Embracing Conflicts: Exploiting Inconsistencies in Distributed Schedules using Simple Temporal Network Representations

Anthony Gallagher

CMU-RI-TR-09-02

April, 2009

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Stephen Smith, Chair
Katia Sycara
Paul Scerri
Luke Hunsberger, Vassar College

Keywords: Distributed Scheduling, Simple Temporal Networks, Continuous Scheduling

To

Sedes Sapientiae

Abstract

Simple Temporal Networks (STNs) have gained increasing acceptance in scheduling applications, and provide key features for dealing with dynamic, uncertain execution environments. STNs provide strong support for *flexible-times* schedules, which can *absorb* some of the deviations encountered during schedule execution, lessening the need for rescheduling. Further, they lend assistance to *incremental* scheduling techniques, which revise the existing schedule instead of formulating a new solution from scratch. This reduces the changes from one schedule to the next, increasing stability. Despite these advantages, STNs have received scant attention in distributed scheduling. A key limitation has been that STNs require consistency of all temporal information: An inconsistent STN provides no solution. Unfortunately, consistency cannot be guaranteed in a distributed, dynamic, and uncertain environment, where agents frequently have conflicting information due to various factors (such as communication lags). Previous approaches have concentrated on designing conservative schedules that do not encounter inconsistencies during execution. However, such guarantees are often impractical, and can lead to diminished performance.

This thesis investigates three important issues surrounding the use of Simple Temporal Networks (STNs) in a distributed scheduling environment. We start by presenting strategies to recover from the inconsistent information that naturally arises during schedule execution, and restore the STN to a consistent state. We leverage the conflict analysis tools presented by the STN framework to detect, explain and resolve inconsistencies as they arise. Our strategies extend previous work on *explaining* STN conflicts in centralized scheduling domains, adapting this technology to the additional complexities presented by a distributed environment. We present a set of "local" and "non-local" conflict resolution actions that an agent can apply to resolve inconsistencies. Local actions make changes to the agent's local schedule. Non-local actions, on the other hand, make assumptions about other agent's activities, restoring consistency while waiting for updated information. Our results show that a team of agents using these conflict resolution actions significantly outperforms a second team of agent that makes conservative assumptions to prevent inconsistencies from arising during execution.

Conflict explanation can also be used on inconsistencies encountered during the scheduling process to enhance the multi-agent schedule. This thesis develops strate-

gies that increase the robustness of the schedule, and enable agents to update previous commitments they had made to each other. Assuming a model of durational uncertainty, we have developed *Just-in-Time* Backfilling (JIT-BF), a framework that monitors the scheduled activities that are close to execution for potential failures, and takes action to prevent these failures by scheduling redundant activities. JIT-BF extracts low-probability durations from the uncertainty model and inserts them into the STN in a "what-if" mode to see *what* would happen *if* the duration does occur. Inconsistencies point to potential impending failures. By analyzing the inconsistency, its effect on the schedule can be determined, and preventative action taken to reinforce the schedule. Our results show that a team of agents that uses JIT-BF significantly outperforms a reactive team of agents that recovers from inconsistencies once they arise.

We conclude this thesis by presenting a *conflict-driven* coordination approach to updating existing temporal commitments between agents. When an agent fails to schedule a new activity because of conflicts with the scheduled activities of other agents, it can analyze the resulting inconsistency to discover what these conflicting activities are and by how much they need to move to allow the new activity to be scheduled. We present two alternative strategies that provide a trade-off between the amount of information an agent needs to initiate a coordination session, and the time needed for coordination. Our results show that agents using either strategy significantly outperform an agent team that unconditionally maintains inter-agent commitments once formed. Further, we show that agents using either strategy deliver equivalent performance, yet, the less information-heavy strategy significantly reduces the amount of communication at a negligible computational cost increase.

Acknowledgments

I would like to thank my advisor Stephen Smith for providing me with the necessary guidance in achieving the completion of this work. His advice has helped me to develop as a writer, presenter and researcher. I would also like to thank the other members of our lab for their advice, encouragement and support. Most of all I would like to thank Zack Rubinstein for giving me a "How to write a PhD thesis" manual, Laura Barbulescu for her valuable suggestions and words of encouragement, and Matt Danish, who wrote many of the software scripts and programs that were crucial to this work. I would also like to express my gratitude to Luke Hunsberger, my external committee member, for his advice on my research and his comments on the organization of this thesis. Finally, I would also like to convey my appreciation to my other committee members Katia Sycara and Paul Scerri for providing me with well-timed advice.

I am infinitely grateful to my mentors and teachers Pujya Swami Dayananda Saraswati, Sri Swami Tattvavidananda, Sri Swami Viditatmananda and Sri Mata Amritanandamayi Devi. I am blessed by their wisdom and teachings every day. I am also greatly indebted to my good friends Tikeswar Naik, Sanjiv Kumar and Aleksa Veselinovic. It is uncertain whether this work would have been completed without their encouragement and support. Most of all, I would like to express my deepest gratitude to my wife, life partner and best friend, Andreea Gallagher. Her companionship and support throughout these long years has made this work possible. This thesis is for you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Coordinating a team of agents to work together towards achieving a common goal is a complex task. The team needs to *plan* what activities the agents will perform, *allocate* these activities to the agents who will perform them, and *schedule* when they will occur [1]. It is often difficult, expensive or impossible to perform one or more of these three processes in a centralized manner, making distributed techniques necessary to solve the problem. Factors such as the geographical separation of the agents, the size of the data the team uses, or privacy concerns may prevent any single agent in the team from obtaining a full view of the problem and producing a solution (e.g., an agreed-upon schedule or plan) for the whole team [81]. Dynamics during execution introduce additional complications. For example, they might invalidate an agreed-upon schedule. In dynamic situations, the agents need to *continuously* update their solution to cope with the changes.

In this thesis, we focus on the problem of online management of a multi-agent schedule in a dynamic, uncertain environment. When a team of agents is presented with a complex plan, containing intricate temporal inter-relationships between the activities, they might try computing a schedule in advance of execution. Advance scheduling can take advantage of the plan structure to produce solutions that (1) lead to greater gains for the team, and (2) avoid pitfalls that could cause failure to fulfill the team goals. However, although a pre-computed schedule can provide crucial benefits to the agent team, the presence of uncertainty (e.g. unexpected activity durations) requires that they make assumptions at schedule-time about what might

[1] Typically, the boundary separating these three processes is hard to define, and two or all of them are often grouped together within a single process.

happen at execution-time. Dynamics during schedule execution frequently violate the assumptions made during the scheduling phase. These violations can invalidate the pre-computed schedules of the agents, giving them a short life-span. This distributed scheduling problem is still an open question: How can a team of agents take advantage of scheduling procedures that build sequences of assigned activities in advance of execution in the face of uncertain execution dynamics that *continuously* invalidate the agreed-upon schedule?

1.1 Problem Characteristics

This thesis presents strategies that address the *continuous* distributed scheduling problem, where scheduling agents need to *continuously* update their inter-dependent schedules to cope with execution dynamics. We focus our attention on problems with the following characteristics:

- *Offline planning and allocation*: The team of scheduling agents is initially equipped with a plan. Each plan activity is allocated to a specific agent in charge of executing it.
- *Oversubscribed plans*: The plan contains more activities than the agents can feasibly execute in the time allotted. The agents need to select which activities in the plan should be scheduled for execution, and when they should be executed.
- *Limited agent views*: Each individual agent is assumed to have only a partial view of the complete plan.
- *Complex temporal interdependencies*: Activities in the plan are linked through various kinds of temporal constraints. These constraints can involve activities executed by different agents, requiring coordination to satisfy the constraint.
- *Uncertain activity outcomes*: Execution dynamics can cause deviations from the pre-computed schedule. For example, activities can take longer than expected, or fail to accomplish their objectives.
- *Dynamic plan modifications*: New activities may be dynamically introduced into the plan during schedule execution, or existing activities modified (e.g. their deadlines could be changed).

1.2 Motivating Example

The distributed scheduling problem we have described can be found in many real-world situations. Take for example a military operation: A team of task forces is tasked with rescuing hostages from a group of terrorists. It is known that the terrorists are holding the hostages in houses at two locations, Alpha and Gamma, and that the houses are wired with explosives. The plan given to the team involves splitting the team into two forces, Charlie and Bravo, and striking *simultaneously* at both locations. If the synchronization fails to occur, the hostages in one of the locations will be lost.

The first action that both strike forces need to perform is to neutralize the terrorists before they can detonate the explosives or otherwise kill the hostages. Once the terrorist threat has been eliminated, the hostages need to be secured and removed from the scene. In addition, Charlie should try to capture the terrorist leader in Alpha location alive for interrogation, if time permits. The whole operation must be performed within a 30-minute time window, since the area is hostile, and the whole operation needs to be completed before a mob has time to congregate. In keeping with a basic "need to know" operating philosophy, the forces involved in the mission are only provided with as much information as they need to perform their tasks. As a result, each force potentially has only limited knowledge of the full mission plan.

While straightforward, this simple plan presents complexities that the team of soldiers need to navigate to achieve the overall goal of rescuing the hostages. First, Charlie and Bravo need to synchronize their strikes. Failure to do so will cause the force that strikes late to fail its part of the mission, since the element of surprise will have been lost. Then, there is the logical sequence of events that needs to be followed to accomplish the mission, e.g. the terrorist elements must be eliminated *before* the hostages can be secured. There is also inherent uncertainty associated with each of the actions. For example, neutralizing the terrorists may take longer than expected, delaying all other actions in the plan. Even if the soldiers stick to their schedules, the results of their actions may still deviate from the original assumptions. For example, Charlie may find that a key hostage in Alpha has been killed, reducing the value of the rescue mission. In addition to coping with uncertainty in the results of actions taken, the teams may need to make a choice among a set of actions to take. For instance, the initial plan may call for the use of a disabling gas to facilitate the entry of the teams and minimize casualties. However, the presence of the team may be detected by a forward enemy lookout, forcing the team to resort to a more forceful

and dangerous entry action. Optional actions, such as the capturing of the terrorist leader, have the potential of boosting the value of the mission.

Coordinating the execution of the joint schedules of the team is a challenging task. All members of the team (the agents) must work in tandem to achieve the mission in the presence of complex interdependencies between the plan activities that possibly include precedence constraints, preference relations, synchronization points, strict deadlines and other constraints. Further, the presence of unanticipated factors (such as psychological stress in the high-adrenaline situation of the previous example) can lead to erroneous decisions or missed opportunities. Automation tools that aid decision-making by carrying out the distributed scheduling process (fully or partially) can provide an invaluable service, reducing the margin of error and enabling the agents to concentrate on performing the tasks at hand.

1.3 General Approach

This thesis addresses the challenges of managing a multi-agent schedule in a dynamic, uncertain environment with the use of scheduling techniques based on *Simple Temporal Networks (STNs)* [33]. STNs are constraint satisfaction solvers for temporal plans. STNs represent the plan activities and their constraints as nodes and edges in a directed graph, and use shortest path algorithms to propagate the effects of the constraints on the activities. To simplify the discussion, this thesis takes the view of STNs as *objects* with *data* (i.e. the graph) and *behaviors* (i.e. the constraint propagation algorithms that act on the graph). This approach differs from previous temporal network related work [33, 75], where STNs are viewed as a *data structure* (i.e. the graph) on which propagation algorithms are applied.

The application of the propagation algorithms on the STN's graph generates a set of *flexible* start (and finish) time windows for the plan activities. These *flexible* time windows indicate the *earliest* and *latest* times that the activities can start (or finish) while meeting all of their constraints. STNs can provide key benefits to schedulers facing dynamic environments, where new schedules need to be *continuously* created. They have found increasing acceptance in a variety of scheduling domains [118, 160, 53, 21].

A scheduler can leverage the *flexible* time windows provided by the STN to create *flexible-times* schedules. These schedules *sequence* the activities to be executed, but

do not assign them *fixed* start (or finish) times. The exact start (or finish) times can then be fixed during schedule execution, using the added information of the execution trace. By delaying the fixing of the start (or finish) times of activities, *flexible-times* schedules can absorb many execution-time deviations by simply sliding the sequence of scheduled activities, thus reducing the number of deviations that require a new solution to be created. A second advantage provided by an STN representation is that a working schedule can be updated by simply adding (or retracting) edges in the STN's graph. This feature provides strong assistance to *incremental* scheduling techniques. Incremental schedulers update a working schedule (that needs to be modified due to an execution-time change) rather than creating a new solution from scratch. Incrementality provides stability between schedule revisions, dampening the effects of events on the agents' schedules, naturally and efficiently restricting the scheduler's search and reducing the need for communication.

Despite the advantages that STNs provide in dynamic execution environments, they have received little attention in distributed scheduling applications. As part of the DARPA Coordinators program, we have investigated the challenges surrounding the design of a team of STN-based scheduling agents, and developed the cMatrix agent (see section 2.3), a scheduling agent system that operates in a complex simulated environment [157, 158]. During the process of developing this system, we have identified a key impediment to wider adoption of distributed STNs: An STN, like other constraint solvers, does not provide an in-built mechanism to resolve *inconsistencies* caused by conflicting temporal constraints.

Lemma 1. *An STN with an inconsistency provides no solution.*

Inconsistencies can arise during the scheduling phase (e.g. when the scheduler tries to sequence two activities) or can be caused by an execution event (e.g. a deviation from the expected schedule that cannot be absorbed by the schedule's flexibility). While a scheduling-time inconsistency can be trivially overcome by retracting the scheduling action that caused the inconsistency, inconsistencies that arise during execution present added complications. Such an inconsistency indicates that the schedule needs to be updated. However, unlike a centralized solver, a scheduling agent may not have enough information (i.e. its view of the plan is limited) or authority (i.e. it cannot make scheduling decisions for activities allocated to other agents) to take the action required to resolve the inconsistency.

