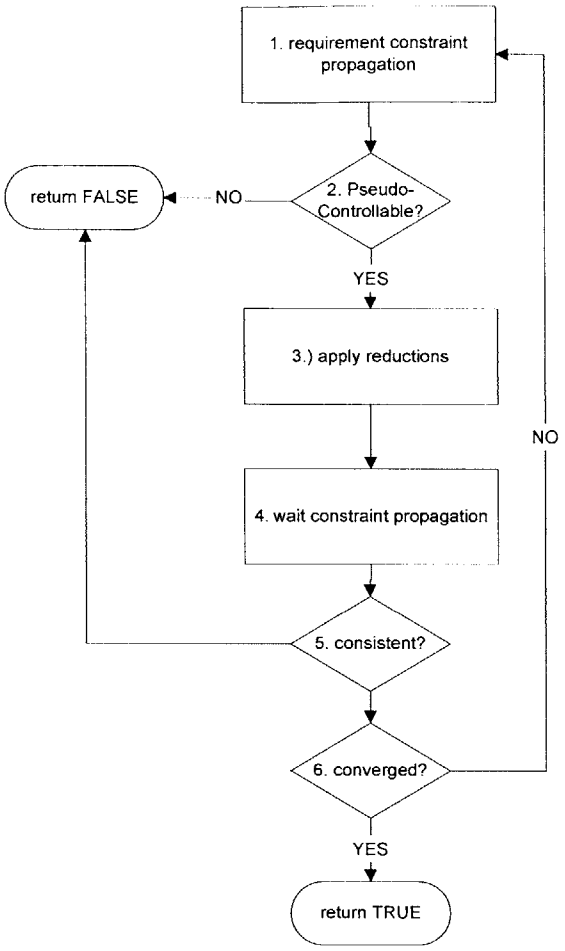


Figure 4-5 Basic Steps of Dynamic Controllability Algorithm

The constraint processing loop iterates between four basic steps: 1. requirement constraint propagation, 2. checking pseudo-controllability, 3. local constraint deduction, and, 4. wait constraint propagation, as shown in **Figure 4-5**. In Step 1, the algorithm resolves the simple temporal constraints by computing the All-Pairs Shortest-Path graph. Conceptually, after the first iteration, Step 1 propagates any change in the requirement constraints throughout the graph. In Step 2, the algorithm checks if any plan is pseudo-controllable. If the plan is inconsistent or any uncontrollable duration has been squeezed, the algorithm returns false. In Step 3, the algorithm applies a set of *reductions* in order to prevent an uncontrollable duration from being squeezed at execution time. The reductions may either modify the simple temporal constraints of the plan or modify the wait constraints of the plan. Finally, the algorithm propagates wait constraints through the plan.[3] If the algorithm determines any inconsistency during this wait constraint propagation, it returns false.

[3] [Morris 2001] called the propagation of the wait constraints regression.

The algorithm loops through these four steps until either 1) the pseudo-controllability checking step fails, 2) the propagation of the wait constraints results in an inconsistency, or 3) the algorithm successfully goes through an iteration of the constraint processing loop without adding (or modifying) the constraints of the plan.

4.3.1 Triangular Reductions

This subsection describes a set of *reductions*, which add (or tighten) the temporal constraints of the plan, in order to prevent the dispatcher from squeezing the contingent execution windows at execution time. [Morris2001] derived the reductions in terms of a triangular STNU; however, here we will derive the reductions using the associated distance graph (DGU). It is more natural to use the distance graph because the dispatcher uses a distance graph to execute the plan.

The reductions are derived from a case analysis of a distance graph of a triangular STNU. The triangular STNU is shown in Figure 4-6(a) and the associated triangular DGU is shown in Figure 4-6(b). Later we show how the reductions derived for the triangular DGU are applied to distance graphs of arbitrary size. The triangular STNU consists of two executable timepoints, A, and C, and one contingent timepoint, B. It contains one contingent link AB $\in$ [x,y], corresponding to an uncontrollable duration, and two requirement links, AC $\in$ [p,q] and CB $\in$ [u,v]. We assume the STNU is pseudo-controllable and the distance graph is in an APSP form. Therefore, each edge in the distance graph corresponds to a shortest path distance.

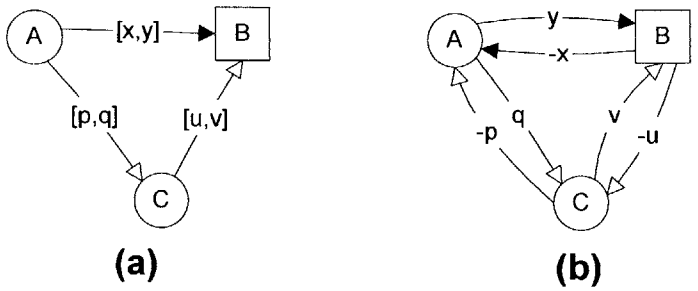


Figure 4-6 (a) The Triangular STNU (b) The associated triangular DGU.

The reductions are used to constrain the execution time of timepoint C, in order to prevent the propagations through CB and BC from squeezing the contingent execution window of B. Recall that the execution window of B can only be squeezed by incoming positive edges and lower bound squeezed by outgoing negative edges. We need to consider three cases. In the *precede* case, timepoint C must be executed before the contingent timepoint B. In the *follow* case, timepoint B must be executed after the contingent timepoint C, and in the *unordered* case, the execution order of B and C is undetermined. Recall that each execution order of a timepoint is determined by considering the negative edges in its DGU, as illustrated in **Figure 4-7**.

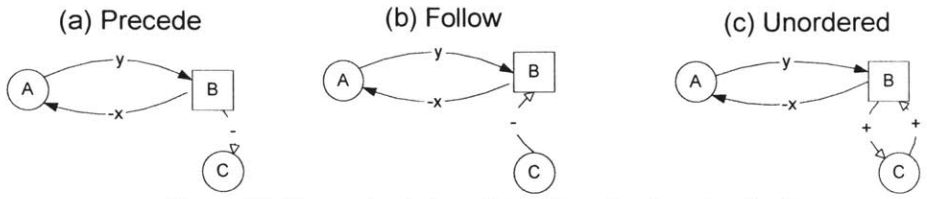


Figure 4-7 Temporal ordering relationships of a timepoint C with respect to a contingent Timepoint B.

Given a DGU, G, in an APSP-form, the order of execution of a timepoint, C, with respect to a contingent timepoint B, is as follows:

- A timepoint C must follow the contingent timepoint B, if there exists a negative edge BC in G.
- A timepoint C must precede the contingent timepoint B, if there exists a negative edge CB in G.
- The execution order of timepoint C is undetermined with respect to the contingent timepoint, B, if both edges BC or CB are non-negative.

Note that if both BC and CB are negative, then there exists a negative cycle and the DGU is inconsistent. In the triangular DGU shown in **Figure 4-6**, the ordering of C with respect to the contingent timepoint B is determined by the sign of u and v.

Follow Case: $u \geq 0$

If $u \geq 0$, then there exists a negative edge BC. In the precede case, timepoint C must be executed after the contingent timepoint B; therefore, the dispatcher is always privy to the execution time of contingent timepoint B when it makes the scheduling decision of C. Therefore, the dispatcher is able to adapt the schedule of C based on the execution time of B. In the follow case, the dispatcher uses edges BC and CB to update the execution window of timepoint C not B. Therefore, the execution of C will never squeeze the execution window of contingent timepoint B. As long as the STNU is pseudo-controllable, the dispatcher will be able to dynamically schedule B. The follow case requires no additional tightening of the constraints.

Example 4-4

Consider a scenario in which a student must meet with an advisor. The advisor's arrival time at the office is uncertain and will take between 5 to 10 minutes. Furthermore, the advisor requires at least 5 minutes to check his email before the meeting; however; the advisor is on a tight schedule so he does not want to wait in his office more than 10 minutes before the meeting. The advisor agrees to notify the student when he reaches his office. The student is willing to wait for up to 20 minutes. The student's plan is shown in **Figure 4-8**(a). The APSP-graph is shown in **Figure 4-8**(b). The APSP-graph is both consistent and the APSP-graph does not tighten the contingent edges; therefore, the plan

is pseudo-controllable. Furthermore, the timepoint C must follow timepoint B; therefore, the plan is dynamically controllable. **Figure 4-8**(c) shows the execution windows after executing timepoint A at T = 0. **Figure 4-8**(d) shows a situation where it takes the advisor 7 minutes to get to his office. This execution time is propagated to timepoint C. The new execution window for C is [12, 17]. The student can successfully execute the plan by getting to the office any time in this execution window.

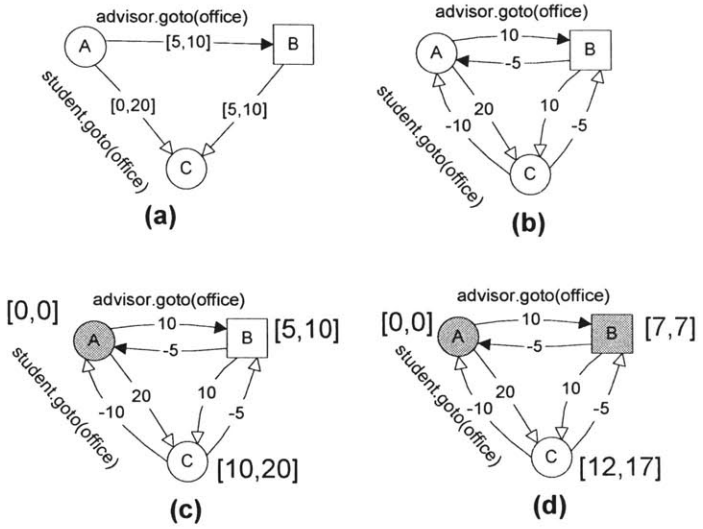


Figure 4-8 (a) In the student's plan, timepoint C must follow the contingent timepoint B. (b) The APSP-graph reveals that the plan is pseudo-controllable. (c) Timepoint A is executed at T = 0 and the execution windows are updated (d) Timepoint B is executed at T = 7, and the execution window for C is updated. In this situation, the student must get to the office some time between 12 and 17 minutes.

Precede Case: v < 0

If v < 0, then there exists a negative edge BC in the triangular distance graph; therefore, B must always be executed before C. In the precede case, the dispatcher will never know the execution of the contingent timepoint B when it needs to make the schedule timepoint C. This is exactly the situation addressed by strong controllability. The dispatcher is not able to adapt the schedule of C based on the execution time of the contingent timepoint B. The edge CB and BC are used to update the execution window of the contingent timepoint C. In order to be dynamically controllable, the algorithm must restrict the execution time of B. Specifically, in order to prevent the contingent execution window from being squeezed by propagations through CB and BC, we need to restrict the execution time of timepoint C with respect to A, by applying the appropriate strong controllability transformation rules. The reductions are simply the executable/contingent and the contingent/executable strong controllability transformation rules as derived in Section 3.5. However, instead of using the rules to compute a new transformed distance graph, as we did in the strong controllability algorithm, here the rules are used to directly

modify the distance graph. Specifically, the precede reductions tightens the edges AC and CA.

(Precede Reduction) *Given a triangular distance graph with uncertainty ABC (as shown in* **Figure 4-6***(b)), with v < 0, the edge AC is tightened to x-u, and the edge CA is tightened to v-y.*

As in the strong controllability case, the precede reduction effectively decouples the timepoint C from the contingent timepoint B. After applying the reduction, any propagation from timepoint C to timepoint B is redundant; therefore, the edges CA and AC can be removed from the distance graph. Any non-redundant information propagated through the edge CB and BC would only serve to squeeze the execution window of the contingent timepoint.

Note that the precede reductions are easily remembered, by first negating and transposing the contingent edges in the distance graph. Next, the shortest paths CBA and ABC are computed.

Example 4-5

Consider the STNU shown in **Figure 4-9**(a). The uncontrollable duration between timepoints A and B will take between 5 to 10 time units, and timepoint C must precede B by 1 to 8 time units. The APSP-graph is shown in **Figure 4-9**(b) is consistent and the contingent edges are not tightened; therefore, the STNU is pseudo-controllable. The edge BC is negative; therefore, C must precede B. In order to prevent the contingent execution window from being squeezed; we need to apply the precede reduction. The precede reduction tightens CA to -2 and AC to 4. **Figure 4-9**(c) shows the tightened distance graph. The edges BC and CB are not dominated. **Figure 4-9**(d) shows the distance graph after removing the dominated edges.

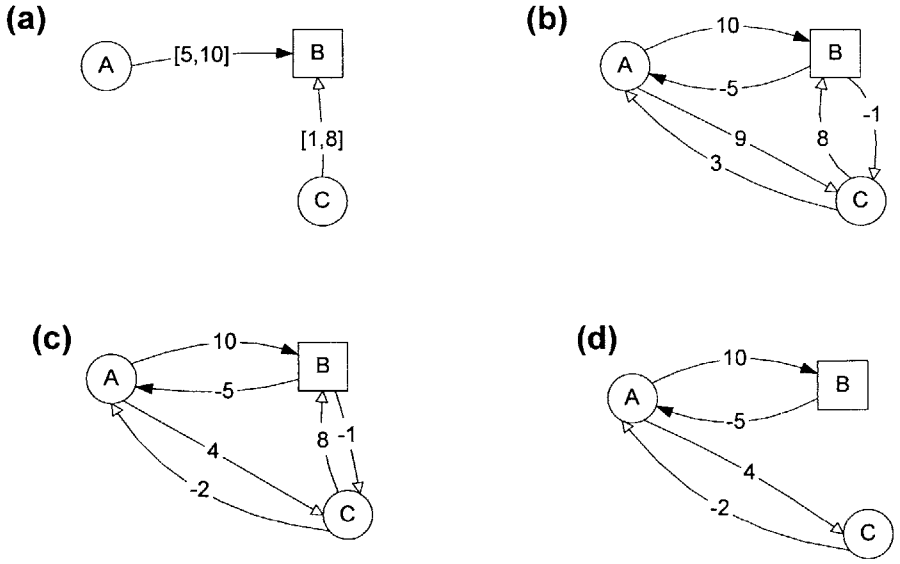


Figure 4-9 (a) The STNU where timepoint C must precede the contingent timepoint B. (b) The APSP-graph of the STNU. (c) The

resulting distance graph after applying the precede reduction. d(CA) + d(AB) = d(CB) and both AB and BC are positive; therefore, CB is dominated. Also, d(BA) + d(AC) = d(BC) and both BA and BC are negative, so BC is dominated. (d) The distance graph after CB and BC are removed.

Unordered Case: $v \geq 0$ and $u \leq 0$

In the unordered case, the edges BC and CB are both positive; therefore, the order of execution of B and C is not *a priori* determined. If C is executed first, then the edge CB is used to update the upper bound of the contingent timepoint B. However, if B is executed first, then the edge BC is used to update the upper bound of C. The simplest way to deal with the unordered case is to unconditionally constrain the execution time of B, in order to prevent the edge CB from squeezing C; this is accomplished by adding edge CA of v-y, as we did in the precede case. However, unconditionally constraining C may prevent the dispatcher from being able to react to the uncertain execution time of B, when B is executed first. Instead, we apply a softer constraint, called a *wait constraint,* which enables the dispatcher to adapt to the schedule of C when B is executed first, yet restricts the execution time of C in order to prevent B from being squeezed.