Previous multi-agent systems that use STNs have proposed two approaches: (1)

Preventing inconsistencies from occurring during execution [180, 74], and (2) using coordination to resolve inconsistencies [63, 14]. Preventative approaches can be problematic in oversubscribed, distributed domains. First, an accurate durational uncertainty model is needed that can predict the worst-case duration of scheduled activities. Such accuracy is difficult to guarantee in practice. Second, the schedules produced tend to be overly conservative, since the goal is to produce "fail-safe" schedules that won't break during execution. This conservatism can lead to sub-optimal performance. To take advantage of potential opportunities for better performance, this thesis takes a less risk-averse approach that schedules activities based on their *expected* durations. This can lead to the introduction of inconsistencies during execution (e.g. when an activity takes longer than expected), which need to be overcome to enable the STNs to continue providing solutions.

Coordination approaches present strategies that allow the agents to resolve these execution-time inconsistencies through negotiation sessions. Unfortunately, until the negotiation concludes, the agents' STNs are stuck in an inconsistent state and, hence, are unable to provide solutions for moving forward. Thus, the agents cannot respond to any incoming events that involve temporal changes, no matter how urgent. If communication between the agents is lost before the negotiation process is finished, the agents can remain in an inconsistent state indefinitely. Therefore, these approaches are undesirable in domains with high dynamics where the agents need to be constantly ready and able to cope with events as they occur, as well as in domains with weak communication facilities.

Recent research using *conflict-explanation* techniques has presented some possible alternatives. Centralized incremental schedulers have made use of *conflict-explanations* [163, 85, 84, 32] to develop strategies to analyze the *inconsistencies* that arise during the scheduling process, and resolve the inconsistencies based on this analysis [156, 15, 21]. This thesis expands the scope of these strategies to address the added complications of distributed domains. We develop strategies that use *conflict-explanation* strategies to (1) overcome execution-time inconsistencies, and (2) leverage scheduling-time inconsistencies to improve the multi-agent schedule.

1.4 Thesis Contributions

This thesis makes the following contributions:

- *Resolving inconsistencies that arise during schedule execution.* STN-based scheduling agents operating in a dynamic, uncertain execution environment must be prepared to face deviations from the assumptions made during the scheduling phase, especially those that cannot be absorbed by the schedule's flexibility and lead to *inconsistencies* (e.g. an activity lasting longer than expected). These inconsistencies need to be resolved for continued operation. In this thesis, we develop a framework to overcome such inconsistencies. In this framework, when faced with an inconsistency, we first use *conflict-explanation* of the inconsistency to categorize the type of conflict that has occurred. Then, based on the category of the conflict, we select an appropriate conflict-resolution action to perform. By equipping STN-based schedulers with techniques for overcoming execution-time inconsistencies, our framework removes a key obstacle to the wider use of STNs in distributed scheduling domains.

Conflict-explanation strategies are not only useful to analyze execution-time conflicts. Similar conflict analysis techniques can also be leveraged during the scheduling phase to analyze conflicts that occur when scheduling activities. This thesis investigates how we can use *conflict-explanation* to improve a schedule by (1) increasing the robustness of the schedule and (2) developing coordination techniques that update the scheduled times of activities in the schedule. Specifically, it makes the following contributions:

- *Exploiting scheduling-phase inconsistencies to increase schedule robustness.* When a model of durational uncertainty exists, a scheduler can make use of it to take *pro-active* actions that increase the robustness of the schedule to deviations from the expected outcomes. Using this model, we define an approach to determining potential schedule weaknesses and, when identified, taking action to strengthen the schedule. First, working in a "what if" mode, we determine *what* would happen to the schedule *if* certain activities lasted longer than their expected durations. This step is based on a probabilistic model of action durations. If an *inconsistency* arises, it is used to flag a potential problem in the schedule (e.g., an activity failing to meet its deadline). Then, we analyze the *inconsistency* to reveal what activities are in danger, and how the schedule would be affected were they to fail. Finally, based on this analysis, we take action to prevent the failure or dampen its effects.

- *A conflict-driven approach to coordinate an update to the multi-agent schedule.* When a plan contains temporal constraints between activities (e.g. precedence

constraints), agents need to form commitments with one another to ensure that those constraints will be satisfied (e.g. two agents committing to perform two activities in sequence to meet a precedence constraint). However, as execution events deviate from schedule time assumptions, the value of maintaining such commitments can change (e.g. if the "target" activity in a precedence constraint is no longer valuable). When this occurs, it may be desirable to update the commitments. We have developed a pair of strategies for accomplishing this type of coordinated schedule change. Our coordination process leverages *inconsistencies* encountered while attempting to schedule a newly introduced activity. These inconsistencies are analyzed to discover if an existing inter-agent commitment is a root cause. Such an inconsistency flags a potential coordination opportunity. Based on the analysis, we determine what activities in the multi-agent schedule need to move to make space for the new activity, and by how much. This information can then be used to guide a coordination process to update the schedule.

In the following subsections we expand on these three areas of contribution.

1.4.1 Recovering from Inconsistencies that Arise during Execution

As mentioned in the previous section, STN-based scheduling agents can produce *flexible-times* schedules that are capable of absorbing many deviations from the expected schedule. Yet, this flexibility is ultimately a function of the problem constraints and scheduled load of the agent, and some deviations can introduce inconsistencies that cannot be absorbed by the STN. An STN with conflicting constraints is in an *inconsistent* state, and cannot be queried for the temporal bounds of the activities in the schedule until it is brought back to consistency. To bring the STN back to consistency, it is necessary to update the agent's schedule.

This thesis expands on previous research in *conflict-explanation* techniques within an STN context [156, 15, 21]. While previous research has studied how to use these explanations in centralized scheduling domains, distributed domains present new challenges that are still unaddressed. In centralized domains, the scheduler can resolve inconsistencies by rearranging any scheduled activities, as needed. In multi-agent domains, on the other hand, an agent's scheduler lacks the ability to modify remote

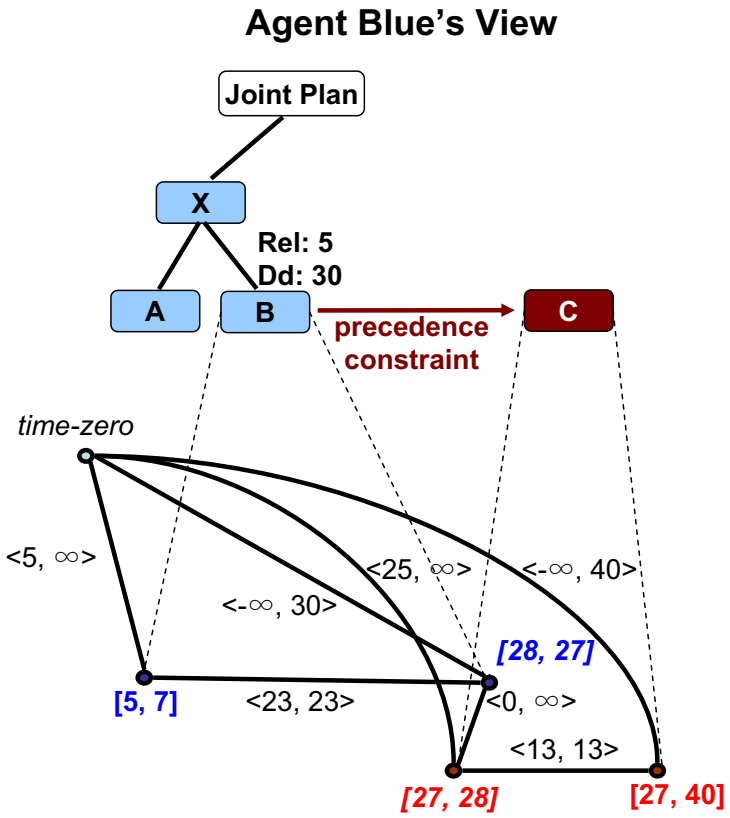

Figure 1.1: An unexpectedly long duration causing a conflict.

activities that are being executed by other agents, yet known to the local agent due to interdependencies. Figure 1.1 displays a situation where two activities, local activity B and remote activity C, are scheduled to execute, and B must finish before C starts. When B finishes late, it introduces an *inconsistency*. To restore consistency, it is necessary to revise the schedule, so that the conflict is removed. While a centralized solver could resolve the conflict by removing C from the schedule, the agent that owns activity B does not have the ability to unschedule C, since it is being executed by another agent.

Our work develops a framework for managing inconsistencies in a distributed set of STNs that represent the local views of different agents, while respecting the limitations that agents cannot make changes to the schedules of other agents. Our approach combines *conflict-explanation* strategies to analyze the conflict and a set of repair strategies to resolve the inconsistency. The analysis of the inconsistency produces a categorization of the temporal constraints involved. This categorization

is then used to match the inconsistency to a given repair strategy. Applying the repair strategy restores local consistency to the STN, ensuring that all scheduled local activities satisfy the known constraints that apply to them. The resolution strategies progress from local repair actions, where revising the local schedule of the agent suffices to resolve the conflict, to more complex schedule-repair actions when constraints involving remote activities cannot be accommodated, and assumptions must be made about the continuing validity of previously reported remote information.

We test a team of agents using this "repair" framework against a "conservative" team using a *preventative* strategy similar to those used in STN controllability approaches [180, 114]. Specifically, the "conservative" agents use an uncertainty model to schedule activities according to their maximum durations. Our results show that the "repairing" agents significantly outperform the "conservative" team on problems with tight deadlines, while having equivalent performance on problems with loose deadlines (i.e. when there is enough time to execute activities to their maximum durations). These results indicate the value of our "repair" framework, showing that it provides a clear advantage to an agent team executing a schedule in a dynamic environment.

1.4.2 Just-in-Time Backfilling

While Section 1.4.1 discussed how to recover from inconsistencies caused by execution dynamics once they have occurred (an inevitability in dynamic, uncertain environments), this section focuses on attempting to either prevent inconsistencies, or dampen their deleterious effects, by taking *pro-active* steps before the inconsistency happens. By performing this analysis ahead of time, advantageous scheduling opportunities can be identified that might otherwise be lost if the agent waited until the deleterious event actually occurred.

This work relies on the presence of a stochastic model of environmental uncertainty for the durations of the plan activities. This model enables agents to identify weak areas in the schedule, and attempt to fortify them, making pro-active decisions that increase the robustness of the generated schedule.

Multi-agent domains add complexity to this process. In situations where agents lack full visibility of the problem and authority to modify problem structures assigned to other agents, unanticipated events that invalidate the agents' schedules may require

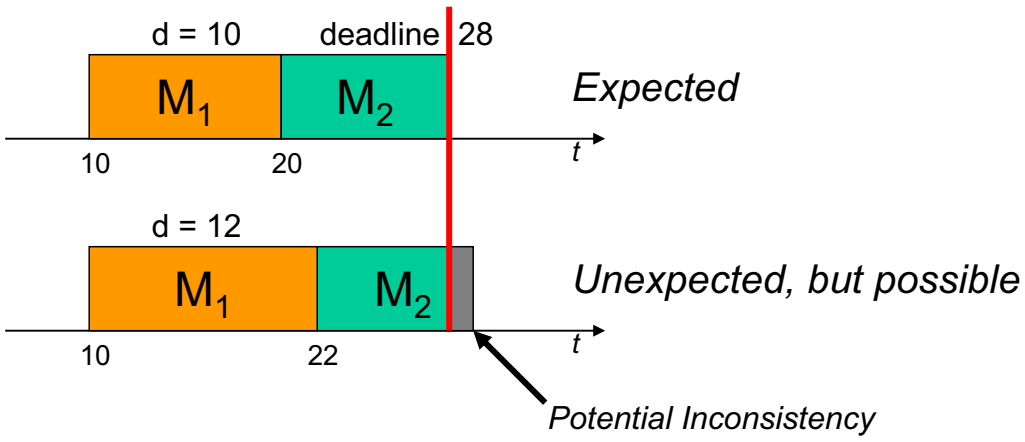

Figure 1.2: Using a *hypothetical* inconsistency to discover a potential impending failure

inter-agent negotiation and a coordinated response.

We define an approach that reasons about schedule robustness *continuously* throughout schedule execution. This *continuous* analysis enables us to discover potential failure points as execution proceeds in a *Just-in-Time (JIT)* fashion. The key intuition behind this JIT approach is that the weak areas of the schedule change as the schedule executes: Activities that previously seemed safe may become risky, and vice-versa. By *continuously* analyzing the schedule for potential weaknesses, past outcomes can be exploited to narrow the search for weaknesses. While previous techniques have used similar *continuous* approaches to increase the schedule robustness during schedule execution in centralized domains [161, 66, 67], distributed applications have received little attention.

We define an approach that uses the durational uncertainty model to uncover potential points of failure by introducing "unexpected" durations [2] for "close-to-execution" activities [3]. Each unexpected duration is entered into the STN in a "what-if" mode to learn *what* would happen *if* the unexpected duration were to occur. An *inconsistency* denotes a potential impending failure. An analysis of this *hypothetical inconsistency* reveals what activities in the schedule would be affected if the *inconsistency* were to occur. Using this information, the agent can calculate how the performance of the schedule would be degraded.

[2] An unexpected duration is a duration that is different from the expected value, based on the durational uncertainty model.

[3] A close-to-execution activity is a scheduled activity whose time for execution is approaching.