The wait constraint, written <B, t>, on edge AC specifies that the execution of C must wait for at least t time units after A executes or until B executes, which ever is sooner [Morris 2001]. In the previous example, B is called the *conditional timepoint* and t is the *wait duration.* Here we introduce a slightly different form of the wait constraint, called a *conditional constraint* or *conditional edge*, which encodes the same information as the wait constraint, except that it puts in a form similar to the edges in the distance graph. A conditional constraint is a directed edge that contains a distance expressing a temporal constraint similar to a requirement edge and a conditional timepoint similar to a wait constraint. The conditional constraint is the negative transpose of the wait constraint. A wait constraint <B,t> on an edge AC, corresponds to a conditional constraint of CA of <B,-t>. As in a requirement edge, the temporal distance of the conditional constraint requires that $T(C) - T(A) \leq -t$, which can be rewritten as $T(A) - T(C) \geq t$. If $t \geq 0$, then the conditional constraint encodes a lower bound temporal requirement (i.e. a wait condition) on C with respect to A. Similar to a wait constraint, this temporal requirement is only enforced until the conditional timepoint B is executed. After the conditional timepoint B executes, we say that the conditional constraint is relaxed. Thus, the conditional constraint CA of <B,-t> specifies that C must wait for at least t time units after A executes or until B executes, which ever is sooner.

In the unordered case, we apply a conditional unordered reduction, as defined below, which introduces a conditional constraint to the plan.

(Conditional Unordered Reduction) *Given a triangular distance graph with uncertainty (as shown in Figure 4-6(b)), where $v \geq 0$ and $u < 0$, apply a conditional constraint CA of <B, v-y>.*

After applying the conditional unordered reduction, if B executes first (follow case), then the conditional constraint is relaxed (i.e. the temporal requirement imposed by the conditional constraint no longer needs to be satisfied) and the dispatcher can react to the execution time of B. However, if C is executed first (precede case), then the temporal requirement of the conditional constraint ensures that the propagation from C will not squeeze the execution window of B.

Example 4-6

Here we revisit the student-advisor meeting problem with a slightly different temporal constraints. The advisor's arrival time is still uncertain. It will take him between 5 and 15 minutes to get to his office, and the student is willing to wait for up to 20 minutes before getting to the office. Both the student and the advisor will only wait a small amount of time in the office. The student will not wait more than 5 minutes after getting to the office, and the advisor, being more impatient, will wait no more than 1 minute. Furthermore, the student and advisor agree to call one another when they reach the office.

The student's plan for this scenario is shown in **Figure 4-10**(a). The APSP-distance graph is shown in **Figure 4-10**(a). The APSP-graph is consistent and the contingent edges are not squeezed; therefore, the plan is pseudo-controllable.

Consider the student's execution strategy. If the student gets to the office anytime before 10 minutes, he runs the risk that he will be waiting more than 5 minutes before the advisor arrives. For example, if the student only waits for 6 minutes, the student will be waiting for more than 5 minutes in a situation where the advisor arrives any time between 11 and 15 minutes. However, if the student unconditionally waits for 10 minutes, the advisor may be waiting around for more than 1 minute after he arrives. For example, if the student waits for 10 minutes and the advisor arrives in 7 minutes, then the advisor will be waiting around for 3 minutes. There is no unconditional strategy for successfully scheduling the arrival time of the student. Applying the conditional unordered reduction encodes a conditional execution strategy. The conditional constraint CA <-10,B> (dashed line), shown in **Figure 4-10**(c), specifies that the student must wait for at least 10 minutes or until the advisor arrives. This enables the student to successfully execute the plan.

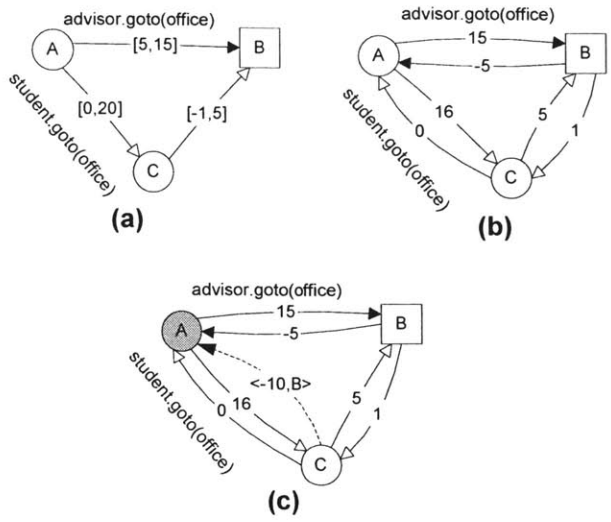


Figure 4-10 (a) The student's plan where the execution order of B and C is unordered (b) The APSP-graph of the student's plan (c) The distance graph after applying the conditional unordered reduction

In order to formally incorporate the conditional constraints with the Distance Graph with Uncertainty (DGU), we introduce a Conditional Distance Graph with Uncertainty (CDGU). The CDGU is a DGU that contains a set of conditional constraints. The dispatcher uses the information contained in the CDGU while executing the plan. The distance graph shown in **Figure 4-10**(c) is a CDGU.

Definition (CDGU): *A CDGU is a 5-tuple* $<N_{ctg},N_{exe},E_{req},E_{ctg},E_{cond}>$ *where* N_{ctg} *is a set of contingent timepoints,* N_{exe} *is a set of executable timepoints,* E_{req} *is a set of requirement edges,* E_{ctg} *is a set contingent edges, and* E_{cond} *is a set of conditional edges.*

There is one important case when a conditional constraint is actually unconditional. In this case the conditional constraint is converted into a requirement edge. Specifically, a conditional constraint is unconditional if the lower bound of the uncontrollable duration associated with a conditional timepoint is greater than the wait duration specified by the conditional constraint. In this case, the conditional timepoint will never be executed before the wait period is completed; therefore, the dispatcher must always wait the full duration, as specified by the conditional constraint. The *unconditional unordered reduction* specifies when a conditional constraint is converted into a requirement edge.

(Unconditional Unordered Reduction) Given a CDGU with conditional constraint CA of <B,-t>, and an uncontrollable duration AB $\in$ [x,y] associated with the conditional timepoint B, if x > t, then the conditional constraint CA is converted into a requirement CA with distance –x.

Note that the unconditional unordered reduction always applies when the temporal distance of the conditional constraint is positive. Therefore, after applying the unconditional unordered reduction, only negative conditional constraints remain.

Example 4-7:

Consider the CDGU shown in Figure 4-11(a). The conditional constraint CA of <-4,B> derived by the conditional unordered reduction. The conditional constraint specifies that the dispatcher must wait to execute C for at least 4 time units after A is executed or until B is executed. However, the contingent edge CA specifies that B will never execute before 5 time units. Therefore, by the unconditional unordered reduction, the conditional constraint CA is converted into a requirement edge CA of distance -4. Figure 4-11(b) shows the resulting CDGU after applying this requirement constraint to the distance graph.

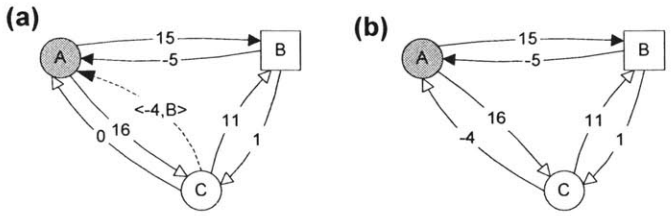


Figure 4-11 (a) A CDGU with conditional constraint CA <-4,B> where lower bound of the uncontrollable duration, 5, is greater than the wait period, 4, of the conditional constraint (b) The unconditional unordered reduction converts the conditional constraint CA of <-4,B> in to a requirement constraint CA of -4.

In this section we reviewed three reductions[4] for triangular STNUs: the precede, conditional unordered, and unconditional unordered reductions. These reductions prevent the dispatcher from squeezing the execution window of the contingent timepoint, while allowing dispatcher to react to the uncertain execution time of the contingent timepoints. If the reductions do not violate the pseudo-controllability of the STNU, then the triangular STNU is dynamically controllable [Morris 2001]. For STNUs of more than three timepoints, the triangular reductions are applied for each triangle that appears in the STNU.

In the next subsection, we introduce a technique, called *regression,* which allow us to determine if the introduction of a conditional constraint violates the pseudo-controllability of the STNU. Regression also serves to enable us to handle conditional constraints for STNUs of more than three timepoints.

4.3.2 Regression of Conditional Constraints

[Morris 2001] showed that conditional constraints need to be propagated through the distance graph. The propagation is a type of constraint propagation that resolves the conditional constraint with the other constraints in the plan. The propagation of a conditional constraint is called *regression.* This regression serves two purposes. First it detects if the conditional constraint is inconsistent with the other constraints of the plan,

[4] [Morris 2001] also introduced a general unordered reduction; however, it is unnecessary.

and second, it ensures that the conditional constraint will not be violated at execution time.

Example 4-8

Consider the distance graph shown in Figure 4-12(a). The conditional constraint CA of <-7,B> may be inconsistent if D propagates an upper bound to C that is less than 7 time units. At execution time, if D is executed at a time before 5 time units, then the propagation through DC requires C to be executed before 7 time units; hence, violating the lower bound imposed by the conditional constraint CA. However, if we impose a conditional constraint DA of <-5,B>, thereby restricting the execution time of D as shown in Figure 4-12(b), the original conditional constraint CA can not be violated. Note that the constraint DA that restricts the execution time of D only needs to be conditional because, once B is executed, the original conditional constraint CA is relaxed; thus it no longer needs to be protected. Also note that the new conditional constraint DA is computed using a similar method to that used for requirement constraints; the value of conditional constraint is equal to the shortest path DCA.

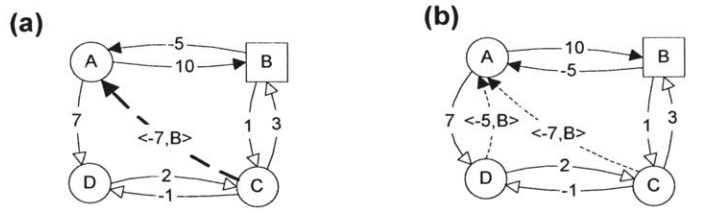


Figure 4-12 (a) The conditional constraint CA is potentially violated by the incoming positive edge DC (b) Imposing a conditional constraint of DA of <-5,B> prevents the original CA from being violated at execution time.

In general, a conditional constraint CA is potentially violated by incoming positive edges in the timepoint C. For a Conditional Distance Graph with Uncertainty (CDGU), there are two types of positive incoming edges: requirement and contingent edges. Note that conditional edges are always negative (any positive conditional edge is converted into a requirement edge by the unconditional unordered reduction). The regression lemma below specifies the means to resolve the potential consistency violations for both cases. For the requirement edge, the conditional edge is regressed using a type of shortest path computation, as illustrated in the previous example. For a contingent edge, the conditional edge is regressed using a slight variation of the precede reduction. For the contingent case, the conditional edge must be regressed, in order to ensure that it will be satisfied for all situations. The regression lemma stated below is a variation of the regression lemma introduced by [Morris 2001].

(Regression): *Given a conditional constraint CA of <B,-t>, where t is less than or equal to the upper bound of AB. Then (in a schedule resulting from a dynamic strategy):*

i.) If there isa requirement edge DC with distance w, where $w \geq 0$ and $D \neq B$, we can deduce a conditional constraint DA of <w-t, B>.

ii.) If t >= 0 and if there is a pair of contingent edges DC, of distance y, and CD, of distance -x, where x, y ≥ 0 and B ≠ C, then we can deduce a conditional constraint DA of <x-t, B>.

The first regression rule is applied when a conditional edge is threatened by an incoming positive requirement edge. The conditional edge is regressed through the incoming positive requirement edge, except when the requirement edge originates from timepoint B (i.e. D = B)[5]. The regression ensures that the wait period encoded in the conditional constraint CA of <B,-t> is never in conflict with an upper bound propagated by the incoming positive edge. The conditional constraint does not need to be regressed through an edge originating from B because, in order for the dispatcher to propagate an upper bound from B, B must be executed. When B is executed, the conditional constraint CA is relaxed (i.e. the temporal requirement is removed from the plan). The upper bound propagated from B can not be inconsistent with a constraint that is no longer exists.

If we were to regress a conditional edge CA of <B,-t> through an edge originating from B, the regression produces a new conditional constraint BA of <B,-x>. This new conditional constraint, BA, would require B to wait x amount of time after A executes or until B executes. The constraint imposes a nonsensical constraint in which B is waiting on itself to execute. One could argue that this constraint precludes B from executing until the full wait period of x as come to pass or one could argue that simply executing B satisfies the constraint; therefore, the conditional constraint is satisfied no matter when B executes. Rather than engaging in a philosophical debate, we simply restrict the regression such that this type of constraint never arises.

The second regression rule is applied when a conditional constraint CA of <B,-t> is threatened by an outcome of an uncontrollable duration. If an uncontrollable duration DC ∈ [x,y] occurs early, such that the execution of C happens before the imposed wait period of t expires, then the conditional constraint is violated. The regression imposes a new conditional constraint on the start of the uncontrollable duration, timepoint D, in order to ensure that the original conditional constraint CA will be satisfied for all situations. The conditional edge is satisfied in all situations if it is satisfied in the worst situation. The worst situation occurs when uncontrollable duration DC occurs at its earliest possible time (i.e. at its lower bound of x). Imposing a conditional constraint of DA <x-t, B> ensures that even when an uncontrollable duration occurs at its lower bound, the conditional constraint CA will not be violated. The distance (x-t) of the new conditional constraint DA is derived by treating the conditional edge as a requirement edge and applying the precede reduction.

Note the distance of the original conditional constraint CA is always less than zero (if distance is positive, then the conditional constraint is converted into a requirement edge per the unconditional reduction rule). Therefore, A must occur before C and the precede reduction rule applies. Therefore, the new conditional constraint applied through regression is only conditioned on the outcome of B. In other words, applying the

[5] [Morris 2001] did not include this exception.

unconditional reduction to positive conditional constraints prevents the regression from introducing a conditional constraint that is conditioned on more than one timepoint.

Regressions are applied recursively until no more regressions are possible. This process is called *full regression.* Each conditional edge introduced by the conditional unordered reduction needs to be regressed through all incoming positive edges. The regression of a conditional constraint through an incoming positive edge leads to either a new conditional constraint or a new requirement constraint (after applying the unconditional unordered reduction). In general, if the regression introduces a new conditional constraint, then that new conditional constraint needs to be regressed. A new conditional constraint does not need to be regressed under three cases: 1) The new conditional constraint is converted into a requirement edge by the unconditional unordered reduction, 2) the new conditional constraint is self looping (the start and end timepoint of the conditional edge are the same) or 3) there are no incoming positive edges to necessitate further regression.

One interesting case arises when the conditional constraint is converted into a positive requirement edge by the unconditional unordered reduction. If the new requirement edge is positive, then it potentially violates a conditional edge. In this case, any conditional constraint threatened by this new positive requirement edge must be regressed through it.