Figure 1.2 shows a simple scenario where an agent discovers a potential impending failure by introducing an unexpected duration for activity M_1. The top of the figure shows the expected agent's schedule. The schedule contains activities M_1 and M_2, with expected durations of 10 and 8 minutes respectively. However, the durational uncertainty model specifies that M_1 might last 12 minutes with some non-zero probability. The bottom of the figure shows that if M_1 overruns to 12 minutes, M_2 will fail to meet its deadline. This potential failure is discovered by entering a duration of 12 for M_1 into the STN, causing a *hypothetical inconsistency*.

When an impending failure is likely (which can be determined by a calculation based on the probability of the "unexpected" duration happening), and its consequences are deleterious to the schedule (based on the analysis of the *hypothetical inconsistency*), it may be worth taking some *pro-active* steps to prevent the failure from occurring, or to at least diminish its effects. In our domain of interest, we assume that the initial multi-agent plan given to the agents includes interchangeable, non-mutually-exclusive *redundant* activities. In such domains, a schedule can sometimes be made stronger by inserting redundant actions. We call this process *backfilling*. We have designed two strategies to insert redundant activities in the schedule: a *passive* strategy, and an *aggressive* strategy. The passive strategy relies on agents informing each other about potential weaknesses in their schedules. When an agent B is informed by agent A that one of A's activities may fail, B will try to insert a redundant activity to the failing activity if its schedule is not negatively affected. The aggressive strategy works similarly, but with the difference that A is allowed to *coerce* B to insert a redundancy, even if B's schedule is negatively affected, provided their collective schedules benefit.

We have tested our strategies by comparing a "reactive" agent team that uses only the recovery framework described in section 1.4.1 to overcome inconsistencies when they occur, and a "pro-active" agent team that additionally tries to protect itself against these inconsistencies using the strategies just described. Our results show that the "pro-active" agent team significantly outperforms the "reactive" agent team across a wide range of problems with different levels of difficulty, including a set of problems used during the evaluation phase of the DARPA Coordinators project.

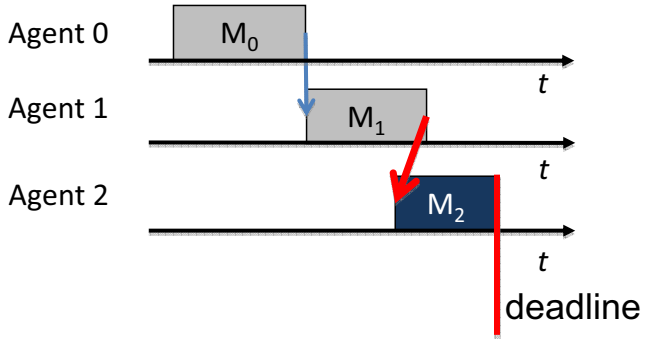

Figure 1.3: Conflict-driven coordination

1.4.3 Conflict-driven Coordination to Update Inter-Agent Commitments

Our work continues to focus attention on how an STN framework can provide significant advantages during the scheduling search by illustrating the unique opportunities for inter-agent coordination that STNs bring to multi-agent scheduling applications. In particular, we develop a *conflict-driven* form of agent coordination that locates potential coordination opportunities by using *conflict-explanation* techniques to analyze scheduling-phase inconsistencies. This conflict-driven coordination technique provides agents with an ability to coordinate the "moving" of currently scheduled activities in cases where it might improve their joint schedule.

We have developed two alternative conflict-driven strategies: the *recursive* strategy and the *full-chain* strategy. Scheduling-phase inconsistencies occur when the scheduler attempts to schedule an activity, but fails because it conflicts with currently scheduled activities. This failure is flagged by an *inconsistency* in the STN. By analyzing this inconsistency, the scheduler can detect if the conflict involves remote activities that, if moved (or removed), would enable the rejected activity to be scheduled.

Consider the situation in Figure 1.3. Agent 2 attempts to schedule activity M_2, but fails, because the precedence constraint between M_1 and M_2 cannot be met. The reason is that Agent 1 has scheduled M_1 to finish too late to enable M_2. Examination of the conflict reveals that if Agent 1 were to slide M_1 earlier, Agent 2 could successfully schedule M_2. Agent 2 can use this information to initiate a query to accomplish this change.

The recursive and full-chain strategies differ in how they organize the coordination

process, presenting a trade-off between the amount of information needed to coordinate and the time spent coordinating the schedule change. The *recursive* strategy requires less information to coordinate: an agent knows about all the activities that are directly linked to its "local" activities (e.g. Agent 2 knows about M_1 because it's directly linked to its "local" activity M_2). For the example in Figure 1.3, Agent 2's failed attempt to schedule M_2 prompts it to send a request to Agent 1 to move M_1. Since M_1 cannot move unless M_0 also moves back, Agent 1 then sends a *recursive request* to Agent 0 to move M_0. The *full-chain* strategy, on the other hand, requires more information, but it is faster. This strategy assumes that each agent knows about every activity in a *chain* of which a local activity is part (e.g. Agent 2 knows about both M_0 and M_1 because its local activity M_2 is part of the M_0-M_1-M_2 chain). For the example in Figure 1.3, Agent 2 sends a request to *both* agents, Agent 0 and Agent 1, to move their activities M_0 and M_1 back, so that M_2 has room to be installed.

Our results confirm that agent teams using either coordination strategy outperform a baseline agent team that continues to enforce all commitments between agents that have previously been made. The comparison between the two strategies confirms the trade-off between them, with the recursive strategy transmitting less information, while taking longer to coordinate than the full-chain strategy.

1.5 Thesis Outline

This rest of this thesis is organized as follows:

Background Chapter 2 provides a survey of topics relevant to the work in this thesis. First, it presents Simple Temporal Networks (STNs) and describes how temporal inconsistencies can be detected when they occur. Then, it presents the problem domain, the C_TAEMS planning language used to describe the multi-agent plans, and the Coordinators problem. Finally, it shows how a C_TAEMS plan can be encoded into an STN, and how the STN flags an inconsistency.

Related Work Chapter 3 presents a review of related work in relevant areas: (1) *Continuous Scheduling*, where a scheduler acts in a tight loop with the executor component, (2) *Multi-Agent Coordination*, both implicit and explicit, and (3) *Temporal Networks*, a review of STNs and related technologies.

Experimental Methodology Chapter 4 presents a description of the experimental setup used in this thesis, the problem generator we used to create scheduling problems, the characteristics of the generated problems, and how we evaluated our results.

Recovering from Inconsistencies Chapter 5 presents a framework for recovering from inconsistencies caused by execution events (e.g. the late finish of an activity). First, the categorization of the temporal constraints involved in the inconsistency is described, followed by how to select a conflict resolution action based on this categorization. Finally, the experimental results obtained using this framework are presented.

Just-in-Time Backfilling Chapter 6 presents our strategies for leveraging a model of durational uncertainty to "pro-actively" increase the robustness of the schedule. We explain how we select the "horizon" activities that we examine for potential failure, and introduce their *unexpected* durations (based on the uncertainty model) into the STN to locate what durations would introduce an inconsistency, were they to occur. This analysis is used to decide whether to augment the schedule with redundant activities. Finally, we present the experimental results obtained using this pro-active techniques.

Conflict-Driven Coordination Chapter 7 presents our strategies for using inconsistencies encountered while attempting to schedule a new activity to guide a coordination process. First, the chapter presents the set of techniques built into the cMatrix agent to form inter-agent commitments. Then, we explain the two techniques we have developed to update these commitments. The experimental results show a comparison of these two techniques.

Conclusions and Future Work We present the conclusions and derived observations from this thesis in Chapter 8. Then we describe several possibilities for enhancing the power of the techniques presented in this thesis. Finally, we conclude the thesis with a discussion of some open issues regarding yet to be addressed problems.

Chapter 2

Background

The scheduling problem studied in this research is *distributed* and *continuous*: Agents coordinate the activities they schedule, and the scheduler and executor components of each agent need to work in a tightly integrated loop. Our technical approach to this problem is centered on Simple Temporal Network (STN) technology. In this chapter, we start with an overview of STNs and how they can be applied to scheduling domains. We then present the problem domain that we have addressed and provide details of the cMatrix agent, our scheduling agent implementation.

2.1 Simple Temporal Networks

Most flexible-times scheduling approaches use the simple temporal problem constraint network representation (or more simply Simple Temporal Network (STN)) [33], as STNs provide powerful semantics for encoding and maintaining networks of temporal constraints. An STN consists of a graph $G =< N, E >$ and the algorithms that act upon it. Each node in the set of nodes, N, represents a time point. Each edge in the set of edges, E, represents a distance constraint between two time points in N. A distance constraint between time points $X_i \in N$ and $X_j \in N$ is of the form:

$$(X_j - X_i) \leq d \tag{2.1}$$

where d is a number in $(-\infty, \infty)$ that represents the maximum distance between the two time points. A special time point, called *time-zero* (which denotes the *beginning of time*), grounds the network and has a fixed value of 0.

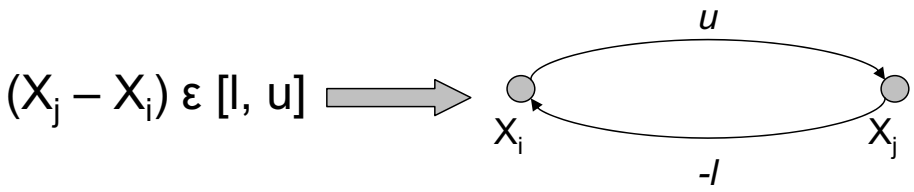

Figure 2.1: A temporal constraint in an STN

Typically, plans and schedules use the *activity* as a basic abstraction. An activity represents an action that must be performed, and it is defined by two time events: its start and finish. Each of these two events can be represented in an STN by a time point in its graph. A wide variety of temporal constraints acting on activities (such as release dates, deadlines, durations, precedence constraints, etc.) can be represented as edges between the time points in the graph. A temporal constraint that specifies the minimum and maximum time that can elapse between two time points can be expressed as:

$$(X_j - X_i) \in [l, u] \tag{2.2}$$

where l and u are the lower and upper bounds on the time elapsed between the time points. Constraints of the form in Equation 2.2 are represented using two edges in the distance graph, corresponding the lower and upper bounds of the constraint respectively. Figure 2.1 shows the graph for the constraint $(X_j - X_i) \in [l, u]$. The lower bound, l, is represented by a directed edge from X_j to X_i with weight $-l$, while the upper bound, u, is represented by a directed edge from X_i to X_j with weight u. Unary constraints that act on a single time point (e.g. a deadline constraint acting on the finish time point of an activity) can be cast into the form in Equation 2.2 by using the *time-zero* point. For example, the constraint $X_k \geq C$ is equivalent to the temporal constraint ($X_k -$ *time-zero*) $\in [C, \infty)$.

The STN computes the temporal bounds of all the time points in N (i.e. the earliest and latest times they can execute) by propagating the effects of all the distance constraints in E through the application of a shortest path algorithm. Some STN implementations have used an All-Pairs Shortest Path (APSP) algorithm, such as the Floyd-Warshall procedure [47, 186], which produces the "distances" (i.e. the feasible time intervals) between every pair of time points in the STN. These approaches use an auxiliary data structure called the *distance matrix* to maintain the time intervals between the time points [33, 75]. However, computing and maintaining all the dis-

tances between the time points is unnecessary in most scheduling applications, where only the temporal bounds of the plan activities are needed. These temporal bounds can be calculated by finding the distances between the special time point *time-zero* and the rest of the time points in the STN. These distances can be found using a Single Source Shortest Path (SSSP) algorithm, such as the Bellman-Ford [5, 82], with *time-zero* as the source time point. The use of an SSSP algorithm reduces the computational ($O(N^3)$ vs. $O(N^2)$) and space ($O(N^2)$ vs. $O(N)$) requirements of the STN over APSP approaches. While incremental APSP approaches can reduce the computational complexity of adding temporal constraints to an existing STN to $O(N^2)$, the removal of temporal constraints, as well as the insertion of additional time points into the STN remains computationally expensive [77]. Given our emphasis In this thesis work on continuous, closed-loop scheduling processes, our STN implements an incremental version of the Bellman-Ford algorithm that supports the efficient addition and removal of temporal constraints and time points into an STN by maintaining state information about each time point [22].

Applying the Bellman-Ford algorithm on the STN graph produces the distances *from* the *time-zero* point *to* every other time point. These distances represent the *latest temporal bounds* of the time points (i.e. the latest feasible time the time points can be executed). A second application of the Bellman-Ford algorithm to a slightly modified graph (with the direction of the arcs reversed, and the edges' weights negated) produces the distances *from* every time point in the graph *to* the *time-zero* point. These distances represent the *earliest temporal bounds* of the time points (i.e. the earliest feasible time the time points that can be executed). The two temporal bounds compose the *feasible time window* of the time points: The window of time over which the time point must be executed to respect its constraints. The temporal bounds of the activities are simply the temporal bounds of the their start and finish time points. These four bounds are given special names: The temporal bounds on the start point of an activity are called its *earliest start time (EST)* and *latest start time (LST)*. The temporal bounds on the finish point are called its *earliest finish time (EFT)* and *latest finish time (LFT)*.

The addition of a temporal constraint or a change to an existing constraint can give rise to an *inconsistency* or *conflict*. The STN detects these conflicts by checking for *negative cycles* in the graph G.

Theorem 2.1. *An STN is* inconsistent *when its distance graph contains one or more* negative cycles *[33, 75].*

A *negative cycle* is a loop in the STN's distance graph where the sum of the edges' weights is negative. When inconsistencies exist, the Bellman-Ford algorithm fails, and the temporal bounds of the activities in the STN cannot be computed (since no feasible solution that respects all constraints can be found). The temporal bounds can only be computed once the conflicts are removed from the STN, restoring it to a consistent state.

To use temporal networks in a continuous scheduling domain, it is important to model the passage of time. A recent technique, called the Augmented Simple Temporal Network (ASTN) [76], adds a *now* time point to an STN that keeps track of the current time. In an ASTN, scheduled, but not yet executed, activities link their start and finish time points to this *now* point through precedence constraints. These links denote that these unexecuted time points must occur *after* the *now* point; in other words, they represent future events. When the time for execution arrives, these precedence links are removed, denoting that these points are now in the past. In addition, new links are created between these executed time points and the *time-zero* point. These constraints represent the actual time the activity started and finished.

2.2 The Problem Domain

We now define the class of multi-agent scheduling problems studied in this thesis work. As outlined in Section 1.1, we assume that plans are generated prior to execution, and that each agent is initialized with a partial view of the multi-agent plan before the start of execution. We further assume that the plans are generally oversubscribed (i.e. there are more activities than can be feasibly done). The agents are tasked with producing a schedule or maintaining an initially provided schedule, and managing these schedules throughout their execution so that the overall quality of all successfully executed activities is maximized (where quality is simply a reward metric). The agents are cooperative: They have no selfish interests, working only for the good of the team.

The DARPA Coordinators program presents a concrete domain that implements this multi-agent scheduling problem formulation. The Coordinators problem is inherently distributed: Agents have limited visibility of the team problem, and they lack the authority to modify problem structures assigned to other agents. Agents are unit-capacity: They can only perform one activity at a time. Agents are tasked with producing and maintaining coordinated schedules, and maximizing the quality

of the combined solution. The agents construct schedules assuming that activities will accrue their *expected* quality and execute for their *expected* duration (based on the uncertainty model). As the schedule executes, the agents monitor the execution of their activities for events that invalidate these assumptions. These events necessitate dynamic rescheduling. The rescheduling process may involve re-coordination between various agents due to the interactions between activities in the plan. The continuous loop between the scheduling of activities and their execution requires tight integration between the scheduler of the agent and the activity executor component.

2.2.1 C_TAEMS Modeling Language

Coordinators plans are defined using the C_TAEMS modeling language [8], a dialect of *TAEMS* (Task Analysis, Environment Modeling and Simulation) [35, 71]. C_TAEMS is a planner-independent representation that expands on the Hierarchical Task Network (HTN) formulation [45], and provides rich semantics to express complex multi-agent plans. Distributed scheduling approaches for finding the optimal solution to a C_TAEMS plan are known to be NEXP-complete [6, 176], making the use of heuristic approaches a necessity.

C_TAEMS plans are characterized by the following features:

- They present a fully-expanded HTN-like tree structure, where higher-level activities called *tasks* represent abstract activities that can be decomposed into more detailed activities, and leaf activities called *methods* are directly executable by the agents. The root of the activity hierarchy is called the *taskgroup*.
- Each task accrues quality from its children according to a *Quality Accumulation Function (QAF)*. The QAF specifies how the quality accrued by the children is combined to obtain the task's quality. This quality is simply a *reward* value.
- The QAFs also define the structure of the activity hierarchy. QAFs determine whether a task is an AND-type activity that requires that all child activities must be accomplished, or an OR-type activity where only one child activity needs to be executed.
- Each method is assigned an *owner* agent. A method can only be executed by its *owner*.

- The only resource requirement of a method is its *owner* agent's time.
- Each method defines a set of one or more *outcomes*, which model the environmental uncertainty during schedule execution along the quality and duration axes. Each outcome specifies a discrete probability distribution for the possible quality values that the method can accrue during execution and its possible durations.
- Methods can be aborted while they are executing, but not resumed once aborted. An aborted method accrues no quality.
- The specification of plan activities can include execution *time windows* that specify a *release* date (i.e. the earliest possible start time of the activity), and a *deadline* (i.e. the latest possible finish time of the activity). Release and deadline constraints on a task also constrain the execution time window of all lower-level activities under the task.
- Complex interrelationships between activities can be defined to require tight coordination between the agents to attain quality. These relations include joint activities requiring synchronization and temporal sequencing between pairs of activities. These sequencing relations are called *Non-Local Effects (NLEs)*. Both hard and soft NLEs can be defined. Hard NLEs specify constraints that must be enforced, while soft NLEs define preferences.
- The *goal* of the agent team is to maximize the quality of the *taskgroup*.

C_TAEMS defines six types of QAFs: SUM, MAX, MIN, SUMAND, SYNCSUM and EXACTLY_ONE. The first three are self-explanatory. Tasks with these QAFs accumulate quality from their children by summing their qualities, taking the maximum quality of any child, or the minimum quality of any child, respectively. SUMAND tasks are similar to SUM tasks except that all their children must execute with positive quality before the task accumulates their quality. SYNCSUM tasks represent synchronization points; these tasks accumulate quality by summing the qualities of all the children that start at the same time. These tasks are designed to model the element of surprise: child activities that start late lose this element and accrue no quality for their parent task. EXACTLY_ONE tasks represent exclusive OR tasks: If only one child accrues positive quality, the task accrues that quality. Otherwise, the task accrues no quality.

C_TAEMS provides four types of NLEs: *enables*, *facilitates*, *disables* and *hinders*. All these NLEs represent temporal relations between activities that govern how the target activity accumulates quality. Enables and facilitates NLEs are similar to causal relations. Enables NLEs specify a hard constraint: The source activity must obtain positive quality before the target activity starts, or the target activity cannot accrue any quality. Facilitates NLEs specify a soft constraint: If the source activity accrues quality before the target activity starts, then the target activity will accrue higher quality and/or have shorter duration. Disables and hinders NLEs represent a temporal relation between a source and target activities, where the target must start executing before the source accumulates quality. Disables NLEs specify a hard constraint: If the relation is not respected, then the target activity will not accrue any quality. Hinders NLEs specify a soft constraint: If the target starts executing after the source has accrued quality, then the target will obtain lower quality and/or have a longer duration.

While these NLEs are in many ways similar to standard temporal relations used in other scheduling domains, they differ in that they are triggered by the source activity attaining positive quality. This property has interesting consequences when the source of the NLE is a task. SUM tasks, for example, accrue positive quality as their child activities finish. An enables NLE with a SUM task as the source activity will be triggered as soon as the source task attains positive quality (i.e. as soon as any child activity finishes).

2.2.2 The Coordinators Problem

Coordinators problems are executed in a real-time simulated environment called the Multi-Agent System Simulator (MASS) [68]. A team of Coordinators agents receives a complete multi-agent plan using the C_TAEMS language described in section 2.2.1. However, each agent is initialized with only a *subjective* view of the problem, which is a subset of the complete, or *objective*, view. This subjective view includes all the methods the agent owns, and the chain of parent tasks linking those methods to the taskgroup. Further, the view includes any activity (task or method) that is connected via an NLE to a method the agent owns. We note that these NLEs may be linked not to the agent's method itself, but rather to one of its ancestor tasks that link the method to the taskgroup.

An example of a simple C_TAEMS task network for two agents (Agent 1 and

Agent 2) is shown in Figure 2.2. The shaded areas encompassing a subset of the activities in the plan denote the subjective views of the agents. The scenario shows a problem where the two agents must cooperate to attain positive quality for the taskgroup.

The taskgroup, TT_1, is a MIN task. Thus, it will accrue positive quality when both T_1 and T_2 accrue positive quality. However, T_1 enables T_2, meaning that T_2 and its children must wait for T_1 to accrue positive quality before starting. T_1, as a MAX task, will attain quality if either of its children methods M_{1a} or M_{1b} attains quality. Since both these methods are assigned to Agent 1, Agent 1 can choose which one to perform. Typically, an agent would attempt the higher quality method, and if unable to perform it, the agent would *fall back* to a lower quality method. When either of these two methods completes with positive quality, T_2 is enabled. Agent 2 owns both methods M_{2a} and M_{2b} under T_2. Since T_2 is a MAX task, Agent 2 will make a decision between them based on its availability.

2.3 The cMatrix Agent

The cMatrix agent [1] has been developed to explore the use of flexible-times, STN-based scheduling techniques in the distributed scheduling scenario of the Coordinators program described in section 2.2. This agent serves as the test-bed for the ideas developed in this thesis. The cMatrix agent is initialized with a subjective view of the team plan in C_TAEMS form, and an initial schedule of methods to execute. This initial schedule is computed prior to execution using a centralized solver that creates schedules using expected values for method durations and qualities. The agent maintains this schedule during execution, monitoring for events that necessitate changes, and coordinating with other agents to reschedule when they occur. The cMatrix agent has a scheduler-centric design: The scheduler is the central component of the agent, not only producing the agent's schedule, but also maintaining the belief state of the agent, and locating opportunities for coordination with other team members [157, 158]. This section will briefly describe how cMatrix agents interact with the

[1] The cMatrix agent was developed by a team of researchers from SRI International and Carnegie Mellon University, in collaboration with individuals at Cornell University, Harvard University, the University of Maryland (College Park) and Vassar College. In Year 1, the Carnegie Mellon team was responsible for developing the core STN-based scheduler. The overall agent was re-engineered by Carnegie Mellon for Year 2.

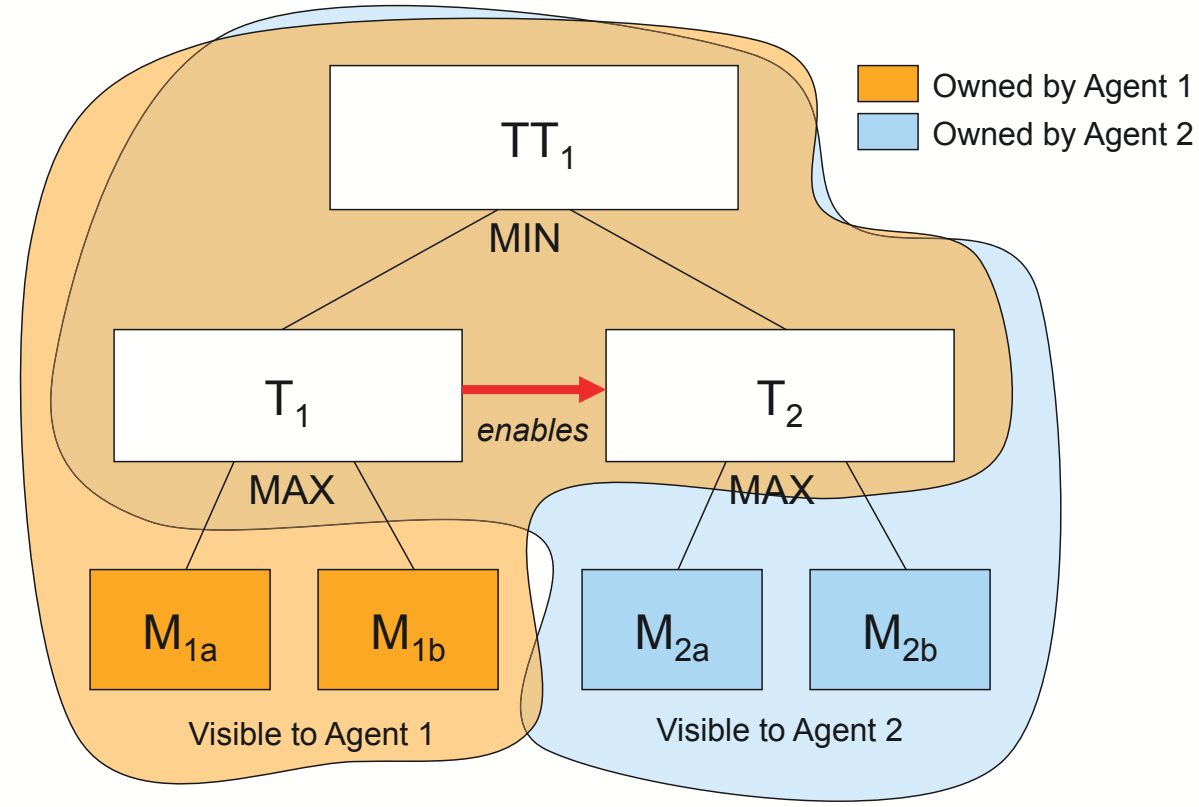

Figure 2.2: A C_TAEMS task network with 4 methods owned by 2 agents, and an enables.

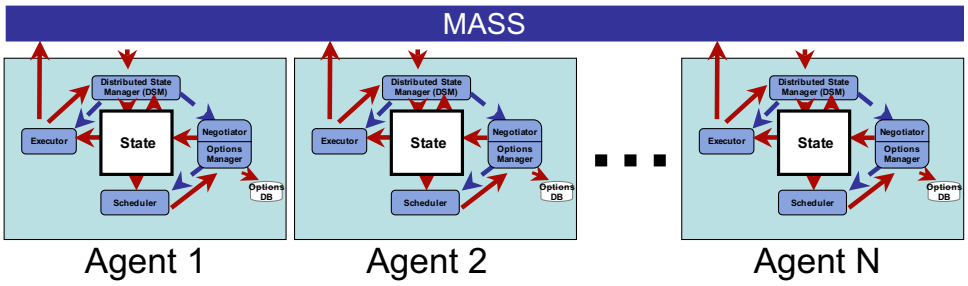

Figure 2.3: The MASS simulator with connected cMatrix agents

environment and each other, and some internals of the agent implementation.