The regression may expose a temporal inconsistency. Specifically, if the regression imposes a self-looping (conditional or requirement) edge with negative distance (i.e. a negative cycle), then the plan constrained by the CDGU is not dynamically controllable. Note that full regression is not in itself sufficient to determine the dynamic controllability of the plan. The regression may introduce a new requirement edge that compromises the pseudo-controllability of the plan; however, it is only detected by resolving the new requirement edge with all the other constraints in the plan. Regression only resolves this new requirement constraint with the conditional constraints of the plan. The mechanism used by [Morris 2001] to detect the potential consistency violations is to recompute the APSP-graph and to recheck if the plan is pseudo-controllable. In the next section, we present a novel scheme to interleave the constraint propagation of requirement constraints with conditional constraints. This new scheme does not depend on recomputing the APSP-graph.

Example 4-9

Consider distance graph shown in **Figure 4-13**(a). In order to prevent the execution window of the contingent timepoint B from being squeezed at execution time, we apply the conditional unordered reduction to the triangle ABC. This introduces a conditional edge CA of <-7,B>, as shown in **Figure 4-13**(b). Note that other reductions are applicable, including the conditional unordered reduction on triangle DCB; however, these reductions are not applied for clarity.

This conditional edge CA needs to be regressed through all incoming positive requirement edges not originating from the conditional timepoint B, and any uncontrollable durations terminating on C. In our example, the conditional edge CA is

regressed through the requirement edge AC, and the uncontrollable duration DC. The regression through AC with distance 9 results in a new self looping conditional edge AA of <1,B>. The regression of the conditional edge CA through the uncontrollable duration DC results in a new conditional edge DA of <-4,B>. The results of these regressions are shown in **Figure 4-13**(c).

This conditional edge AA is converted into a requirement edge by the unconditional unordered reduction, because the distance of the conditional edge is positive. Fortunately, this new requirement edge does not introduce a negative cycle into the CDGU. The distance of the conditional edge DA is -4, which imposes a wait of 4 time units between A and D, which is less than the lower bound of the uncontrollable duration AB of 5. Therefore, the conditional constraint DA of <-4,B> it is converted into a requirement edge DA with distance 4 by the unconditional unordered reduction. The results of these reductions are shown in **Figure 4-13**(d). Note that there now exists a negative cycle between AD; however, this is not detected during regression.

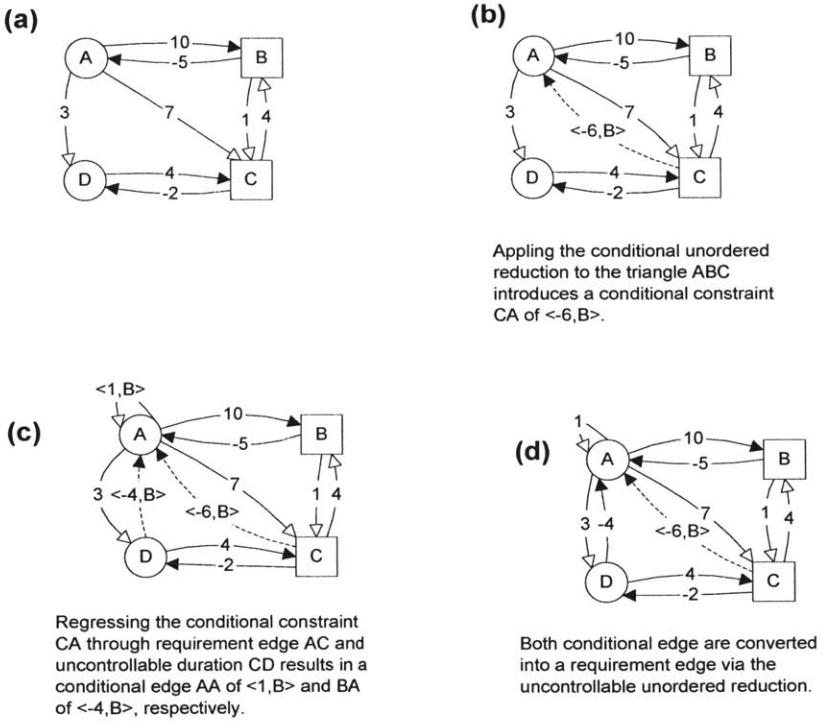


Figure 4-13 (a) A four timepoint DGU. (b) The CDGU after appling the conditional unordered reduction to triangle ABC. (c) The CDGU after regressing the conditional edge CA through AC and DC. (d) The CDGU after converting the conditional constraints to requirement edge via the unconditional unordered reduction

Several important patterns arise during regression. First, all conditional edges introduced by regressing a conditional edge CA of <B,-t> always points to C and the regression rules prevent a conditional edge of BA; therefore, there is at most N-1 conditional edges conditioned on B. For a plan containing P uncontrollable durations,

there can be at most P*(N-1) conditional constraints in the plan. Second, the only way two conditional constraints can exist between the same timepoints is when the uncontrollable durations start the same timepoint. Third, the regression always increases the temporal distance of the conditional constraint (i.e. progressively imposes a less restrictive constraint).

In this subsection we presented a constraint propagation technique, called regression, that enabled us to resolve the conditional constraint with other constraints in the plan. Regression enabled the ternary conditional constraints to be propagated similar to simple requirement edges. In the next subsection, we combine a pseudo-controllability checking algorithm, with the triangular reductions and regression, to form the dynamic controllability algorithm.

4.3.3 Pseudo-Code for the Dynamic Controllability Algorithm

The following completes the description of the dynamic controllability (DC) algorithm [Morris 2001] by presenting the pseudo-code. The pseudo-code for the DC algorithm is shown in **Figure 4-14**. Dynamic controllability transforms the STNU into a dispatchable CDGU, if this reformulation is successful, then the algorithm is dynamically controllable and returns true, otherwise the DC algorithm returns false. Recall that the general structure of the DC algorithm is described in the flow diagram shown in **Figure 4-5**.

Line 1 computes the associated distance graph, G, of the STNU, Γ. The DC uses the distance graph formulation of temporal constraints. In Line 3 the DC algorithm first computes the All-Pair Shortest-Path graph (APSP-graph) of the distance graph G while ignoring the distinction between contingent and requirement edges. Line 4 checks if the plan is pseudo-controllable by calling the IS_PSEUDO_CONTROLLABLE function. If any contingent edges are squeezed or if any negative distance graph contains a negative cycle, then the IS_PSEUDO_CONTROLLABLE function returns false; otherwise, it returns true. If the plan is not pseudo-controllable, the DC algorithm returns false. Recall that if the plan is not pseudo-controllable, then the plan is not dynamically controllable. Line 6 initializes the variable modified to false. This variable is used to determine if the algorithm converges.

Lines 7-14 loops through all possible triangles, ABC, that contain a contingent timepoint B, and applies any tightening required by the precede reduction, and any conditional constraint required by the conditional unordered reduction. If the reductions tighten or add a new constraint to the distance graph, then the algorithm assigns the variable modified to true and breaks out of the loop. If the algorithm loops through all possible triangles and the constraints of the distance graph are not modified, the variable modified remains false.

In Line 15 The REGRESS_WAITS function applies all possible regressions of conditional constraints, while converting the conditional constraints to requirement constraints when the unconditional unordered reduction applies. If the regression introduces a temporal inconsistency, then the REGRESS_WAITS function returns false;

otherwise, it returns true. If the REGRESS_WAITS function returns false, then so does the DC algorithm in Line 16.

Line 17 checks if the regression modified the constraints (conditional, requirement, or contingent) of the distance graph by calling the function IS_MODIFIED. The variable modified is true if the distance graph is modified, by either applications of the reductions or regression.

The algorithm loops through Lines 2-17, tightening the edges of the distance graph until the algorithm converges on a dispatchable CDGU (the algorithm completes on a successful loop when no edges are modified) or loops until the algorithm detects that the plan is not dynamically controllable (either in Line 5 or Line 16).

```
function DC1(Γ)
input A STNU Γ
effects computes a dynamically controllable CDGU if the plan is dynamically controllable.
returns true if Γ dynamically controllable, otherwise false
1  G ← DISTANCE_GRAPH(Γ)
2  do
3     G ← COMPUTE_APSP_GRAPH(G)
4     if ¬IS_PSEUDO_CONTROLLABLE (G)
5        return FALSE
6     modified ← FALSE
7     for each contingent timepoint B ∈ N(G) associated with uncontrollable duration AB
8        for each incoming positive edge CB
9           modified ← apply tightenings required by the precede reduction to triangle ABC.
10          modified ← apply conditional constraints required by conditional unordered reduction
                       to triangle ABC
11          if modified break
12       end for
13       if modified break
14    end for
15    if ¬REGRESS_WAITS(G)
16       return FALSE
17    modified ← IS_MODIFIED(G) or modified
16    end if
17 while modified=TRUE
18 return TRUE
```

Figure 4-14 Pseudo-Code for Dynamic Controllability (DC) algorithm [Morris 2001]

The DC algorithm is sound because it only derives new constraints based on the original constraints and the assumption of dynamic controllability [Morris 2001]. Furthermore, the completeness of this algorithm was shown in [Morris 2001].

The time complexity of the DC algorithm is shown to be polynomial [Morris 2001]. The individual tightenings are clearly polynomial, and convergence is assured because the domains of the constraints are strictly reduced by the tightenings. However, only a crude estimate was provided for how long the convergence would take. Moreover, a crude estimate is in terms of the maximum value of the edges and the fixed precision on the edges. Each time the algorithm loops through Lines 2-17 it applies at least one tightening. If all the distance on the edges are bounded by $\pm$ B, and there is a fixed level of precision δ, and E edges. Then, after at most BE/δ loops, the algorithm will converge. Each loop requires an $O(N^3)$ APSP computation and there are N^3 edges in the APSP graph; Therefore, the crude bound becomes N^6*B/δ !

The DC algorithm depends on repeated calls to an expensive $O(N^3)$ APSP computation in Line 2 to perform requirement edge constraint propagation. Furthermore, it uses an inefficient looping scheme that first resolves the requirement edges with one another via the APSP algorithm, then resolves the requirement edge with the contingent edge propagation via reductions, and, finally, resolves the conditional edges with the requirement, and conditional edges with contingent edges via regression. In the next section, we show how to improve on the performance of the dynamic controllability algorithm by resolving all possible combinations of constraints all at once. This general frame work enables our new fast dynamic controllability algorithm to remove the repeated APSP computations.

4.4 Fast Dynamic Controllability Algorithm

In this section, we describe our novel fast dynamic controllability algorithm (fast-DC algorithm). This fast-DC algorithm has a significant performance improvement compared to the dynamic controllability algorithm introduced by [Morris 2001]. This fast dynamic controllability algorithm achieves its enhanced speed via several new improvements. The speed of the fast-DC algorithm is verified empirically.

1. We show how to exploit the fact that a dispatchable plan can by incrementally executed during the reformulation phase. We introduce a local incremental algorithm for maintaining the dispatchability of a plan constrained by STNs. In this algorithm, when an edge length changes in a dispatchable distance graph, only a subset of the constraints need to be notified of this change. Specifically, the change only needs to be *back-propagated*, similar to regression. Then we show how to apply this technique to plans constrained by STNUs. In order to make this transition from STNs to STNUs, we introduce a new property, called *pseudo-dispatchability*, and show that for any for pseudo-dispatchable STNU only changes in requirement edges need to be back-propagated. This removes the need to compute the APSP-graph, which updates all edges in the distance graph, every time a requirement edge changes.

2. The constraint propagations of requirement edges, contingent edges and conditional edges required by dynamic controllability are combined into an efficient general framework. This general framework enables the different types of constraint propagation to be interleaved with one another rather than applying them sequentially. Interleaving the different types of propagation enables the dynamic controllability algorithm to reduce the number of propagations required. The idea is to apply the required tightening as soon as we can deduce them, so that the next round of propagations has the most up-to-date constraint set as possible.

3. We trim the distance graph of redundant constraints prior to performing the integrated constraint propagation. This can drastically reduce the number of propagations required.

First we introduce the incremental algorithm for maintaining the dispatchability of STNs. Next we show how this incremental algorithm applies to STNUs by introducing the idea of pseudo-dispatchability. This provides the basis for the new requirement edge propagation technique, which removes the dependence of the dynamic controllability algorithm on repeated APSP calls. Next, we describe the set of back-propagation rules that make up the general constraint propagation framework and present the back-propagation algorithm. Finally, we describe the new fast-DC algorithm pseudo-code, which uses this new back-propagation algorithm. After presenting the algorithm, we demonstrate the fast-DC algorithm on several examples and review how the fast-DC algorithm fits in with the Hierarchical Reformulation algorithm, presented in Chapter 3.

4.4.1 Incremental Dispatchability Maintenance

In order to understand how the new requirement constraint propagation technique works, let's revisit the problem of dynamically executing a STN. [Muscettola 1998a] showed that any dispatchable STN can be executed incrementally using a dispatching algorithm. If a STN is dispatchable, as long as each execution decision is consistent with the past assignments, then we can guarantee that there is a consistent assignment for future timepoint assignments. Recall that executing a timepoint is equivalent to adding a set of rigid constraints between the start of the plan and the timepoint being executed. During execution, the dispatcher ensures that the addition of these additional constraints is consistent with the past, by propagating information at execution time. However, if a random constraint is modified in a dispatchable graph, we need to make sure that the change is consistent with the past using *back-propagation*. Back-propagation informs all constraints that relate timepoints in the past. If the back-propagation does not introduce an inconsistency, then the constraint change is consistent with all the constraints. This leads to an efficient algorithm for incrementally updating a dispatchable STN distance graph.

In order to develop a back-propagation algorithm for a dispatchable STN, we use a logic similar to that used when developing the reduction and regression rules. Specifically, any positive edge AB that is either added or modified is only threatened by

outgoing negative edges from B. In addition, any negative edge BA that is either added or modified is only threatened by incoming positive edge to A. Therefore, we need to back-propagate any change in a positive edge AB through all negative edges originating from B. Similarly, we need to back-propagate any change in a negative edge BA through all incoming positive edge into B.

These back-propagations need to be applied recursively in order to ensure that the change is consistent with the past. This back-propagation technique only requires us to update a subset of the edges (i.e. constraint that may happen in the past), instead of all the edges which would happen if we were to recompute the APSP every time an edge changed. The future constraint will be notified of the change when they need to be notified, which is at execution time. Thus we defer the future updates until execution time. Furthermore, the back-propagation preserves the dispatchability of the distance graph.

First, we give an example, then we provide the formal back-propagation rules for distance graphs associated with STNs. Finally, we show how this back-propagation is extended to distance graphs associated with STNUs.

Example 4-10

Consider the dispatchable distance graph shown in Figure 4-15. Figure 4-15(a) shows the original dispatchable graph. Consider a scenario in which the edge DC is reduced from -2 to -5 for some reason, as shown in Figure 4-15(b). During execution, the edge DC is used to propagate a lower bound to timepoint D. We call timepoint D the timepoint of interest. In order to maintain the dispatchability of the graph, the tightening of DC needs to be propagated through the graph. However, the effects only need to be propagated backward from the node of interest, because, as long as D is consistently executed, the dispatcher is able to consistently execute E.

The negative edge DC is threatened by the incoming positive edge CD and BD. We resolve the new edge DC with the threats (CD and BD), by computing the local shortest path through the threats. The shortest path BDC results in a tightening of edge BD from 8 to 5, and the shortest path CDC results in a new edge CC of 5. Figure 4-15(d) shows the result of the first step of back-propagation. The tightening of the constraint BC is then back-propagated where node C is the timepoint of interest. The edge positive BC is threatened by all outgoing negative edges from C (CB and CA), as shown in Figure 4-15(e). BC is back-propagated through its threats. The shortest path BCB results in a new edge BB of 0, and the shortest path BCA results in a tightening of BA from 0 to -1. The results of the second stage of back-propagation are shown in Figure 4-15(f). Note that BA needs to be back-propagated through AB, resulting in a new edge AA of 9.

Back-propagation does not introduce any negative cycles; therefore, the change is consistent. Furthermore, the tightenings introduced through the back-propagation preserve the dispatchability of the distance graph.

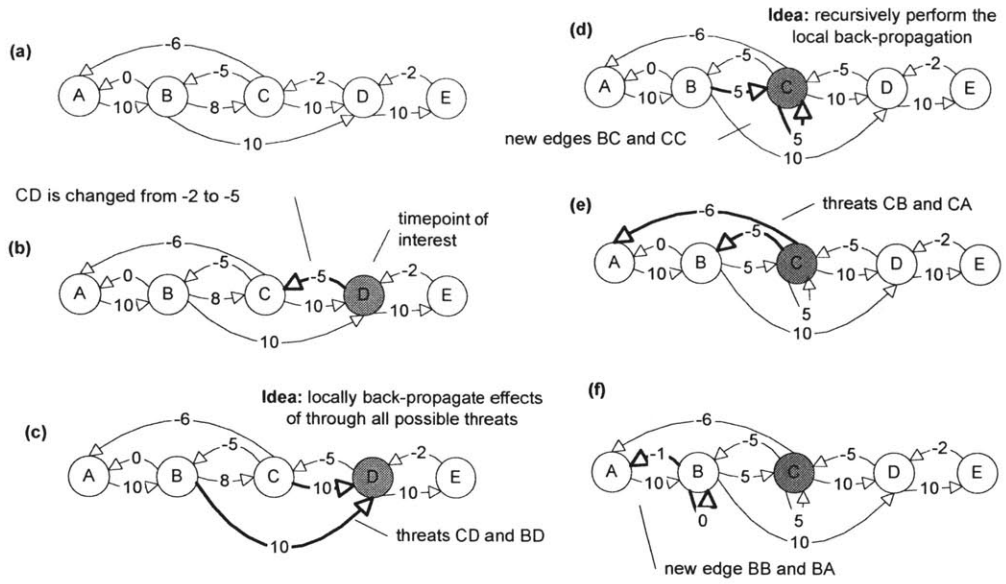


Figure 4-15 Back-Propagation Example

Now we give the back-propagation lemma and incremental algorithm used to maintain the dispatchability of a STN distance graph.

Lemma (Incremental Dispatchability) *Given a STN and associated dispatchable distance graph G,*
i) any change or addition of an edge AB of distance x, where x >0, for all edges BC of length y, where y <= 0, we can deduce a new constraint AC of length x+y.
ii) any change or addition of an edge BA of distance z, where p <= 0, for all edge CB of length q, where q >= 0, we can deduce a new constraint CA of length p+q.
Furthermore, recursively applying rules i and ii maintains the dispatchability of the G.

The algorithm for maintaining the dispatchability of the distance graph, recursively applies the Incremental Dispatchability propagation rules until no more back-propagations can be deduced. If back-propagation introduces a negative cycle then the algorithm returns false, otherwise, the algorithm returns true

The Incremental Dispatchability (ID) algorithm used to replace the APSP computation in the slow dynamic controllability algorithm introduced by [Morris 2001]. In order to apply the ID algorithm to distance graphs with uncertainty (DGUs) we introduce the idea of *pseudo-dispatchability.* If we ignore the distinction between contingent and requirement edges in the DGU (as we did when we computed pseudo-controllability), then the DGU is effectively converted into STN distance graph. If this associated STN distance graph is dispatchable, then we say the DGU *pseudo-dispatchable.* In order to maintain the pseudo-controllability of DGU when a constraint is changed, we apply the ID algorithm to the DGU. This resolves a change in a requirement constraint with all the other requirement constraints.

We also introduce the term *pseudo-minimal dispatchable graph (PMDG).* A PMDG is DGU in which the associated STN distance graph contains the fewest number of edges. The edges of the DGU are trimmed using the same dominance relationships introduced by [Muscettola 1998a].

4.4.2 Back-Propagation

In this subsection we describe a set of local constraint propagation rules that determine how one constraint change affects the values of other constraints, in order to maintain the dispatchability of a dynamic controllability Conditional Distance Graph with Uncertainty (CDGU). These rules all share one important property - they only affect constraints that occur earlier in the plan; thus, we call them *back-propagation* rules for STNUs. This idea is illustrated in **Figure 4-16**. These rules and the associated back-propagation algorithm form the basis of the Fast-DC algorithm. The back-propagation rules integrate the Incremental Dispatchability rules, the reduction rules and the regression rules. The back-propagation rules put all these rules in to common framework.

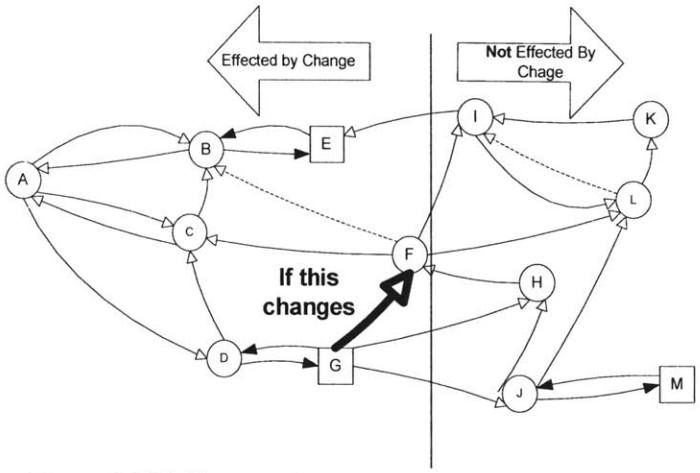


Figure 4-16 If either a requirement, or conditional edge changes, in order to maintain the dispatchability of the CDGU, the effects only need to be back-propagated

Each back-propagation rule differs, depending on the types of edges involved, the signs of the edge distances, and the relative direction of the edges. In a DGU, there exist five types of edges: positive and negative requirement edges, positive and negative contingent edges, and negative conditional edges.

Requirement	Contingent	Conditional
+ -	+ -	-

We group our back-propagation rules into three groups: negative requirement edges, positive requirement edges, and negative conditional edges because these are the only three types of edges that may be added or modified during reformulation. Any positive conditional edge is converted to a requirement edge by the unconditional unordered reduction rule. The rules are used to determine what new constraints need to be enforced to ensure consistency and dynamic controllability. The following table summarizes the back-propagation rules used in the Fast-DC algorithm.

If This Changes	**Must Back-Propagated Through**	**Derived From**
Negative requirement edge BA	1. any positive requirement edge CB	ID(i)
	2. any positive contingent edge CB	Precede Reduction
Positive requirement edge AB	1. any negative requirement edge BC	ID(ii)
	2. * any negative contingent edge BC	CUR
	3. any negative conditional edge BC of <-t,D> where D ≠A	Regression(i)
Negative conditional edge BA	1. any positive requirement edge CB of <-t,D> where D ≠A	Regression(i) Regression(ii)
	2. any positive contingent edge CB	

* apply conditional constraint in both precede or unordered cases
ID: Incremental Dispatchability
CUR: conditional unordered reduction

Table 1 Back-Propagation Rules Summary

4.4.3 Back-Propagating when a Negative Requirement Edge Changes

Recall that when a dispatcher executes a timepoint it propagates that execution times through the distance graph in order to update the execution windows of the neighboring nodes. The dispatcher uses the negative edges to update the lower bound of the timepoint's execution window. The only way a timepoint's lower bound, derived from a negative edge propagation, can be violated is if some other positive edge propagates an upper bound that is smaller than this lower bound. The back propagation rules are used to prevent this inconsistent condition from happening.

Recall that there are two types of positive edges in the DGU: a positive requirement edge (Case1) and a positive contingent edge (Case2). The back-propagation rule for changing negative requirement edges handles both cases. In [Morris 2001], the first case was handled by the APSP computation, and the second case was handled by the precede reduction.

Case 1: Back-propagating a negative requirement edge through a positive requirement edge.

This back propagation rule is called in case when there exists some change in or creation of a negative requirement edge BA with weight *a*, such that there exist some positive requirement edges CB with weight *b*. The back-propagation rule derives a new constraint with CA with weight a+b. If this new edge CA provides a tighter constraint then it update the DGU accordingly. Note that the arbitrary timepoint C may be the timepoint B, in which case the derived constraint loops on the timepoint B. This example is depicted in **Figure 4-17** (Case1).

Case 2: Back-propagating a negative requirement through a contingent link.
This is exactly the same case as the precede case derived in the dynamic controllability algorithm [Morris 2001]. This propagation is illustrated in **Figure 4-17** (Case2). The correctness of this propagation rule is shown in [Morris 2001], for the case where there exists some negative requirement edge.

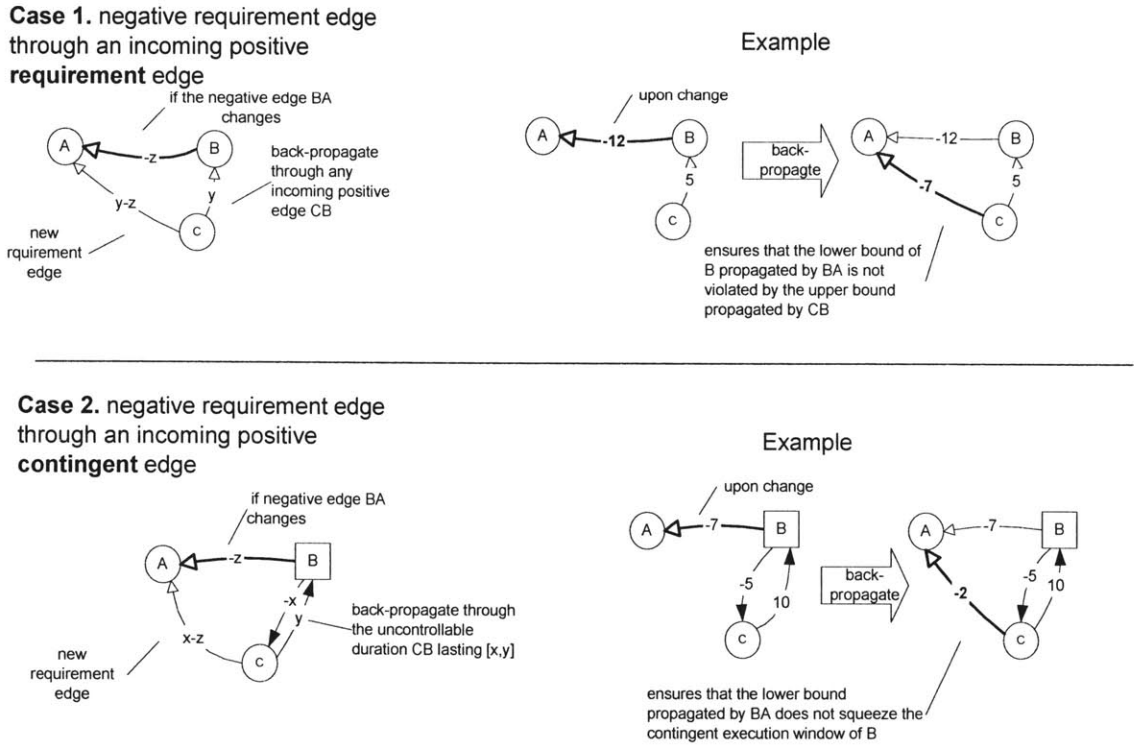


Figure 4-17 Back-Propagation Rules for Negative Requirement Edge

4.4.4 Back-Propagation Rule when Positive Requirement Edge Changes

If a positive requirement edge changes, there are three cases to consider. All three cases are illustrated in **Figure 4-18**. The rationale for each rule is shown in Table 1.

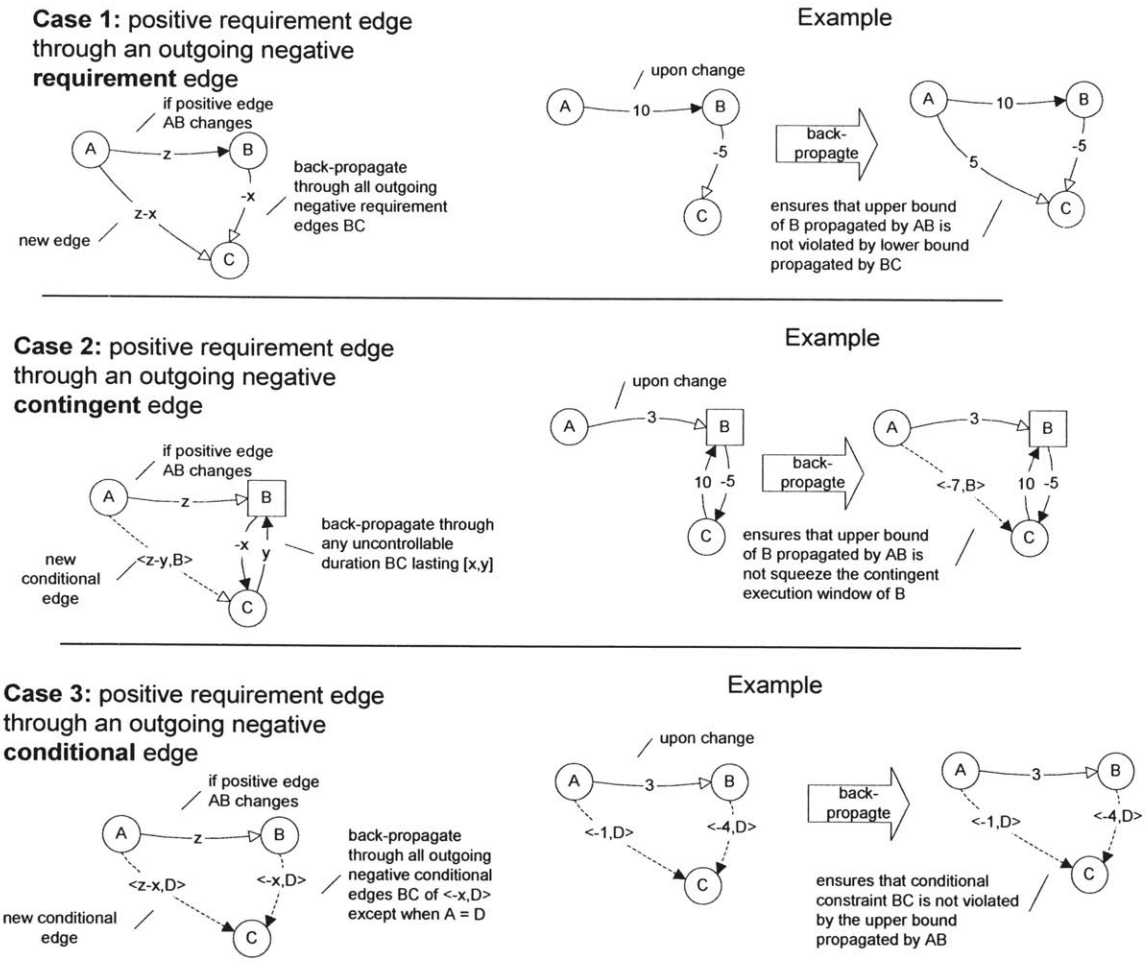


Figure 4-18 Back-Propagation Rule for Positive Requirement Edges

4.4.5 Back Propagating Conditional Edges

The back-propagation rule for the addition or change of a conditional constraint is exactly the same as the regression rules. The back propagation rules are shown here for completeness. They are illustrated in Figure 4-19.