2.3.1 Interaction with the Environment

cMatrix agents interact with the MASS real-time simulator. All agent communication is channeled through the MASS, including agent-to-agent messaging. While channeling all agent communication through a single point would be unrealistic in a fielded application, this mechanism facilitates the gathering of communication statistics in the simulated environment. During the simulated execution of a plan, the MASS is in charge of keeping agents updated about the current time (measured in discrete intervals called *ticks*), inserting uncertainty into activity execution, and notifying the agents about changes to the plan. In turn, the agents make requests to the MASS for methods to start execution when their time arrives, request methods to be aborted when no longer useful, and send messages to other agents for the MASS to relay. Figure 2.3 shows the structure of the cMatrix multi-agent system.

Coordinators scheduling problems have been designed to incorporate environmental uncertainty. The MASS introduces both *modeled* and *unmodeled* uncertainty. The MASS injects modeled uncertainty during schedule execution by selecting values for the quality and duration of methods sent by the agents for execution that deviate from the expected values of these quantities (based on the uncertainty model in the C_TAEMS plan). Unmodeled uncertainty also occurs, with random method failures (where a method completes but doesn't accrue any quality), and delays in the actual start time of methods after they are sent for execution. These failures and delays are outside of the uncertainty model in the C_TAEMS plan.

The MASS can also change the original plan of the agents, forcing the agents to reschedule. The current implementation of the MASS can introduce two types of model changes: introduction of additional activities, and changes to the existing activities. Changes to existing activities include modifications to their release times or deadlines, and the quality/duration probability distributions of methods.

2.3.2 The cMatrix Agent Architecture

An architectural depiction of a cMatrix agent is illustrated in Fig 2.4 [2]. Each agent is composed of five subsystems:

- The Current State Database: The state database consists of the activity model

[2]For simplicity, the agent-to-agent communication is shown as a direct process.

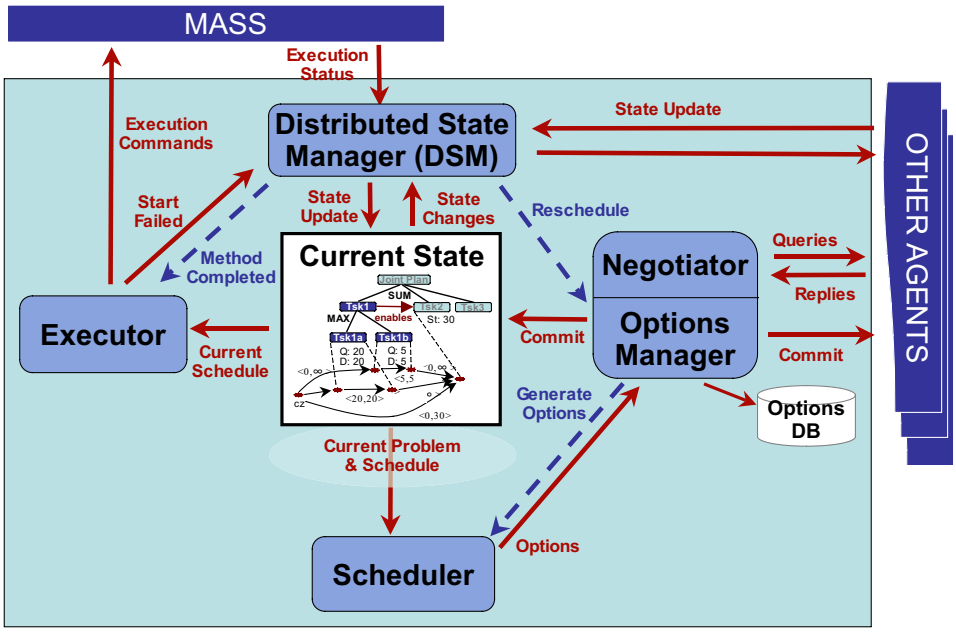

Figure 2.4: Coordinators Agent Architecture

and its encoding in the agent's STN as time points and constraints. The state of activities and scheduler-imposed constraints (such as sequencing of scheduled methods) introduces further constraints between the activities. Passage of time during schedule execution is modeled using a slightly modified version of the ASTN algorithm described in Section 2.1. The underlying STN uses an incremental Bellman-Ford algorithm [22] that updates the temporal bounds of activities as constraints are added, deleted or modified. The state database is shown in a central position as it is used by all other subsystems.

- The Executor: The executor is responsible for sending methods to the MASS to begin their execution. The executor may *fail* a scheduled method if its latest possible start time has been reached and all the activities that enable the method have still not completed executing. The executor may also *abort* an executing method if its deadline is reached and the method has not completed.

- The Distributed State Mechanism (DSM): The DSM monitors local activities and sends updated information about them to other agents in the system when they incur changes in their quality, duration, time bounds or status. The DSM sends the updated activity information to all agents whose subjective view of the

problem includes the activity. The DSM provides a form of implicit coordination between the agents, pushing the information needed about activities so that agents can make more informed local decisions.

- The Scheduler: The scheduler manages the agent's state, updating it in response to outside messages (from the MASS or other agents) or as a result of the scheduling process. Other components in the agent react to the actions of the scheduler. For example, the DSM sends updates about the local activities to other agents when these activities experience changes, and the negotiator sends requests to the scheduler to generate coordination options that can be pursued with other agents to try to improve the collective schedule.

- The Negotiator: The negotiator is in charge of coordination sessions with other agents to find opportunities to improve the collective schedule of the team. The negotiator looks for coordination opportunities when certain events occur during the execution of the schedule. The agent initiating a coordination session queries other agents for information or scheduling options and decides whether the agents involved in the negotiation should make a coordinated change to their schedules based on the information it receives.

The Scheduler

The cMatrix agent's scheduler uses a greedy quality-maximizing heuristic to determine an agenda of high-quality methods, and attempts to install this agenda [3]. The scheduler consists of two components: a *quality propagator* and a *method installer*, which cooperate in a tightly integrated loop to manage the schedule of the agent so that all problem constraints are respected. The quality propagator processes the C_TAEMS activity hierarchy received from the MASS and produces an agenda of *contributor* methods for the installer to schedule. The contributors are the methods that, if scheduled, would lead to the highest local quality for the agent. They are generated without regard to resource or temporal constraints. In essence, the quality propagator solves a relaxed infinite-resource capacity problem, leaving the task of imposing the unit capacity and temporal constraints to the method installer. The installer iterates through the contributors attempting to schedule them. Although the main task of the procedure is to install each method given to it, the scheduling

[3]The installation of a method involves the insertion of the method on the agent's timeline

process it spawns can be more extensive. Many methods have enablers, and if these are not scheduled, the method installer will try to schedule these enabler activities before attempting to install the method. When the installation of a method fails, the quality propagator is invoked again to produce a new agenda of contributor methods that does not include the failed method. This new agenda of contributors may no longer include some previously installed methods. When this occurs, these methods are removed from the agent's schedule, and the installer is invoked on the new agenda.

The quality propagator assumes that activities will accrue their *expected* quality, and ignores their durations while forming the contributors list. The method installer, on the other hand, can be operated in two modes: *expected* and *conservative*. In *expected* mode, the installer assumes that methods will last their expected durations, while in *conservative* mode, it assumes maximum durations. The default installer mode is *expected*, which can result in inconsistencies when methods take longer than their expected durations during execution. The *conservative* mode prevents inconsistencies from arising during execution, but can lead to lower quality.

The Negotiator

The negotiator is in charge of coordination sessions to improve the multi-agent schedule by increasing its overall quality. The negotiator identifies a set of *non-local options* that pinpoint a set of modifications that other agents can perform to their schedules that would lead to an increase in the local schedule's quality. These options are used by the *initiating agent* (i.e. the agent that starts the coordination session) to formulate *queries* to other team members. These remote agents respond to the queries by assessing how the schedule changes they make to fulfill the *requests* in the *query* would impact their local schedules. If the combined quality change (of the initiating and queried agents) produces a net gain, the initiating agent commits all agents in the coordination session to perform the joint action.

The initiating agent generates non-local options through *optimistic synchronization*. Using optimistic synchronization, the scheduler is directed to search for opportunities to increase the quality of the schedule by making optimistic assumptions about remote enabler activities. In brief, the scheduler enters a *what-if* mode where it optimistically assumes that a remote enabler to a local contributor method can be scheduled to precede it. If the remote enabler is unscheduled, the scheduler assumes it is scheduled. If it is scheduled, but too late to precede the local contributor, the

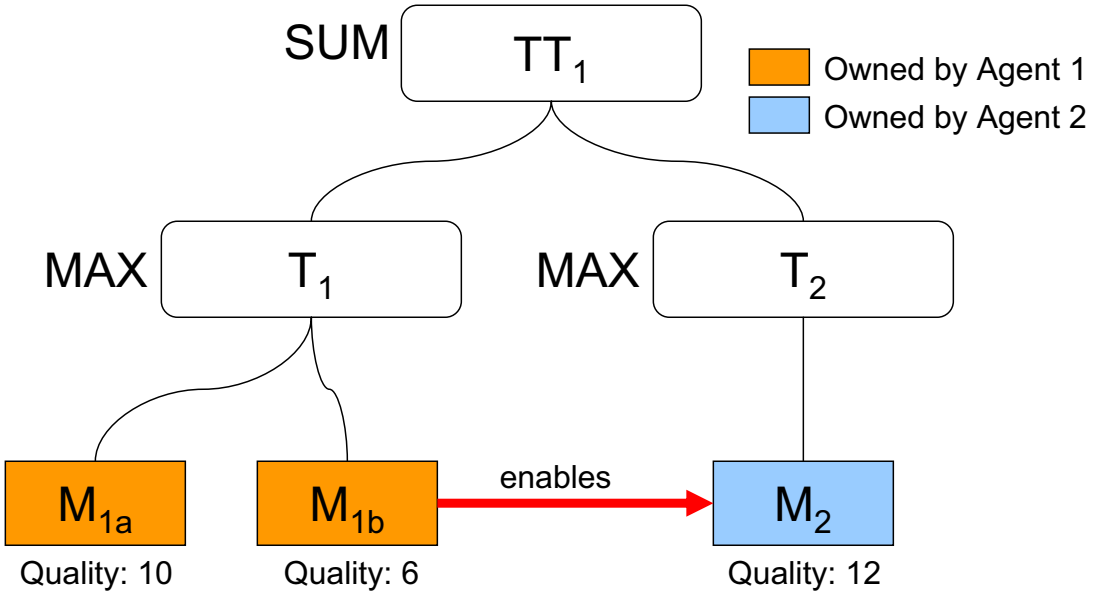

Figure 2.5: Non-local option generation with optimistic synchronization

scheduler assumes it can be rescheduled earlier. The scheduler then constructs a hypothetical schedule under this condition and calculates the local quality that could be attained. If the local quality increases, a non-local option is created. This non-local option specifies the remote enablers that were assumed scheduled.

Consider for example, the structure in Figure 2.5. Let's assume that Agent 1 has scheduled M_{1a}, but M_{1b} is unscheduled, contributing 10 to the quality of the taskgroup. Since M_{1b} is not scheduled, Agent 2's method M_2 is not enabled, and it cannot contribute to quality. Using optimistic synchronization, Agent 2 will optimistically assume that M_{1b} is scheduled and generate a non-local option. Under this option, Agent 2 contributes 12 to the quality of the taskgroup. This option is sent to Agent 1, and Agent 1 assesses the impact of scheduling M_{1b} on its schedule. Let's assume that Agent 1 responds positively to Agent 2 with a new schedule that discards M_{1a} in favor of M_{1b}. The contribution of Agent 1 to the quality of the taskgroup drops to 6. However, the net gain of the negotiation session is $12 - 4 = 8$, and the agents will commit to this option.

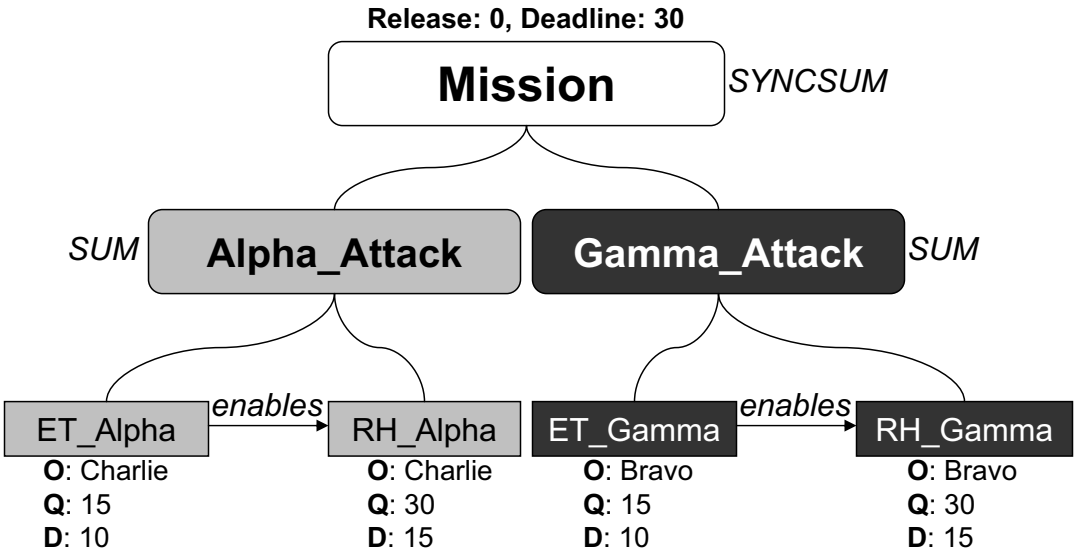

Figure 2.6: An example C_TAEMS problem for a hostage rescue mission.