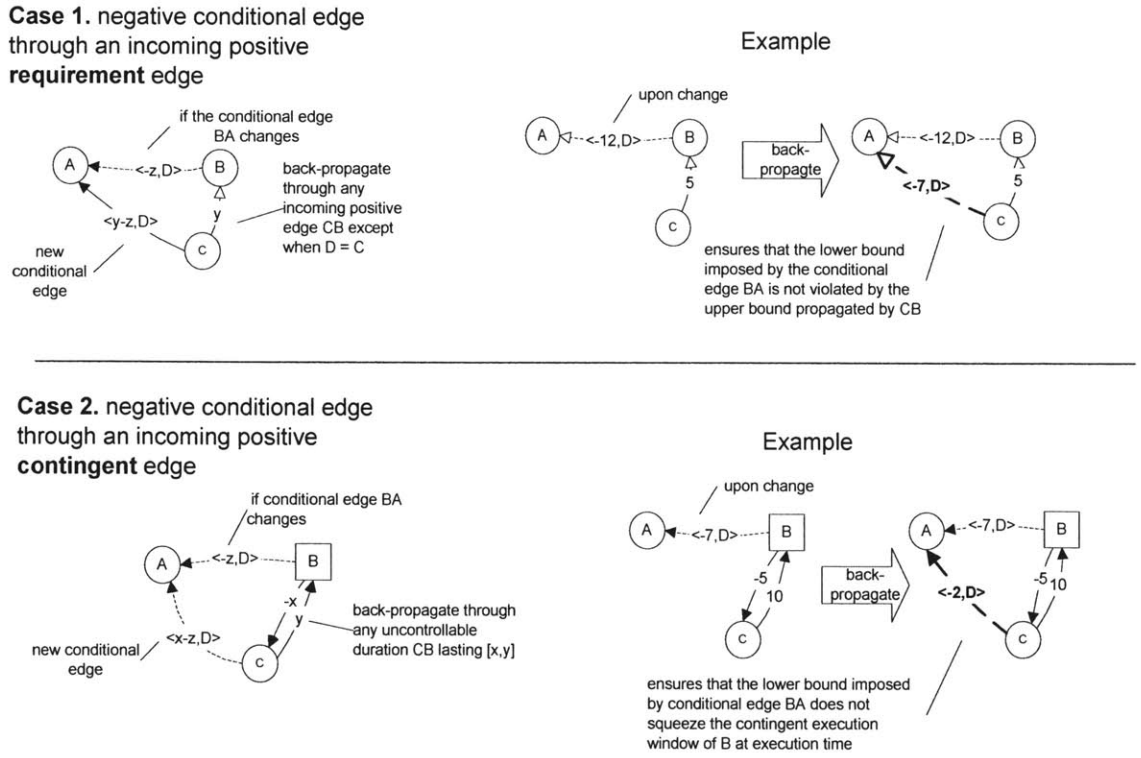


Figure 4-19 Back-Propagation Rules for Conditional Edges

5.1.1 Pseudo-Code for BACK-PROPAGATE

The pseudo-code for BACK-PROPAGTE is show in Figure 4-20. BACK-PROPAGATE is a function that recursively applies the back propagation rules in the previous sections. It accepts a CDGU G, a start timepoint u, and an end timepoint v. BACK-PROPAGATE is initiated in order to prevent an positive requirement edge (u,v) from squeezing the upper bound of the contingent timepoint *v* or in order to prevent a negative requirement edge (u,v) from squeezing the lower bound of a contingent timepoint *u*. The algorithm returns true if the edge (u,v) is successfully back-propagated through G. (i.e. no inconsistencies were introduced) otherwise the algorithm returns false.

Lines 1-2 detect two possible termination conditions. If the timepoint u = v, the edge(u,v) is a loop. If this loop is positive, thus, does not introduce a temporal inconsistency, then the algorithm returns true in Line 1. However, if the loop is negative then the algorithm returns false in Line 2

Lines 3-15 applies the all applicable back-propagations associated with edge (u,v). Specifically the algorithm back-propagates (u,v) through all appropriate edge (x,y) resulting in a new edge (p,q). In line 5 it applies the unconditional unordered reduction when appropriate, which converts a conditional edge (p,q) into a requirement edge (p,q). This new edge (p,q) (conditional or requirement) is resolved with G. If G is modified it does two things. It checks if the new edge (p,q) introduces any local negative cycles. Specifically, it checks if the cycle p-q-p is negative. If there is a local negative cycle, then the algorithm returns false, otherwise the algorithm recursively calls BACK-PROPAGATE(G, p, q). If this BACK-PROPAGATE returns false, then the orginal BACK-PROPAGATE function returns false. If the algorithm successfully applies all possible back-propagations of (u,v) in line 3-13, then the algorithm returns true in Line 16.

In the next section we give a example of the BACK_PROPAGATE function in the context of the Fast Dynamic Controllability algorithm.

```
Method BACK-PROPAGATE(G,u,v)
Input: CDGU G, start timepoint u, and end timepoint v
Effects: recursively called function that back-propagates the constraints through G
Returns: true if the no inconsistencies where introduces, otherwise false
1     if IS-POS-LOOP( u, v ) return TRUE
2     if IS-NEG-LOOP( u, v ) return FALSE
3     for each edge (x,y) where the back-propagation rules apply to edge (u,v)
4        back-propagate (u,v) through (x,y) to create new edge (p,q)
5        convert any conditional constraint (p,q) to a requirement edge (p,q) as required
         by the unconditional unordered reduction
6        resolve the edge (p,q) with G by tightening (or adding) corresponding edge (p,q) in G
7        if G is modified
8           if resolving (p,q) with G introduces a local negative cycle
9              return FALSE
10          end if
11          if ¬BACK-PROPAGATE(G,p,q) // recursive call
12             return FALSE
13          end if
14       end if
15    end for
16    return TRUE
```

Figure 4-20 Pseudo-Code for BACK-PROPAGATE

4.5 Fast Dynamic Controllability Pseudo-Code

The pseudo-code for the Fast-DC algorithm is shown **Figure 4-22**. The algorithm is used to reformulate the group plans in the Hierarchical Reformulation Algorithm. The Fast-DC algorithm operates on group plan's associated STNU. If the STNU associated with the group plan is dynamically controllable, then the algorithm returns a pseudo-minimal dispatchable CDGU, which can be directly executed by the STNU dispatching algorithm introduced by [Morris 2001], otherwise, the algorithm returns NIL. The description of the Fast-DC pseudo-code is interleaved with a example. The TPNU used in the example is shown in Figure 4-21. This group plan is part of the Mars rover example originally introduced in Section 3.4.

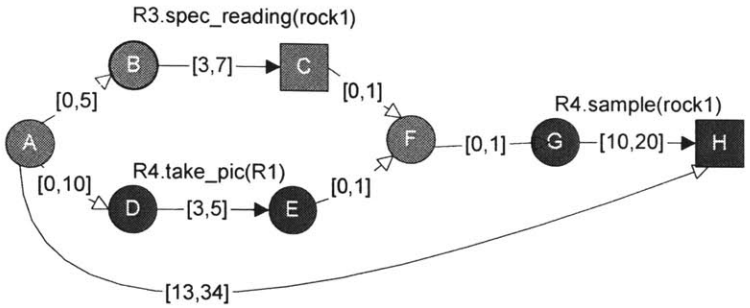


Figure 4-21 Sample Group Plan

---

function FAST-DC(Γ)
input: A Simple Temporal Network with Uncertainty Γ
returns minimal dispatchable CDGU if Γ is dynamically controllable, otherwise NIL
1 $G \leftarrow$ STNU_TO_CDGU(Γ)
2 **if** ¬COMPUTE_MPDG(G)
3 **return** NIL
4 **end if**
5 **if** ¬ IS_PSEUDO_CONTROLLABLE (G)
6 **return** NIL
7 **end if**
8 S ← start timepoint of G
9 Bellman_Ford_SDSP(S, G)
10 Q ← ordered list of contingent timepoints according to the SSSP distances
11 **while**(¬Q.IS-EMPTY())
12 $n \leftarrow$ Q.POP_FRONT()
13 **if** ¬ BACK_PROPAGATE_INIT(G, n)
14 **return** NIL
15 **end if**
16 **end while**
17 COMPUTE_MPDG(G)
18 **return** G

Figure 4-22 Pseudo-Code of Fast Dynamic Controllability Algorithm (Fast-DC)

Line 1 converts the STNU into a CDGU. This conversion is trivial. It converts the links of the STNU into a pair of directed edges. Note that initially, the CDGU does not contain any conditional constraints; therefore, the original CDGU is similar to a Distance Graph with Uncertainty (DGU). For example, **Figure 4-23** shows the CDGU of the sample group plan.

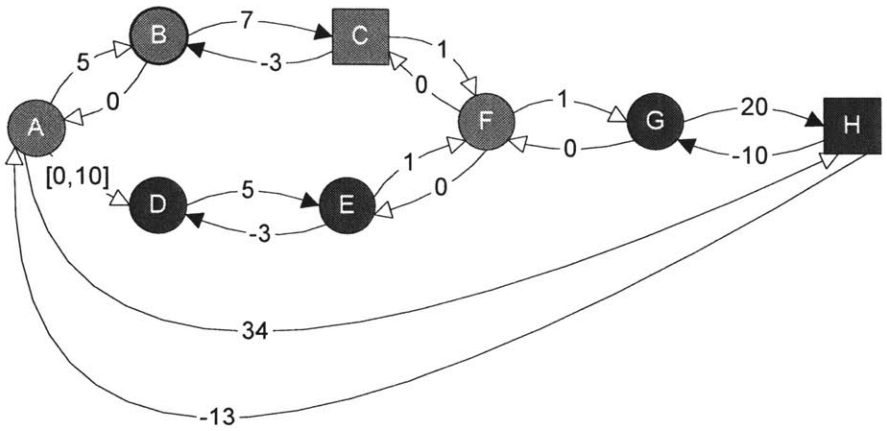


Figure 4-23 CDGU of the Sample Group Plan

Recall that in order to apply the back-propagation rules, the CDGU must be pseudo-dispatchable. In addition, in order to efficiently apply the back-propagation rules, the CDGU should contain the fewest number of edges. Line 2 transforms the CDGU into a Minimal Pseudo-Dispatchable Graph (MPDG) by calling the COMPUTE_MPDG function. This function applies the basic STN Reformulation Algorithm [Muscettola 1998a] on the CDGU. The STN Reformulation algorithm is applied by ignoring the distinction between contingent and requirement edges in the CDGU. This function reformulates the constraints of the CDGU. If the CDGU is pseudo-dispatchable, then the function COMPUTE_MPDG returns true, otherwise it return false. If the CDGU is not pseudo-controllable, then the FAST-DC algorithm returns NIL. The minimal pseudo-dispatchable graph for the sample group plan is shown in **Figure 4-23**.

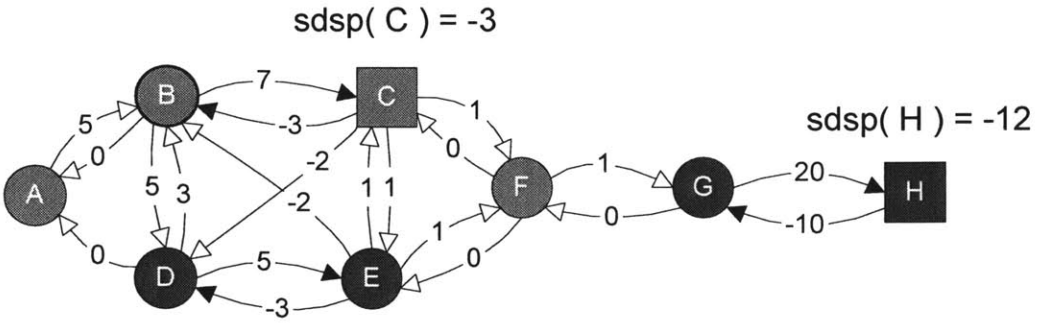


Figure 4-24 MPDG of the Sample group plan

The CDGU is only dynamically controllable if it pseudo-controllable. Recall that if a graph is pseudo-controllable then the constraints do not strictly imply a tighter constraints on the contingent edges. Lines 5-7 of the FAST-DC algorithm checks if the contingent edges are squeezed during the process of converting the CDGU into a minimal pseudo-dispatchable graph. If the CDGU is not pseudo-controllable, then the FAST-DC algorithm returns NIL. In our example, contingent edges BC, CB, GH, and HG all remain unchanged; therefore, the CDGU is pseudo-controllable.

Recall that our goal is to reformulate the graph to ensure that the plan can be dynamically executed. This reformulation is done by applying the a recursive BACK_PROPAGATE function. The BACK-PROPAGATE function needs to be applied to any edge that may squeeze the contingent timepoint at execution time. Each initial call of BACK_PROPAGATE causes a series of other edge updates. However, they will only update edges closer to the start of the plan. In order to reduce the amount of redundant work, we initiate the back-propagation cycle near the end of the plan to the start of the plan. This way we slowly build up a solution from the end of the plan to the start of the plan. In order to organize the back-propagations, we need to create a list of contingent timepoints ordered from timepoint that are executed near the end of the plan to the timepoints that are executed near the beginning of the plan.

Lines 8-10 create this ordered list, Q, of the contingent timepoints. The contingent timepoints are ordered based on their Single-Destination Shortest-Path (SDSP) distance, sdsp(x). Specifically, the contingent timepoints are ordered from lowest to highest SDSP distances. Note that all SDSP distances are less than or equal to zero. The SDSP distances are computed in Line 9, and the contingent timepoints are ordered in Q in Line 10. The sample group plan contains two contingent timepoints C and H with sdsp(C) = -3 and sdsp(H) = -12. Therefore, timepoint H comes before timepoint C.

Lines 11-16 of the FAST-DC algorithm apply the back-propagation rules. Line 12 pops the first contingent timepoint, n, off of the list Q and calls the BACK-PROPAGATE_INIT function, which starts one round of back-propagations. If that back-propagation round results in a inconsistency, then the FAST-DC algorithm returns NIL. The pseudo-code for the BACK-PROPAGATE_INIT is shown in Figure 4-25. BACK-PROPAGATE_INIT ensure the that contingent timepoint n is never squeezed during execution.

```
BACK-PROPAGATE-INIT( G, v )
Input: A CDGU G and contingent timepoint v
Returns: true if no inconsistencies were introduced during the back-
propagation cycle, otherwise false
1  for all incoming positive edges (u,v ) into the contingent timepoint v
2     if ¬BACK_PROPAGATE(G,u,v)
3        return FALSE
4     end if
5  end for
6  for all outgoing negative edges (v,u) from the contingent timepoint v
7     if ¬BACK_PROPAGATE(G,v,u)
8        return FALSE
9     end if
10 end for
11 return TRUE
```

Figure 4-25 Pseudo-Code for BACK-PROPAGATE-INIT

The BACK-PROPAGATE-INIT function initiates the back-propagation by applying the back-propagation rules to ensure that the contingent timepoint v is never squeezed during execution. Recall the contingent timepoint can only be squeezed by incoming positive edges or outgoing negative edges. Lines 1-5 calls BACK_PROPAGATE for all incoming positive edges into the contingent timepoint v and Lines 6-10 calls BACK_PROPAGATE for all outgoing negative edges from v.

For example, consider the series of back-propagations shown in Figure XXX.
There does not contain any possible edges to violate contingent timepoint H so no back-propagation is required. For timepoint C the edge EC is back-propagated through BC resulting in a new conditional edge EB of <C,-6>. This edge then back-propagated DE which modifies the requirement edge DE to -1. This requirement edge is then back-propagated through edge BD resulting in the edge BB of distance 4. This thread of back-propagation terminates here because of a positive self-loop.

The contingent timepoint C is also threatened by the outgoing negative edge CD of length -2. This edge CD is back-propagated through BC which modifies BD = 1. This is then back-propagated through DB resulting modifying the self looping edge BB to 0. No more propagations are necessary. The resulting dispatchable CDGU is shown in Figure 4-26. The back-propagation did not introduce an inconsistency; therefore, the sample group plan is dynamically controllable.

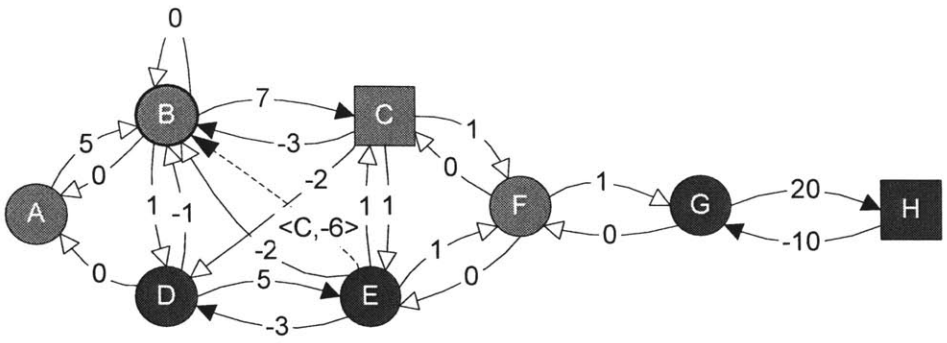


Figure 4-26 Dispatchable CDGU after back-propagation

The last step of the Fast-DC algorithm is to trim the dominated (redundant) edges from the CDGU. This is done by calling the basic STN reformulation algorithm. The resulting graph is a minimal dispatchable CDGU which can be executed by the dispatching algorithm introduced by [Morris 2001]. For example, the minimal dispatchable CDGU for the sample group plan is shown Figure 4-27.

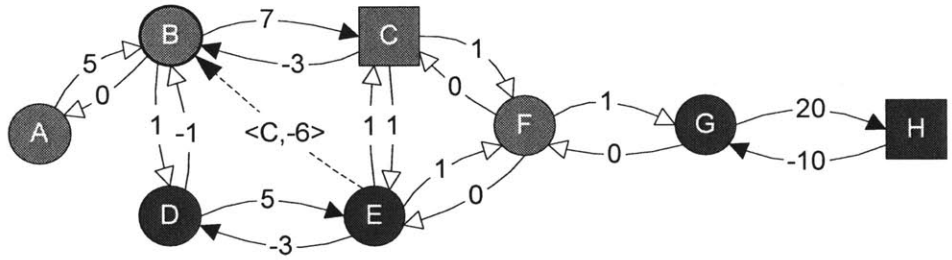


Figure 4-27 CDGU of sample group plan after trimming the redundant edge

4.6 Summary

In this chapter we reviewed the dynamic controllability algorithm introduce by [Morris 2001]. Then we generalized the reduction rules introduced to [Morris 2001] in order to develop an efficient dynamic controllability algorithm. This new Fast-DC algorithm is used by the HR algorithm presented in Chapter 3 to reformulate the group plans. In the next chapter we present empirical data demonstrating the speed of this new Fast-DC algorithm.

5 Results and Conclusion

5.1 Introduction

The outline for this chapter is as follows. First we discuss the implementation of the Hierarchical Reformulation (HR) algorithm. Then we discuss the experimental results of the fast dynamic controllability (Fast-DC) algorithm. We then discuss the limitations of our approach and directions for future work. We conclude with a summary of the major contributions of this thesis.

5.2 Implementation of the Hierarchical Reformulation algorithm

The Hierarchical Reformulation (HR) algorithm has been implemented in C++ and tested with a variety of hand coded examples, including the cooperative Mars rover scenario presented in Chapter 3.4. In order to generate the two-layer multi-agent plans, we developed a MTPNU Graphical User Interface (GUI) implemented in Java.[6] The GUI enables the user to create and visualize the MTPNUs. The screenshot of the editor, shown in **Figure 5-1**, shows the mission plan for the Mars rover scenario described in Chapter 3.4. The editor allows the user to create, modify, and visualize the multi-agent plans in a variety of different forms. All of the plans generated for the HR algorithm were created using the MTPNU GUI.

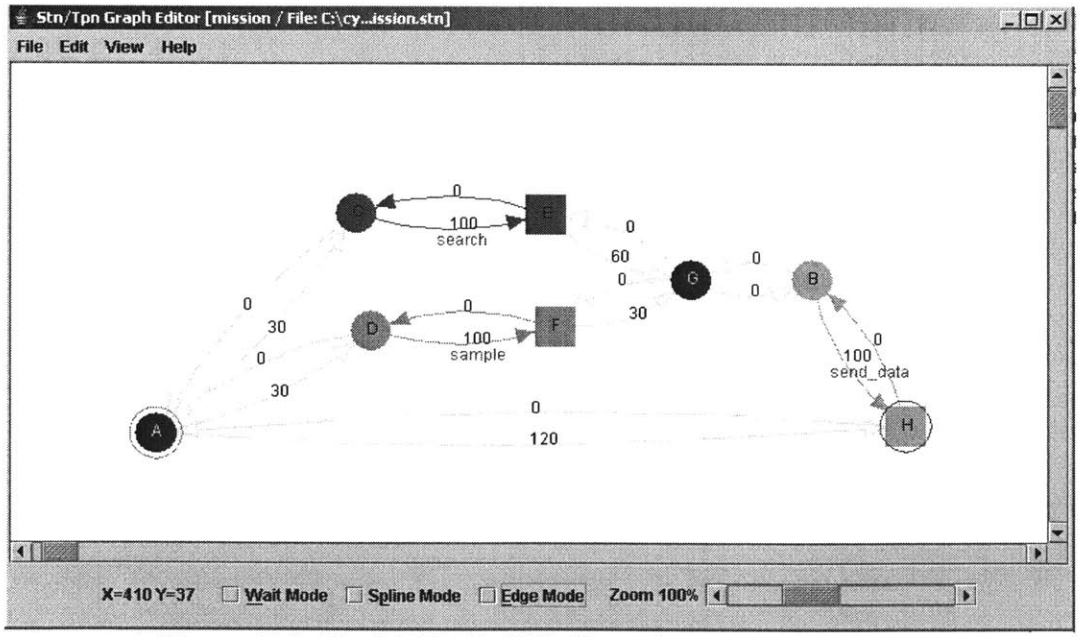


Figure 5-1 The MTPNU GUI allows the user to quickly create, visualize and manipulate multi-agent temporal plan with uncertainty.

[6] The MTPNU GUI was developed by Andreas Wehowsky and myself.

Figure 5-2 Rover Test-Bed used to test STN reformulation algorithms

The Hierarchical Reformulation algorithm was implemented in conjunction with the STN reformulation and dispatching algorithms presented in Chapter 2. Implementing the STN reformulation provided the basis for implementing the HR algorithm. Specifically, the implementation of the basic STN reformulation algorithm [Muscettola 1998a] and the "fast" STN reformulation algorithm [Tsarmardinos 1998], along with the associated STN dispatching algorithm [Muscettola 1998a] were implemented in C++ prior to implementing the HR algorithm. The HR algorithm implementation leveraged the data structures and many of the graph algorithms used in these STN algorithms. The STN reformulation and dispatching algorithms were integrated with the KIRK temporal planner [Kim 2001]. This entire planning and execution system has been tested on a set of ATRV and ATRV Jr. robots in our rover test bed, as shown in **Figure 5-2**. Together the KIRK temporal planner and STN executive have been successfully used to execute hundreds of multi-rover plans in the rover test-bed.

Recall, that our novel fast dynamic controllability algorithm uses the STN reformulation algorithm as both a pre-processing and post-processing step. Currently, only the basic STN reformulation algorithm has been integrated with our implementation of the Fast-DC algorithm. Integrating the fast STN reformulation algorithm into the Fast-DC is left for future work.

5.3 Run Time Complexity of the FAST-DC Algorithm

In this section we discuss the empirical results of the fast dynamic controllability (Fast-DC) algorithm. In order to test the speed of Fast-DC algorithm, we developed a random TPNU generator. The challenge is to generate a TPNU and associated STNU that is sufficiently random, and provides a good chance of being dynamically controllable. We are interested in knowing how long it takes the algorithm to detect that the STNU is not dynamically controllable; however, we are more interested in knowing how long it takes for the algorithm to complete.

Simply randomly generating a mixture of contingent and requirement edges would yield a very low likelihood of producing a dynamically controllable STNU. We introduce RAND_TPNU algorithm in order to generate the TPNU and underlying STNU that are biased towards being dynamically controllable. This is achieved by keeping the requirement constraint flexible compared to the duration of the uncontrollable activities.

The RAND_TPNU function accepts two parameters: *num_acts*, which is the number of activities in the TPNU, and *ctrl_pct*, which determines the number of controllable activities versus uncontrollable activities. Given these parameters it generates a TPNU.

First the RAND_TPNU algorithm generates *num_acts* activities each of which have a *ctrl_pct* chance of being controllable. The duration for each activity is chosen such that upper bound, *ub*, is always greater than the lower bound, *lb*. Thus, each activity has a nonzero duration and is locally consistent.. The upper bound, ub, is randomly chosen between [1, *max_duration*] and the lower bound, lb, is randomly chosen between [0,ub]. Each activity has a start timepoint, S_i and end timepoint E_i and either a contingent or requirement link connecting the S_i to E_i.

The RAND_TPNU uses a 2D plan space with dimensions *plan_length* by *plan_height*, as shown in Figure 5-1. The left hand side occurs and Time = 0 and the right hand side occurs at Time = plan_length. The plan space is similar to a simple scheduling timeline where overlapping activities represent concurrent activities. The start timepoint, S_i, of each activity is randomly placed in the 2D plan space and the end timepoint, E_i , of each activity is shifted to the right by a distance equal to its upper bound. For example, in Figure 5-1 the plan space has dimensions 50 by 25. The start timepoint of act1, s1, is randomly placed at (5,20). Act1 has an upper bound of 10; therefore, the end timepoint of act1, e1, is placed at (15,20). By controlling the length and height of the plan space, we can control the relative proximity of each activity in the plan space.

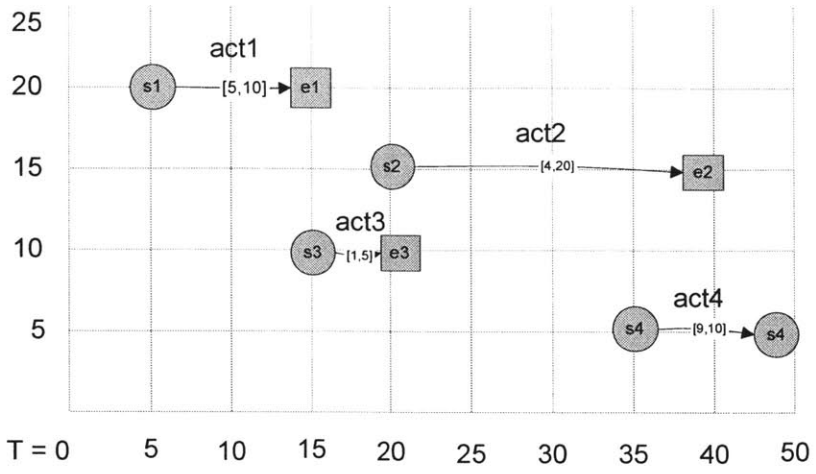


Figure 5-3 **Randomly placing activities within the 2D plan space**

After placing each activity in the plan space, requirement links are introduced in order to constrain the activity timepoints with respect to one another. For each timepoint, the algorithm searches in its local "search region" for a neighboring timepoint for which there does not already exist any link (requirement or contingent). If it finds such a timepoint, it orders the timepoints based on their x distance. Whichever timepoint is further to the left in the plan space, becomes the start of the requirement link and the other timepoint becomes the end timepoint of the requirement link. This ensures that the plan execution goes generally from left to right in the plan space.

The algorithm then randomly generates locally consistent values for the lower and upper bound of the requirement link. The algorithm uses the separation distance to generate these values such that timepoints near one another tend to be more tightly constrained than timepoints farther apart. This results in a good mixture of tight and loose temporal requirement; furthermore, the general structure of the plan can be determined by visual inspection of the plan. However, in order to prevent extremely tight constraints being placed in the plan, which tends to reduce the change of the plan being dynamically controllable, the algorithm uses a different policy for generating the timebounds when the two timepoints are close to one another. The process of adding a requirement link is shown in **Figure 5-4** The specific policy for generating the requirement links is given in Figure 5-6. Note that the algorithm generates links with both positive and negative lower bounds.

The pseudo-code for the full RAND_TPNU generator is given in Figure 5-6. It uses the function RANDOM(x,y) which produces a random integer between x and y.