2.4 Representing C_TAEMS Plans using an STN

To solve the Coordinators problem using STNs, the temporal components of a C_TAEMS plan need to be represented in the STN's graph, where activities are mapped to time points and temporal constraints to edges. In the next sections, we introduce a simple C_TAEMS plan and show how we can encode the different temporal components of the plan in an STN.

2.4.1 Our Motivating Example as a C_TAEMS Plan

We now revisit a simplified version of the hostage rescue scenario described in section 1.2 and sketch it as a C_TAEMS plan: Two teams, Charlie and Bravo, need to *simultaneously* attack at locations Alpha and Gamma. Each team will eliminate the terrorists it encounters at the site (estimated time 10 minutes) and rescue the hostages held there (estimated time 15 minutes) [4]. The mission must be finished in 30 minutes.

The C_TAEMS plan shown in Figure 2.6 encodes the requirements of the mission. *Mission*, the taskgroup activity, has a release time of 0 and a deadline of 30, specifying

[4]The remainder of the chapter assumes that the installer is operating in the default expected-duration mode. All activity durations are inserted into the STN, assuming they will last their expected duration.

the 30-minute time window for completing the rescue operation. *Mission* has a SYNC-SUM QAF, indicating that the child activities (Alpha_Attack and Gamma_Attack) must start simultaneously. These children, *Alpha_Attack* and *Gamma_Attack*, are tasks aggregating the activities necessary for attacking locations Alpha and Gamma respectively. They have SUM QAFs, indicating that the qualities of any successfully executed child activities are added up. The children under these tasks are the executable methods. Task *Alpha_Attack* has two children methods, *ET_Alpha* and *RH_Alpha*. *ET_Alpha* represents the executable goal of eliminating the terrorists at location Alpha. The method's "owner" is Charlie. It has an expected quality of 15 and an expected duration of 10 minutes. *RH_Alpha* represents the goal of rescuing the hostages at location Alpha. The method is again owned by Charlie, and it has an expected quality of 30 and an expected duration of 15 minutes. An enables NLE links *ET_Alpha* and *RH_Alpha*, indicating that eliminating the terrorists must be accomplished before the hostages can be rescued. The methods under *Gamma_Attack* mirror those under *Alpha_Attack* except that they are owned by Bravo.

2.4.2 Transforming the C_TAEMS Plan into an STN

The C_TAEMS plan shown in Figure 2.6, although small, presents five types of temporal constraints acting on the activities: *Duration* constraints, *time window* constraints, *structural* constraints, *precedence* constraints (the enables NLEs), and *synchronization* constraints. We will first examine each of these constraints and show how they can be represented within an STN framework[5], and then proceed to show how to tie these constraints together so that the C_TAEMS plan is fully represented in an STN.

Duration Constraints

Duration constraints represent the time that must elapse between the start and finish of an activity. The duration d of an activity X (represented in an STN by start time point X_S and finish time point X_F) can be expressed by the constraint $(X_F - X_S) \in [d, d]$ [6]. The top of Figure 2.7 shows activity X with its duration d, and the bottom

[5]The representations are not unique. There is typically more than one way of representing temporal constraints in an STN.

[6]Note that this equation assigns a *fixed* duration to activity X (i.e. the minimum and maximum durations are both d). An alternative approach, similar to that used in STN controllability approaches (see Section 3.2.2) would be to assign only a minimum duration to X, with an unconstrained maximum duration. While this approach maintains greater flexibility, it does not interact

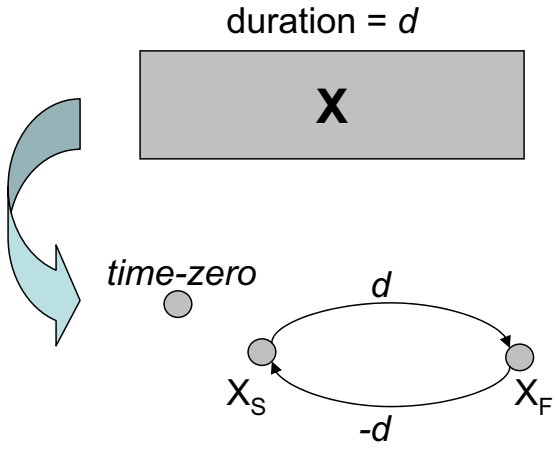

Figure 2.7: The STN for activity X and its duration constraint.

shows the STN encoding.

Time Window Constraints

Time window constraints represent the period of time over which an activity can execute. A *release* constraint establishes the earliest time an activity can start execution; a *deadline* constraint establishes the latest time an activity can finish execution. The release time r and deadline d of an activity X (represented by start time point X_S and finish time point X_F) can be expressed by the constraints ($X_S -$ *time-zero*) $\in [r, \infty]$ and ($X_F -$ *time-zero*) $\in [-\infty, d]$ respectively. Figure 2.8 shows activity X with its release and deadline constraints at the top, and their STN encoding at the bottom. Note that the ∞ edges can be omitted, since they do not constrain the distance between the time points.

Structural Constraints

Structural constraints represent *containment* in an activity hierarchy. Parent activities *contain* their children, implying that the execution of a child activity must occur within the time window of its parent. For a parent activity P with child activity C, two structural constraints link their start time points (P_S and C_S) and their finish time points (P_F and C_F). These constraints can be expressed as $(C_S - P_S) \in [0, \infty]$

well with some NLE constraints in C_TAEMS plans (i.e. disables and hinders).

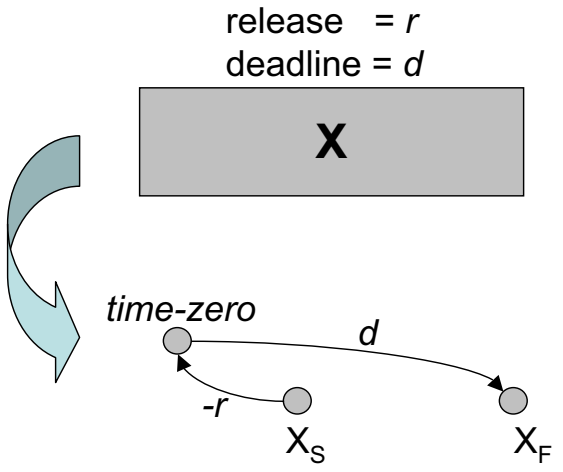

Figure 2.8: The STN for activity X and its time window.

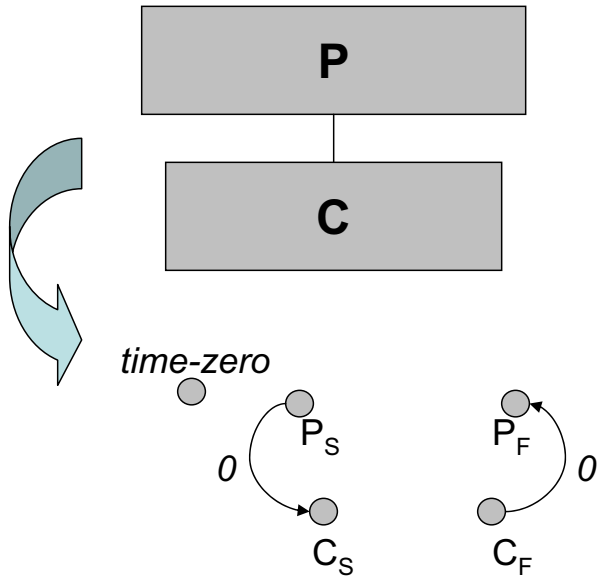

Figure 2.9: The STN for activities P and C and the structural constraints between them.

and $(P_F - C_F) \in [0, \infty]$. Figure 2.9 shows activities P and C at the top, and their STN encoding at the bottom.

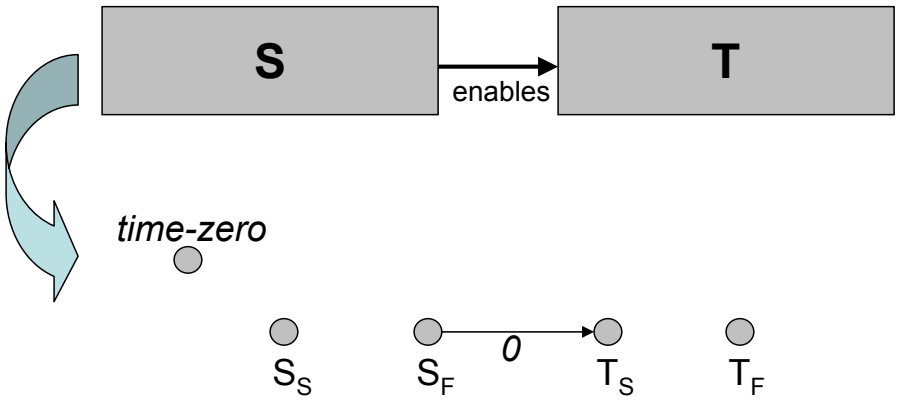

Figure 2.10: The STN for activities S and T and the precedence constraint between them.

Precedence Constraints

Precedence constraints represent the sequencing of activities. The *source* activity must finish before the *target* activity starts. For source activity S and target activity T, a precedence constraint links the finish time point of S (S_F) and the start time point of T (T_S). The constraint can be expressed as $(T_S - S_F) \in [0, \infty]$. Figure 2.10 shows activities S and T at the top, and their STN encoding at the bottom. Again, the ∞ edge can be omitted, since it does not constrain the distance between the time points.

Synchronization Constraints

A synchronization constraint represents the simultaneous execution of two time points, which are said to be "rigidly connected" [173]. In our example, the start of two activities must be synchronized. For two activities X and Y that must start simultaneously, a synchronization constraint links the start point of X (X_S) to the start point of Y (Y_S). The constraint can be expressed as $(Y_S - X_S) \in [0, 0]$. Figure 2.11 shows activities X and Y at the top, and their STN encoding at the bottom.

Tying it All Together

Now that we have all the pieces in place, we can put them together to form the distance graph of the STN for our sample C_TAEMS plan. We follow the same convention as

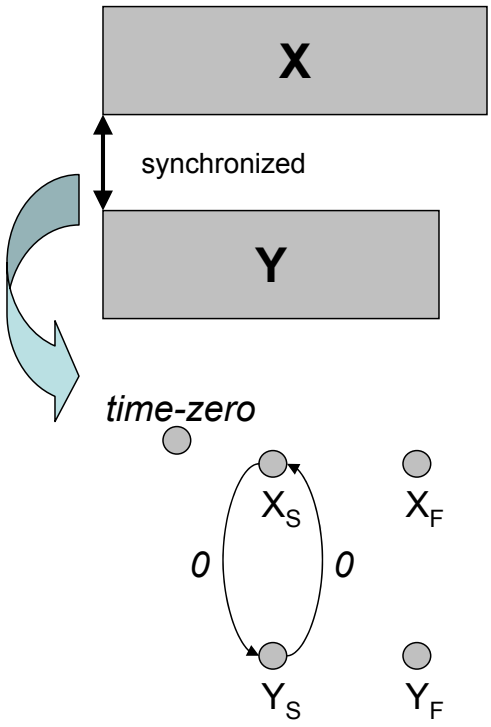

Figure 2.11: The STN for activities X and Y with synchronized starts.

in previous sections, denoting the start and finish time points of an activity X by X_S and X_F, respectively. Figure 2.12 shows the temporal information in the C_TAEMS plan encoded in an STN.

2.5 Resolving STN Inconsistencies

Changes to the schedule that cannot be absorbed by the flexibility of the activities' temporal bounds introduce inconsistencies into the STN. STN inconsistencies can arise from execution dynamics that violate schedule-time assumptions or from scheduling actions that are infeasible. The former denote events that have occurred and need to be assimilated. In such cases, the inconsistency must to be resolved to assimilate the event. The latter, on the other hand, occur as the agent attempts to improve its schedule. Although these inconsistencies can be resolved by simply retracting the scheduling action that caused them, an analysis of the inconsistency can point to potential areas of schedule improvement.

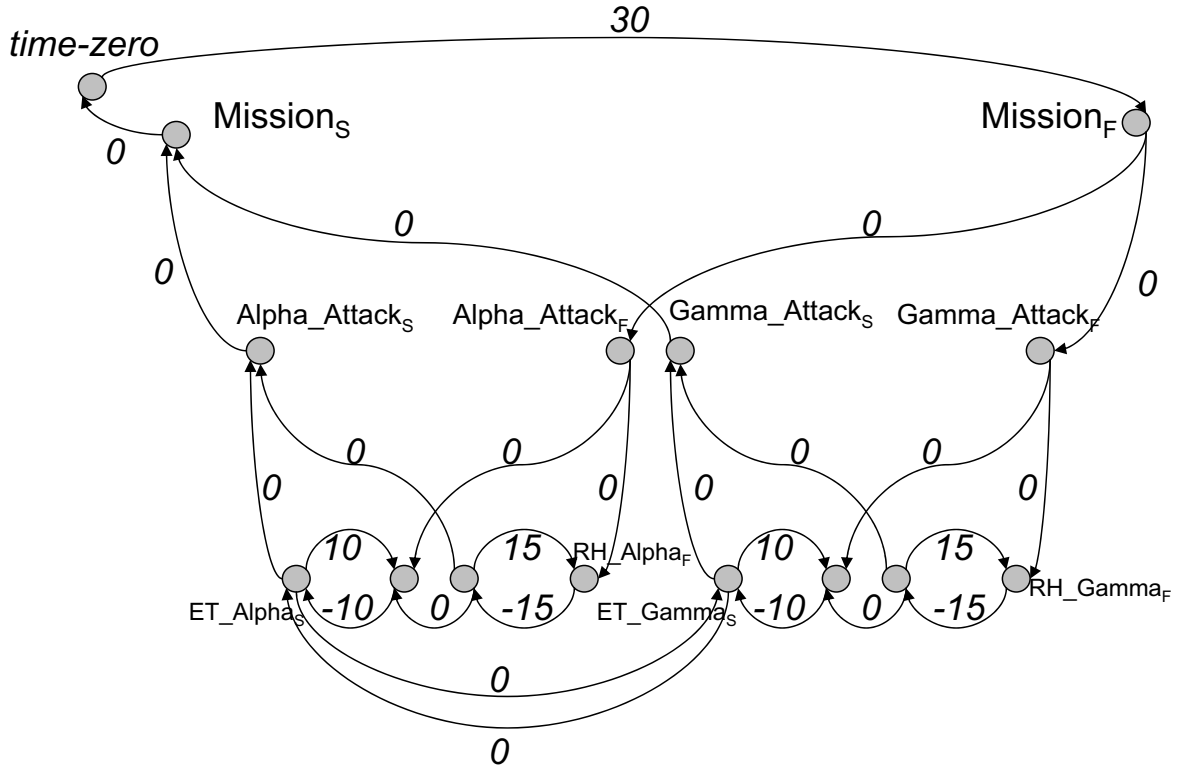

Figure 2.12: The STN for the example C_TAEMS plan.