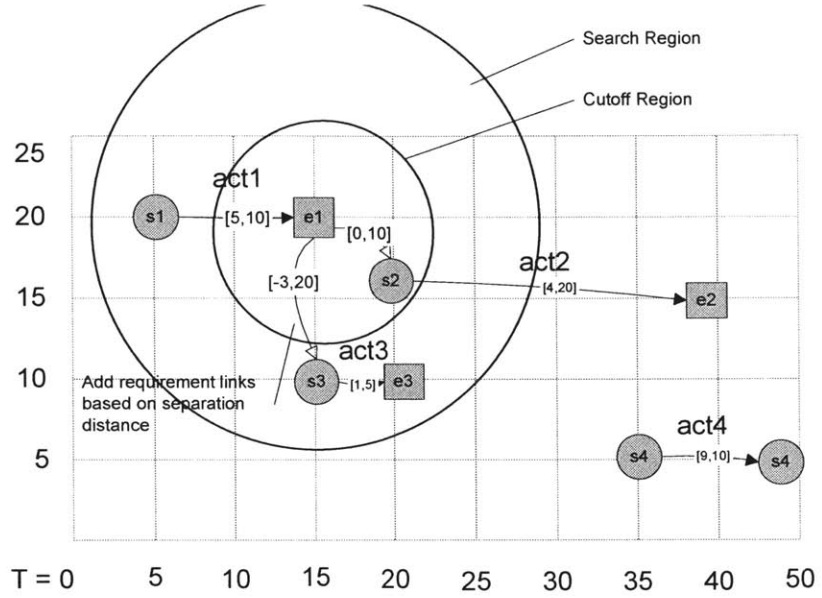


Figure 5-4 Place requirement edges between neighboring timepoints

```
    ADD_REQUIREMENT_LINKS(G)
    Input: A TPNU G with a set of contingent and requirement activities
    Effects: adds a set of randomly generated requirement links to G
1   r1 ← radius of search region
2   r2 ← radius of cutoff region
3   for each timepoint A
4       B ← find neighboring timepoint s.t. distance < r1 and link AB = NIL and BA = NIL
5       if A.x < B.x
6           S ← A
7           E ←B
8       else
9           S ← B
10          E ← A
11      end if
12      dist ← abs( distance(S,E) )
13       if dist < r2
14          dist ← max_duration
15      endif
16      ub ← random( dist/2, 2*dist)
17      lb ←  random( -dist, dist/2 )
18      add requirement link SE with bound [lb,ub] to G
19  end for
```

Figure 5-5 **Pseudo-Code for ADD_ REQURIEMNT_LINK**

```
function RAND_TPNU( num_acts, ctrl_pct)
Input: number of activities num_acts
    percent chance of being controllable
Output: randomly generated STNU G
1   G ← NIL
2.  for i = 1 to num_acts
2.      generate an activity with ctrl_pct change of being controllable
        with start timepoint S and end timepoint E.
3.      ub ← rand( 1, max_duration)
4.      lb ← rand( 0,ub )
5.      create activity with bounds [lb,ub] and add to G
6.      assign random (x,y) location for S within (plan_length, plan_width)
7       assign location  of E of (x + ub, y )
8   end for
9   ADD_REQUIREMENT_LINKS(G)
10  return G
```

Figure 5-6 **Pseudo-Code for RAND_STNU**

In order to use the RAND_TPNU we need to set several more parameters. These include:

max_duration: the maximum duration of the activities
plan_width: : x dimension of the plan space
plan_height: : y dimension of plan space
r1 : : radius of search region
r2: : radius of cut region

For testing the DC algorithm we fixed values of each parameter except for the *plan_width* parameter, which is a function of the *num_acts.*

max_duration = 20
plan_length = 10*num_acts
plan_height = 30
r1 = 30
r2 = 10

A sample randomly generated plan is shown in Figure 5-7

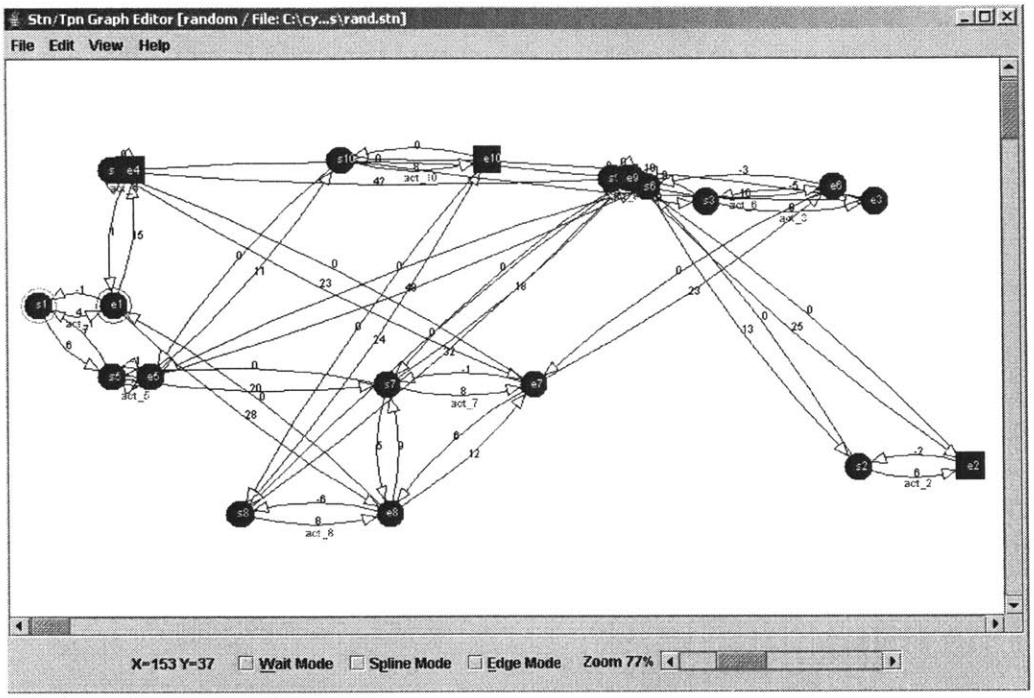


Figure 5-7 **A randomly generated TPNU generated by the RAND_TPNU algorithm**

The TPNU has 10 activities with 50% chance of each activity being controllable.

We used the RAND_TPNU generator in order to test the Fast-DC algorithm. We generated random TPNUs from 5 activities to 70 activities in increments of 5, using a ctrl_pct = 50%. At each activity level, we generated a random TPNU then ran the Fast-DC algorithm on the associated STNU. This cycle was repeated until we found a TPNU that was dynamically controllable. Figure 5-8 shows the run time of the Fast-DC algorithm plotted against the number of activities. The tests were run on an IBM laptop with a 500 MHz Pentium III processor. Figure 5-8 shows two sets of data. The data labeled "total" represents the total run time of the Fast-DC algorithm and the data labeled back-propagation represents the time the algorithm spent doing back-propagation. Specifically, this is the amount of time the algorithm took to run Lines 8-13 in the Fast-DC algorithm shown in Figure 4-21. A more detailed view of the time spent doing back-propagation is provided in Figure 5-9. The maximum time the algorithm took to do back-propagation was .41 seconds while running the trial of 50 activities.

In general, the amount of time the algorithm spent doing back-propagation was two orders of magnitude smaller than the time it took to do the other computations. Our test

show that our Fast-DC algorithm experimentally runs in $O(N^3)$. A cubic regression curve was fit to the overall data and shown in Figure 5-8. These results are significant because the current literature contains no experimental data on the run time complexity for any dynamic controllability algorithm. Recall that [Morris 2001] only provided a dynamic controllability algorithm; however, they only provided a crude upper bound on the time complexity of $O(N^6)$.

Recall the overall structure of the dynamic controllability algorithm along with the run-time complexity is as follows:

1. Run basic STN Reformulation Algorithm $\Theta(N^3)$
2. Check for Pseudo-Controllability $O(E)$
3. Run SSSP $O(NE)$
4. Back-Propagating ????
5. Run Basic STN Reformulation Algorithm $\Theta(N^3)$

The only missing link required in order to compute the theoretical worst case time complexity was the Back-Propagation Step. However, our results show that propagation costs are small compared with the other computations. Note that the STN reformulation runs in $\Theta(N^3)$. Given that the back-propagation cost is small, it follows that expected run time is $\Theta(N^3)$. This is why the overall run time fits so well to a cubic curve.

In our Fast-DC algorithm we currently use the basic STN reformulation algorithm. We have implemented the "fast" STN reformulation algorithm introduced by [Tsarmardinos 1998] which runs in $O(NE + N^2 \lg N)$ time; however, its has yet to be integrated into our Fast-DC algorithm. Substituting the fast STN reformulation algorithm for the basic one will directly improve the run time of our Fast-DC algorithm, because the STN reformulation algorithm is dominating the other computations. This simple step should greatly improve the run time of our Fast-DC algorithm for sparse plans.

The most striking result is the relatively small amount of time the Fast-DC algorithm spent doing back-propagation compared to the other computations. In Chapter 4 we provided intuitions on why the back-propagation technique would yield a fast algorithm; here we provide the empirical results to support our intuitions.

The small amount of time required to do the back-propagation also suggests an efficient incremental DC algorithm. Specifically, in Chapter 4 we showed that after running the DC algorithm once, if an edge changes in the plan, it is only necessary to call the back-propagation step in the Fast-DC algorithm to maintain the dispatchability of the plan.

More experimental work still needs to be performed. First the Fast-DC algorithm needs to be tested on more examples to more fully characterize the run-time complexity of the back-propagation step. Second, we need to implement the dynamic controllability introduced by [Morris 2001] in order to do a side by side comparison of our Fast-DC algorithm and their algorithm.

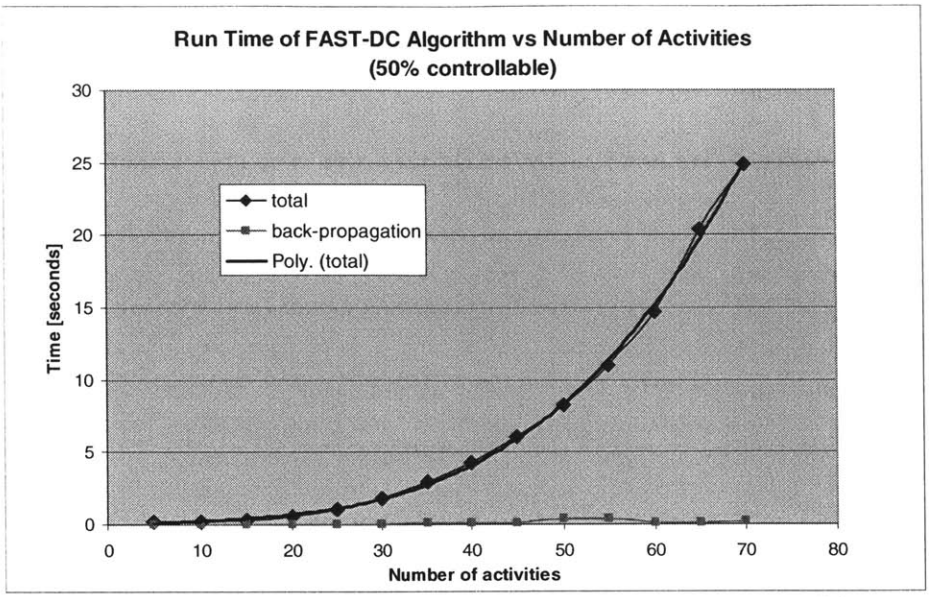


Figure 5-8 **Experimental Results of the Run-Time Complexity for the FAST-DC algorithm. The graph shows the results of a cubic regression curve fit to the overall run-time of the DC algorithm.**

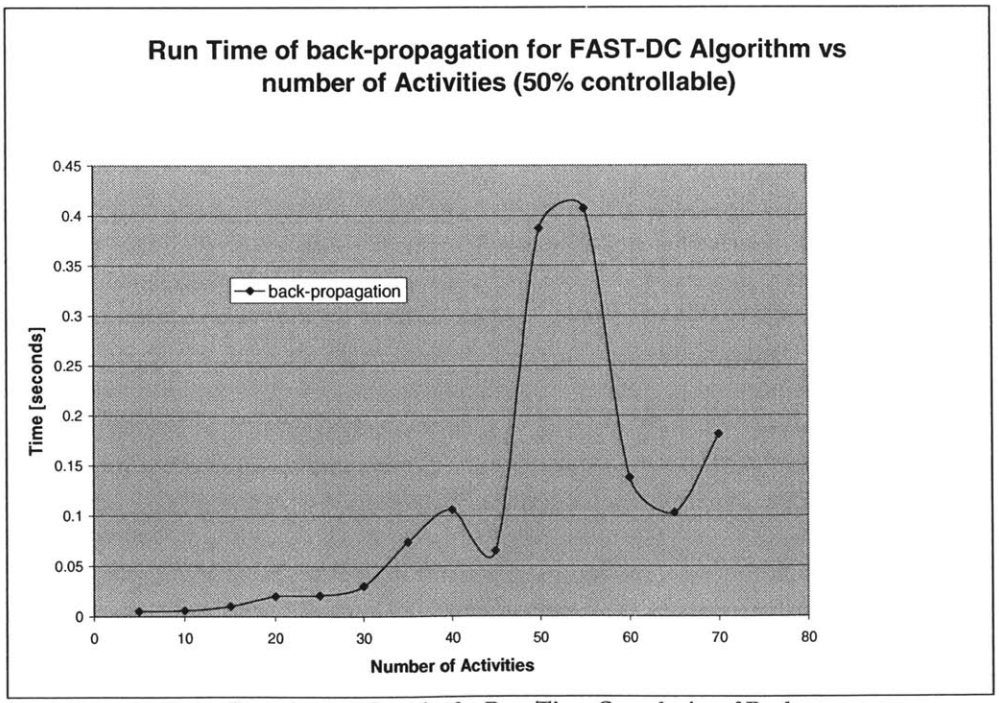


Figure 5-9 **Experimental Results for Run-Time Complexity of Back-Propagation for the FAST-DC algorithm**

5.4 Limitations and Future Work

In this section we will address some limitations to the current Hierarchical Reformulation algorithm along with direction for future work.

5.4.1 Improvements in Group Macro Representation

Currently we use a very simple model for representing the group plans in the mission plan. Specifically we model the group plan using a macro. Recall that this macro consists of an executable start timepoint, and a contingent end timepoint and a contingent link between them. The timebounds used on the contingent link is determined by computing the minimum and maximum feasible durations of the associated group plan. The different groups are free to choose any execution strategy. This technique preserves maximum flexibility within each group plan, but it sacrifices completeness of the Hierarchical Reformulation (HR) algorithm. Recall that the HR algorithm consists of three basic steps. First it reformulates each group plan in order to ensure that they can be dynamically executed. Second it determines the range of feasible execution times for each group plan and uses those time to generate an associated macro. Third it generates a fixed start time for each group plan by considering the constraints placed between the macro representations of the group plans. When the HR performs this last decoupling step it must respect the timebounds placed on the macros. Specifically, it must ensure that the fixed start times that is uses for each group plan will be consistent with the constraints of the plan no matter when the group plans finish with their activities. If the HR algorithm is unable to generate these fixed start times, the HR algorithm simple fails.

Our algorithm uses a very group centric approach. The decoupling algorithm is not able to ask the group plans to modify their plans in order to enable the groups to coordinate at the high level. In general, each group plan will tend to have some amount of flexibility it could sacrifice and still be dynamically controllable; however, our decoupling algorithm is not able to ask the group plans to give up some of this flexibility in order coordinate the groups.

In our current approach, the groups are not "team players". Consider a situation where the boss asks each employee in his research group to compute an expected time for completion of his/her individual software component and employ1 know he can complete his task in 1 week but tells the advisor an extremely conservative estimate of 4 weeks. However, upon review of the overall schedule, the boss determines that employee 1 must complete no later than 3 weeks. In our current approach, the decoupling algorithm plays the role of the boss. Currently our decoupling algorithm cannot perform this type of negotiation. This is the most sever limitation of our HR algorithm.

In order to allow the decoupling algorithm to negotiate with the groups we need a better macro representation that encodes the amount of flexibility each group is capable of sacrificing. This will allow the decoupling algorithm to steal flexibility from the group plans when it needs it.

5.4.2 Improvements in the Decoupling Algorithm

Recall the STNU decoupling algorithm removes the dependence from one group plan. Currently we generate a fixed schedule for the group plans in order to do this decoupling. However, it is possible to enable each group plan to be executed independently while still allowing the start time of each group activity to remain flexible. This flexibility will allow each group to adapt to some level of unmodeled uncertainty, hence increasing the robustness of execution. An overview of the improved STNU decoupling algorithm is illustrated in **Error! Reference source not found.**.