In this section, we illustrate how we can use STN *conflict-explanation* techniques to analyze an inconsistency and resolve it. We present an example that uses an execution-time inconsistency, based on a possible execution trace for team Charlie's cMatrix agent during the execution of the C_TAEMS plan in Section 1.2. The execution trace introduces an unexpected late finish for one of the scheduled activities, resulting in an STN conflict.

2.5.1 An Execution Trace of Team Charlie's Schedule

At the start of the plan execution, both teams (Charlie and Bravo) are provided with their subjective views of the plan, and an initial schedule to execute. The subjective view of team Charlie is shown in Figure 2.13. It contains the methods that Charlie is charged with executing, along with their ancestor tasks. Charlie's initial schedule and initial STN at the start of execution (minute 0) are shown in Figure 2.14

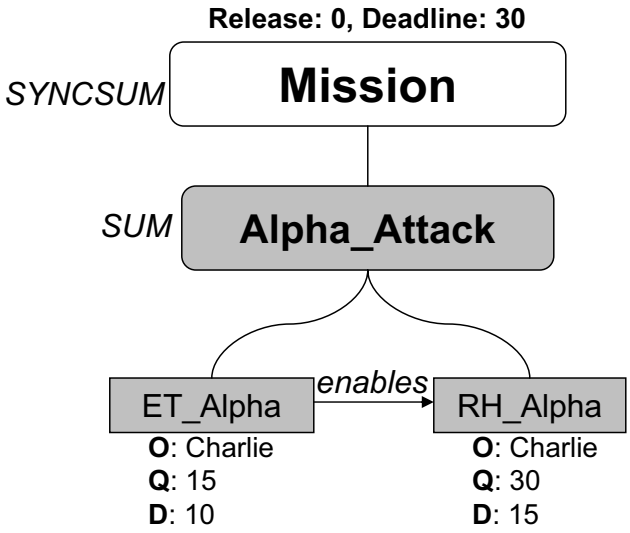

Figure 2.13: The subjective view of Team Charlie

[7]. Note that the synchronized counterpart to Charlie's *ET_Alpha* method, Bravo's *ET_Gamma*, is not part of Charlie's subjective view. This makes the imposition of the synchronization constraint between these two methods (as advocated in section 2.4.2) infeasible. The cMatrix agent overcomes this difficulty by *fixing* the start time of the synchronized method to the time obtained from the initial schedule (time 0 in this case). The constraint fixing the start time is expressed by $ET\_Alpha_S -$ *time-zero* $\in [0, 0]$ (where ET_Alpha_S is the start time point of ET_Alpha). Method *RH_Alpha*, in contrast, retains a flexible start time (with EST=10 and LST=15 minutes).

Team Charlie starts executing its schedule at time 0 minutes. The terrorists at location Alpha are engaged and a battle commences (method *ET_Alpha*). Unexpectedly, the enemy's resistance is much stiffer than anticipated. Despite Team Charlie's best efforts, after 16 minutes, the enemy still has not been subdued. At this point, there is insufficient time to rescue the hostages: There are only 14 minutes remaining of the initial 30 minute time window for the mission, and the hostage rescue activity (method *RH_Alpha*) is expected to last 15 minutes. The conflicting constraints place the STN in an inconsistent state. The inconsistency disables the STN, and resolution actions need to be taken to obtain an updated set of time bounds for the plan activities.

[7]The cMatrix agent uses an ASTN representation that includes a *now* time point to manage passage of time. The *now* point is ignored in the example for simplicity's sake.

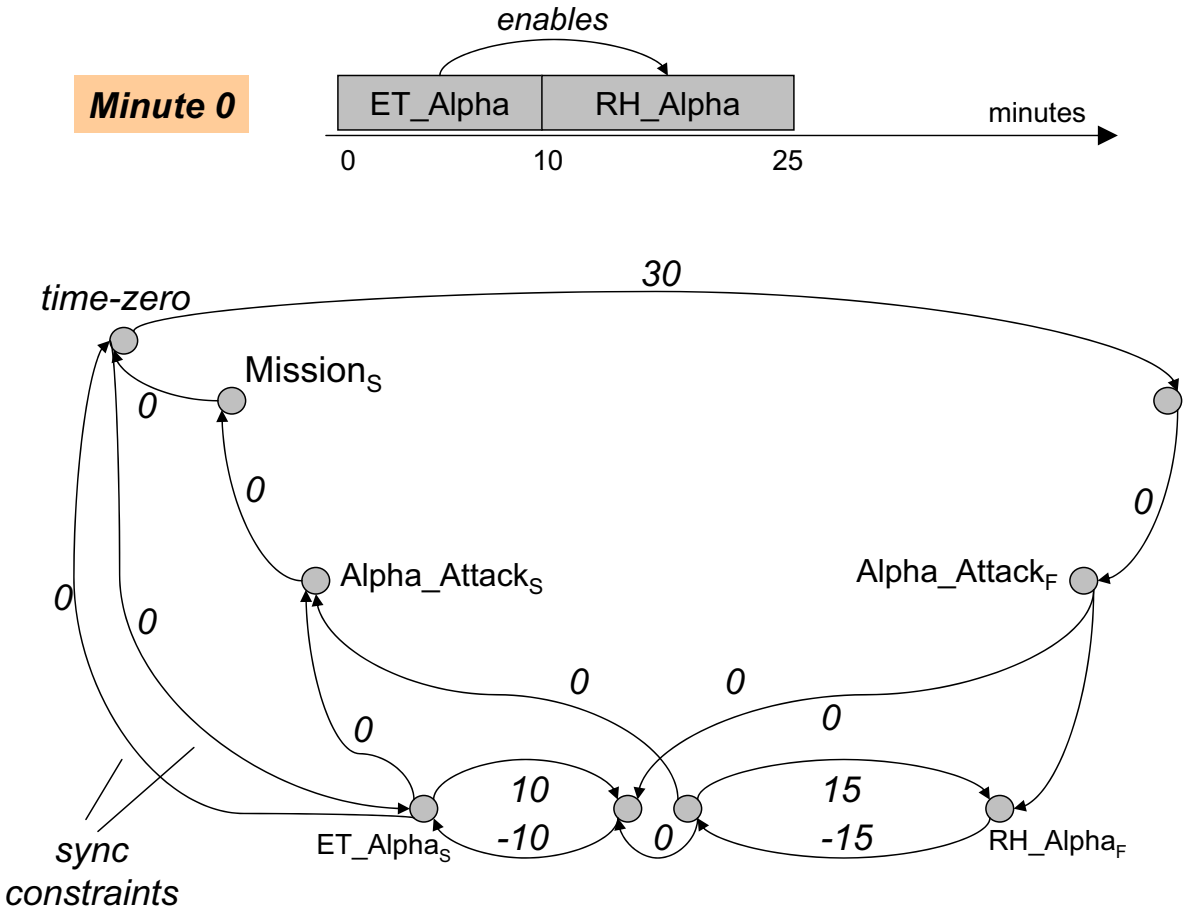

Figure 2.14: The initial schedule and STN of Team Charlie

2.5.2 Conflict Explanation in STNs

As explained in section 2.1. STNs are a graphical encoding of a Temporal Constraint Satisfaction Problem (TCSP). STNs use shortest path graph algorithms to solve the TCSP. Unfortunately, when these algorithms (like the Bellman-Ford) detect the presence of an inconsistency, they fail without providing any help to resolve the inconsistency. Recent research has developed strategies to exploit the *negative cycles* that appear in the STN's graph when an *inconsistency* occurs to provide an *explanation* of the conflict. The explanation of an *inconsistency* is found by identifying the edges and nodes involved in the *negative cycle*, and then mapping these edges and nodes to specific temporal constraints and activities using the higher-level domain model of the problem [156, 15, 21]. Once the conflict explanation has been

obtained, it is possible to design conflict resolution strategies that restore the STN to consistency.

Explaining the Conflict in Team Charlie's Schedule

Figure 2.15 shows Charlie's schedule at minute 16, and the edges in the STN's graph involved in the *negative cycle*. Following these edges in clockwise direction starting from the *time-zero* point, we can see that the sum of the edges' weights is

$$30 - 15 - 16 = -1$$

The absolute value of this sum (1) is the *magnitude* of the conflict.

Mapping these edges to the temporal constraints they represent, we obtain the *conflict set*: the set of constraints involved in the STN conflict. These constraints are:

1. $(ET\_Alpha_S -$ *time-zero* $) \in [0, 0]$: The constraint fixing the start of ET_Alpha at 0.

2. $(ET\_Alpha_F - ET\_Alpha_S) \in [16, 16]$: The constraint specifying the duration of ET_Alpha as 16 (updated from the original 10 after the unexpected delay).

3. $(RH\_Alpha_S - ET\_Alpha_F) \in [0, \infty]$: The enables constraint specifying that ET_Alpha must precede RH_Alpha.

4. $(RH\_Alpha_F - RH\_Alpha_S) \in [15, 15]$: The constraint specifying the expected duration of RH_Alpha as 15.

5. $(Alpha\_Attack_F - RH\_Alpha_F) \in [0, \infty]$: The structural constraint linking the finish point of RH_Alpha to the finish point of its parent task Alpha_Attack.

6. $(Mission_F - Alpha\_Attack_F) \in [0, \infty]$: The structural constraint linking the finish point of Alpha_Attack to the finish point of its parent task Mission.

7. $(Mission_F -$ *time-zero* $) \in [-\infty, 30]$: The constraint specifying the deadline of the Mission as 30.

The retrieval of the conflict set enables us to explain the cause of the underlying STN conflict by examining the constraints involved: The failure to subdue the terrorists by time 16 causes an update to the duration constraint of the corresponding

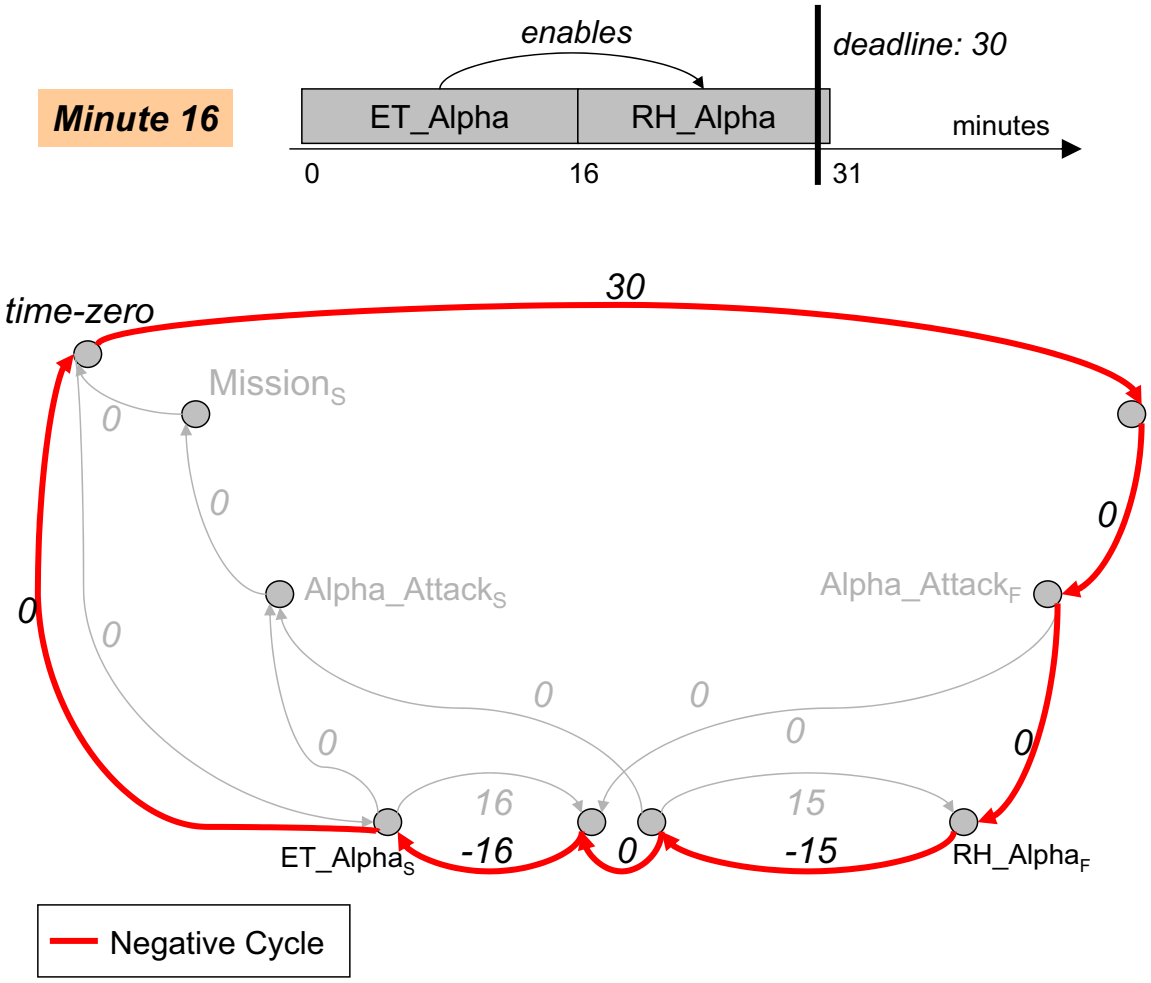

Figure 2.15: Team Charlie's Schedule with a Negative Cycle

method, ET_Alpha. The updated duration of 16 pushes (through the enables constraint) the start of the hostage rescue activity, RH_Alpha, to 16. Given that the expected duration for this method is 15, RH_Alpha can not expect to finish before time 31. However, through the structural links tying the finish point of RH_Alpha to its parent, Alpha_Attack, and Alpha_Attack to its parent Mission, the method RH_Alpha inherits the deadline for the mission at time 30. The delay in completing ET_Alpha in time means this deadline can no longer be met. Having explained the conflict, we can now turn to determining how to resolve it, thus re-enabling the STN of Team Charlie.