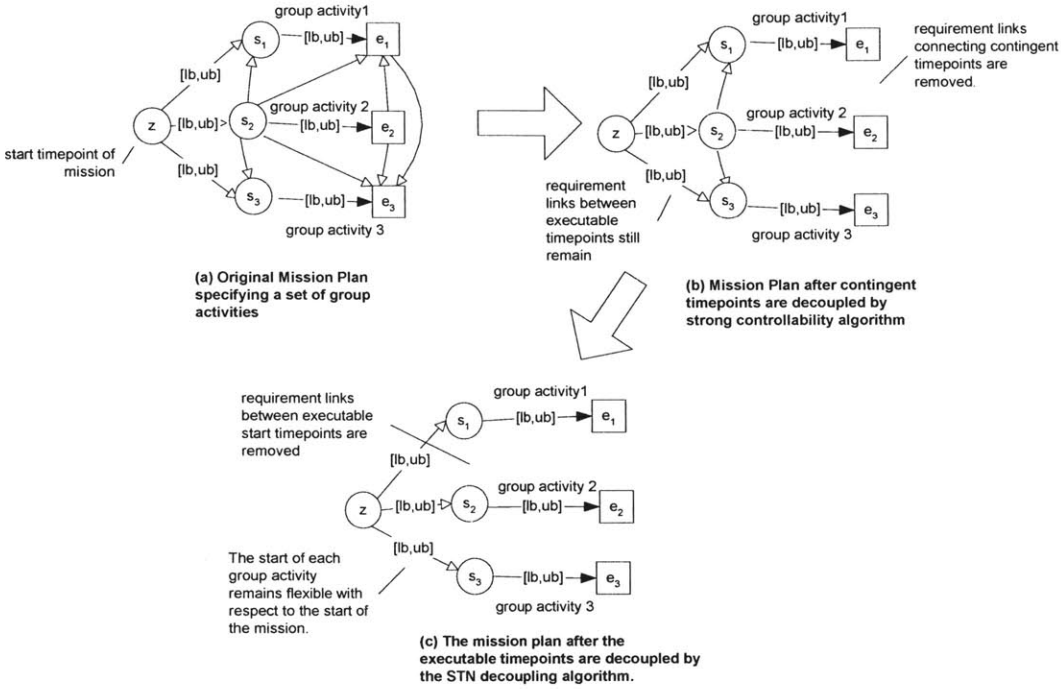


Figure 5-10 (a) **The group activities in the mission plan are decoupled using two steps.** (b) **First the contingent end timepoints are decoupled using the strong controllability algorithm [Vidal 2000] (c), Second, the activity start timepoints are decoupled using the STN decoupling algorithm [Hunsberger 2002].**

This improved STNU decoupling algorithm combines a strong controllability checking algorithm introduced by [Vidal 2000] with the STN decoupling algorithm introduced by [Hunsberger 2002]. The improved STNU decoupling algorithm breaks the problem of decoupling the group activities into two parts. First, the contingent timepoints associated with the end of each group activity are decoupled from rest of the timepoints by the using the strong controllability algorithm. This is similar to our current STNU decoupling algorithm. After this first step, we only need to consider the constrains

on the start timepoints of the group activities. Hence, we are now only working with an STN. However, instead of generating a fixed schedule for the start of each group activity, the improved STNU decoupling algorithm uses the STN decoupling introduced by [Hunsberger 2002] to decouple the start of each of the group activities. The STN decoupling algorithm makes the constraints connecting the start timepoints of each group activity dominated (redundant) by modifying the constraints that pass through the start of the mission. [Hunsberger 2002] showed that it is always possible to do this decoupling in polynomial time if the STN is consistent. This improved STNU decoupling algorithm keeps start of each group plan flexible with respect start of the mission, which in known to be executed at Time = 0.

5.4.3 Variations on the Two-Layer Architecture

There exists several variations of the two layer architecture described in the thesis. Recall that our two layer approach uses a high level mission plan and a set of lower level group plans. This two layer approach can be extended to multiple layers. Second, we currently use a static execution strategy at the mission layer and a dynamic execution strategy at the group layer. The static execution strategy is used so the different group can execute their plan independently. The agents in the different groups never need to communicate. It seems reasonable that the agents in different groups should not communicate after every action; however, in some cases the different groups should and can communicate with one another at the start and end of their group plans. For example, consider large project in which the jobs are divided up between several groups of people. A project manager does not need, nor does the manager want to know, about all the detailed interactions between the group members. However, in order to manage the project effectively, the manager needs to know when each group completes a particular group plan. In this situation, group members dynamically adjust their schedules in order to achieve coordination at the lower level and the project manager dynamically adjusts the schedule of the groups to coordinate the groups at the high level. This suggests a two layer approach which uses a dynamic controllability algorithm at both layers. This is achieved by providing a minimal amount of communication between the groups. In general, our two layer approach can be extended to a general hierarchy which use either a static or dynamic execution strategy depending on the amount of communication that is available.

5.4.4 Towards a Fully Distribution Architecture

Recall that the grand vision of this work is to create a completely distributed architecture in which all the agents participate in the planning, scheduling, and execution using a message passing. Our thesis makes steps towards this goal. Specifically, we showed how to enable different teams of agents to cooperate at execution time without requiring communication. Thus, the execution of the reformulated plan is distributed among the different teams. However, we would like both the computations of the reformulation step and execution to be distributed among all the agents. A rough technology development plan required in order to create a completely distributed architecture is provided below.

1. Develop a distributed version of the dispatching algorithm for STNs and STNUs [Muscettola 1998a, Morris 2001]. This will enable all of the agents within group plans to participate in the dynamic scheduling process. In order to distribute the dispatching algorithm, the group plan needs to be partitioned such that each agent within the group owns a portion of the group plan. Specifically, each agent should schedule its own activities. During execution each agent in the group plan will run instances of the centralized dispatching algorithm, and send execution update to other agents using a message passing system.

2. Develop a distributed version of the basic centralized STN reformulation algorithm [Muscettola 1998a]. Recall that this algorithm depends on 1) computing an APSP-graph and 2) performing set of local edge trimming operations. The APSP-graph can be computed in a distributed fashion by using by using N calls to a Distributed Bellman-Ford SSSP algorithm [Lynch 1996]. Furthermore, the developing a distributed version of the edge trimming phase should be straight forward, because each edge trimming operations is a instance of the triangle rule, which is a local computation. Furthermore, each application of the triangle rule can be done independently.

3. Develop a distributed version of the "fast" centralized STN reformulation algorithm [Tsarmrdinos 1998]. The centralized algorithm depends on 1) a Strongly Connected Components (SCC) algorithm, 2) Dijkstra's SSSP algorithm, and 3) a set of other local computations. The centralized SCC algorithm can be replaced by distributed Cidon's DFS algorithm [Tel 1994] and the centralized Dijkstra's algorithm can be replaced by the distributed Bellman-Ford SSSP algorithm. The rest of the computations are local; therefore, it should be straight forward to perform these computation is a distributed fashion.

4. Develop a distributed version of our fast dynamic controllability algorithm. Recall that this algorithm depends on the 1) STN reformulation algorithm 2) a SSSP computation and 3) a set of local back-propagation rules. We already discussed distributing both the STN reformulation algorithm and the SSSP computations above. Furthermore, the structure of the back-propagation rules should allow a simple transition from centralized to distributed version of the local back-propagation rules.

5. Develop a distributed version of the centralized HR algorithm. This depends on the strong controllability and dynamic controllability algorithms. Again these are local algorithms; therefore, creating the distributed versions should be possible without too much effort.

5.4.5 Other opportunities for future work

First, in Chapter 3 we introduced a formal definition of communication controllability. The time complexity for determining communication controllability is unsolved. Second, we assumed that the programmer was able to perform the clustering in order to create a two layer structure. Work needs to be done to do this clustering autonomously. Third,

we provided experimental evidence that the run time complexity of the Fast-DC algorithm is O(N^3). Further work is need to prove the worst case run-time complexity of the DC algorithm.

5.5 Conclusion

In this thesis we showed how to reformulate multi-agent plans in order to enable teams of agents to loosely coordinate their inter-team activities without communication; while enabling the agents to tightly coordinate their intra-team activities in the presents of uncertainty. Our two-layer approach enables the executive to focus on preparing for communication limitation at the high level and prepare for dynamically adapting to the temporal uncertainty at the low level. Recall that our approach is motivated by the observation that tight coordination under uncertainty requires communication, and fortunately, whenever the agents require tight coordination, the agent typically can communicate. On the contrary, when the agents have difficulty communication, they typically only need to loosely coordinate their activities. In these cases, it is possible to synchronize their activities by using a fixed execution strategy, therefore eliminating the need to communicate. Therefore, our two-layer approach works with nature rather than against it.

This thesis provides three primary contributions. First, at the beginning Chapter 3 we presented a formal treatment of dynamic controllability under communication limitation. Our formal analysis resulted in new type of controllably, called *communication controllability*. It is our hope this formal problem description will help stimulate research focused on planning for both communication limitations and uncertainty. Second, in the latter part of Chapter 3, we presented our novel Hierarchical Reformulation (HR) algorithm eloquently combined a strong and dynamic controllability algorithms. The HR algorithm allows different groups of agents to coordinate their activities without being in direct contact with one another. Third, in Chapter 4 we presented our new Fast Dynamic Controllability (FAST-DC) algorithm. This dynamic controllability algorithm is used in the HR algorithm; however, its use transcends the HR algorithm. One of the most significant result of thesis is that we showed that our new Fast-DC algorithm runs in $O(N^3)$ time. This is an important result because many real-time planning and scheduling problems requires the use of a dynamic controllability algorithm. We believe our Fast-DC algorithm will enable a new planning and scheduling packages to be able to efficiently cope with temporal uncertainty. Although one may argue that the applicability of the HR algorithm is still a few years off; the fast dynamic controllability algorithm introduced in this thesis is applicable to many real-world problems today![7]

[7] Please send me an e-mail at stedl@mit.edu if you have read my thesis. This will allow me to determine how many people read this thesis and allow me to answer any questions.

References

[Aldridge 2004] Aldridge, E. Jr. et. al, Report of the President's Commission on Implementation of United States Space Exploration Policy: A Journey to Inspire, Innovate, and Discover Moon, Mars and Beyond ..., June 2004.

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

[Chien 2000] Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated planning and scheduling for space mission operations. In 6th International Symposium on Space missions Operations and Ground Data Systems (SpaceOps 2000)

[Brooks 1985] Brooks, R. A. "A Robust Layered Control System for a Mobile Robot", *IEEE Journal of Robotics and Automation,* Vol. 2, No. 1, March 1986, pp. 14–23; also MIT AI Memo 864, September 1985

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

[Drummond 1994] Drummond, M., J. Bresina, and K. Swanson, "*Just-In-Case Scheduling*," in Proc. 12th National Conf. on Artificial Intelligence, 1994.

[Eberhart 2001] R. Eberhart, Y. Shi, and J. Kennedy, "Swarm intelligence", Morgan Kaufmann, 2001.

[Hunsberger 2002] L. Hunsberger. Group decision making and temporal reasoning. PhD Thesis, Harvard University, Cambridge, MA, June, 2002.

[Huntesberger 2003] Huntsberger, T.L., Pirjanian, P., Trebi-Ollennu, A., Nayar, H.D., Aghazarian, H., Ganino, A.J., Garrett, M.S., Joshi, S.S., and Schenker, P.S. 2003a. CAMPOUT: A control architecture for tightly coupled coordination of multi-robot systems for planetary surface exploration. *IEEE Transactions on Systems, Man, & Cybernetics: Special Issue on Collective Intelligence*, 33:550–559.

[Huntsberger 2004] T. L. Huntsberger, A. Trebi-Ollennu, H. Aghazarian, P. S. Schenker, P. Pirjanian, and H. D. Nayar, "Distributed Control of Multi-Robot Systems Engaged in Tightly Coupled Tasks," *Autonomous Robots*, Vol. 17, pp. 79-92, 2004

[Jonnson 2000] Jonsson, A. K.; and Morris P. H.; Muscettola, N.; Rajan, K.; and Smith, B. 2000.Planning in Interplanetary Space: Theory and Practice. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), 177-186

[Kim 2001] P. Kim, B. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of IJCAI-2001*, Seattle, WA, 2001.

[Long 2002] Long D. & Fox M. Fast Temporal Planning in a Graphplan Framework. In Proceedings from the Sixth International Conference on Artificial Intelligence Planning and Scheduling.2002

[Morris 1999] P. Morris and N. Muscettola. Managing temporal uncertainty through waypoint controllability. In *Proceedings of IJCAI-1999*, 1999

[Morris 2000] P. Morris and N. Muscettola. Execution of temporal plans with uncertainty. In *Proc. Of Seventeenth Int. Joint Conf. on Artificial Intelligence* (AAAI-00), 2000.

[Morris 2001] P. Morris, N. Muscettola, and T, Vidal. Dynamic Control of plans with temporal uncertainty. In: *Proceedings of the 17th International Joint Conference on A.I.* (IJCAI-01). Seattle (WA, USA).

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

[Muscettola 1998b] N. Muscettola, P.P. Nayak, B. Pell, and B.C. Williams. Remote agent: to boldly go where no AI system has gone before. Artificial Intelligence, 103 (1-2):5-48, August 1998.

[Parker 2000] L. E. Parker, "Current State of the Art in Distributed Autonomous Mobile Robotics", in Distributed Autonomous Robotic Systems 4, L. E. Parker, G. Bekey, and J. Barhen eds., Springer-Verlag Tokyo 2000, pp. 3-12.

[Schetter 2003] T. Schetter, M. Campbell, D. Surka. Multiple agent-based autonomy for satellite constellations. Artificial Intelligence, 145 (1-2):147-180, April 2003

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

[Tsarmrdinos 1998] I. Tsarmardinos, N. Muscettola, and P.Morris. Fast transformation of temporal plans for efficient execution. American Association for Artificial Intelligence (AAAI-98), 1998.

[Vidal 1996] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. Of 12th European Conference on Artificial Intelligence* (ECAI-96), pages 48-52, 1996.

[Vidal 1999] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistencies to controllabilities. Journal of Experimental & Theoretical Artificial Intelligence, 11:23-45, 1999.

[Vidal 2000] T. Vidal. Controllability characterization and checking in contingent temporal constraint networks. In *Proc. Of Seventh Int. Conf. on Principles of Knowledge Representation and Reasoning* (KR'2000), 2000.

[Wehowsky 2003] A.F. Wehowsky. Safe distributed coordination of heterogeneous robots through dynamic simple temporal networks. Masters Thesis, MIT, Cambridge, MA, May, 2003.

[Williams 2001] Williams, B.C., P. Kim, M. Hofbaur, J. How, J. Kennell, J. Loy, R. Ragno, J. Stedl and A. Walcott, "Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration." *Int. Symp. on Artificial Intelligence, Robotics and Automation in Space,* St-Hubert, Canada, June 2001

[Williams 2003] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott. January 2003. "Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," invited paper in *Proceedings of the IEEE*: Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pp. 212-237.