Resolving the Conflict in Team Charlie's Schedule

Resolving an STN conflict involves the removal or modification of constraints in the conflict set. While the inconsistency could be trivially resolved by removing constraints from the conflict set until the STN is conflict-free, this approach is impractical. Removing problem constraints (such as release or deadline constraints), or constraints representing past events (such as the duration of an executed activity) would lead to a meaningless schedule that cannot be executed. Instead, the explanation of the conflict can be used to reason about which constraints to select for removal or modification. In our example, Team Charlie is still battling the insurgents at time 16, and can't expect to finish the mission by the specified deadline. Out of the constraints in the conflict set, the constraints representing the start time of ET_Alpha and its current duration (constraints 1 and 2) cannot be modified as they represent past events. Reasoning over the remaining five constraints gives us three possible options for resolving the conflict:

1. Unscheduling RH_Alpha: Given that the hostages can no longer be rescued in time, the mission can be aborted. Unscheduling an activity can be accomplished by removing the constraints that sequence it to other methods on the timeline. In our case, the enables NLE is the constraint that sequences the two methods (ET_Alpha and RH_Alpha) on the timeline. Removing this constraint resolves the inconsistency. To see the effect of this operation, we can observe the updated STN edges in Figure 2.16. Removing the enables constraint breaks the chain of edges that formed the negative cycle, and the conflict disappears.

2. Reducing the expected duration of RH_Alpha: Team Charlie can opt to assume that after the terrorists are eliminated, it can achieve the hostage rescue activity faster than planned. Decreasing the duration of RH_Alpha by 1 minute (the magnitude of the conflict) removes the conflict from the STN. Observing the updated STN edges in Figure 2.17, we see that decreasing the duration of RH_Alpha causes the negative cycle edge representing this duration constraint to change from -15 to -14. The sum of the edges in the cycle is now 0, no longer negative.

3. Increasing the deadline of Mission: Alternatively, team Charlie can opt to take longer to complete the mission. Increasing the deadline of Mission by 1 minute removes the STN conflict. We can see the effect on the STN edges in Figure 2.18.

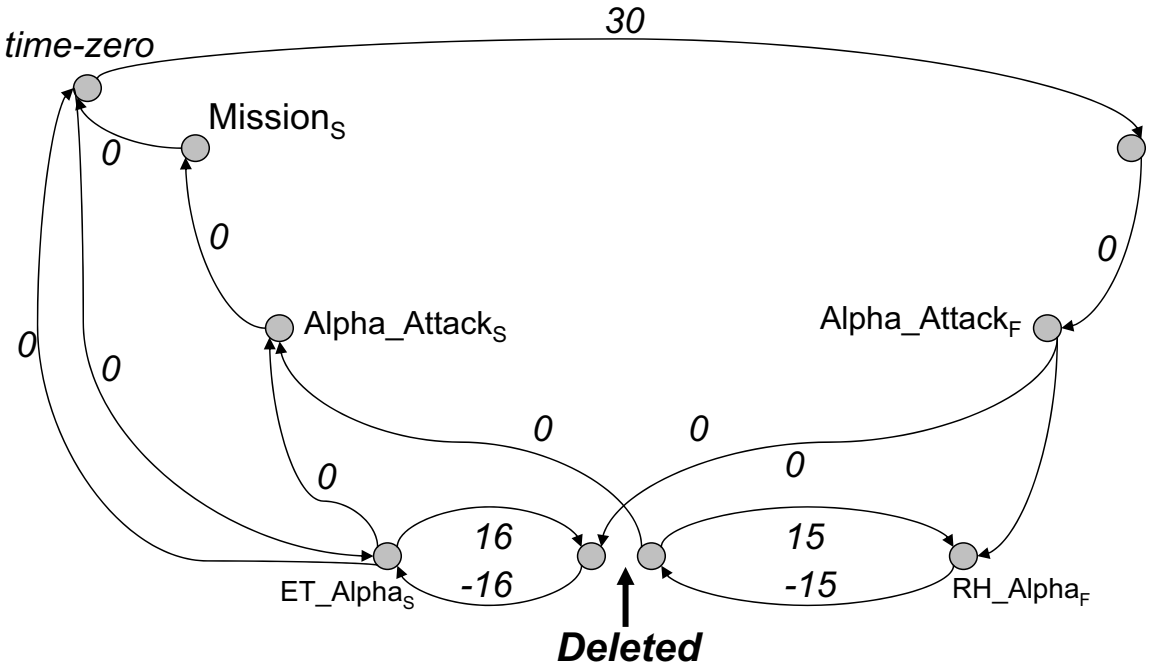

Figure 2.16: Team Charlie's STN after Aborting the Mission

Increasing the deadline of Mission causes the negative cycle edge representing the deadline constraint to change from 30 to 31. The sum of the cycle edges becomes 0.

cMatrix agents generally do not assume that the duration of a method (as specified by the plan) can be reduced [8], and deadlines in the Coordinators problems are hard (i.e. they cannot be modified). Consequently, options 2 and 3 are eliminated and the method RH_Alpha would be unscheduled. The following chapters will describe how we use similar *conflict explanation* techniques to (1) create a general framework that resolves execution-time conflicts in a distributed environment, and (2) analyze scheduling-time inconsistencies to find opportunities for improvement of the multi-agent schedule.

[8] One situation in which this can occur is if *facilitates* constraints are involved.

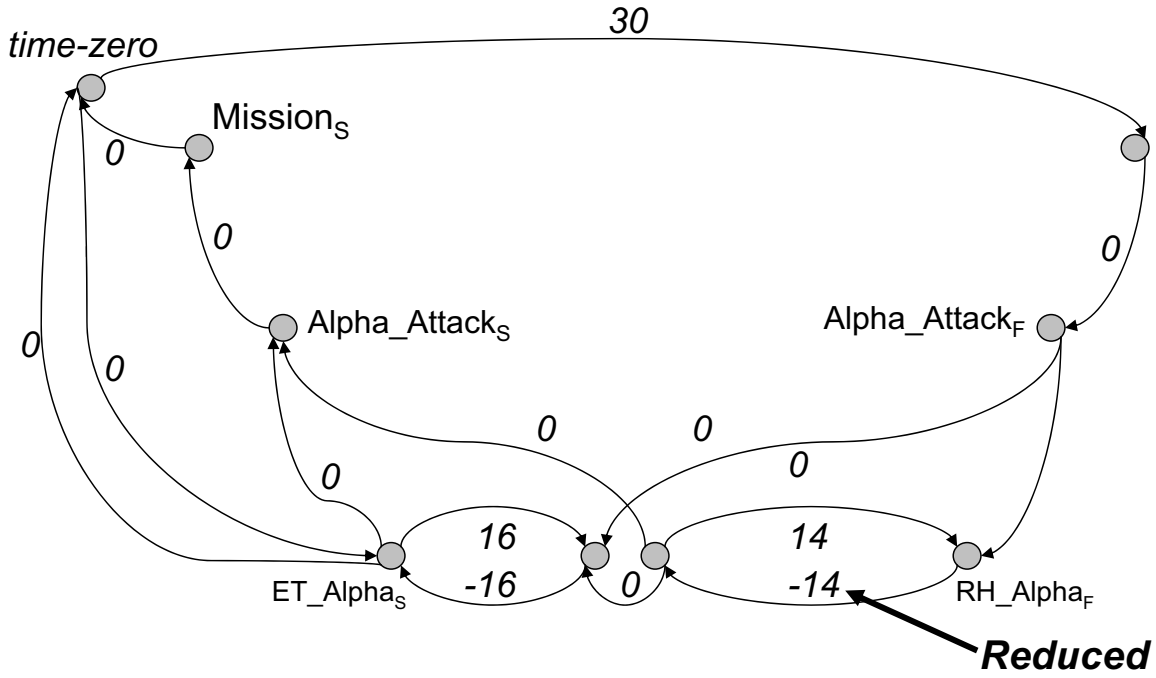

Figure 2.17: Team Charlie's STN after the Expected Duration of ET_Alpha is reduced

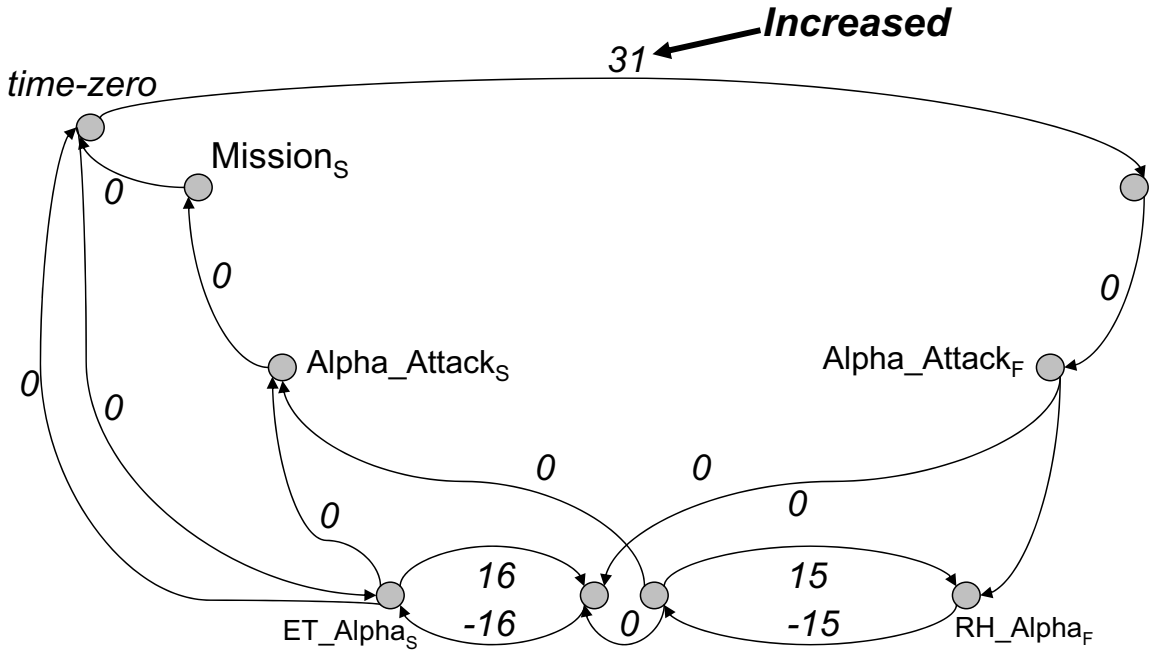

Figure 2.18: Team Charlie's STN after the Deadline is Increased

Chapter 3

Related Work

This chapter surveys previous research in areas related to the topics of this thesis. The first section of the chapter presents a set of related topics that provide a framework within which this thesis work fits in. The second part of the chapter examines previous research that has similarities to the specific contributions made by this thesis.

3.1 Thesis Context

In this section, we present topics that frame the context within which this thesis work fits in. We start by presenting the family of temporal networks, of which STNs are part. We then survey the field of continuous scheduling, which studies the integration of scheduling and schedule execution in a tight-loop. Third, we present the TAEMS language, on which C_TAEMS is based on, and how it has been applied to scheduling domains. Fourth, we survey *conflict explanation* strategies, a set of techniques to analyze inconsistencies in temporal network scheduling applications. Finally, we provide a brief outline of multi-agent coordination, touching on two broad coordination categories: implicit and explicit coordination.

3.1.1 Family of Temporal Networks

STNs are a special case of Temporal Constraint Satisfaction Networks (TCSNs). As explained in Section 2.1, the domains of STN temporal constraints are single contin-

uous intervals of the form:

$$(X_j - X_i) \in [l, u] \tag{3.1}$$

In contrast, general TCSNs allow temporal constraints to have domains with disjunctive intervals. Temporal constraints between time points in TCSNs are of the form:

$$(X_j - X_i) \in \vee_k [l_k, u_k] \tag{3.2}$$

The disjunctive intervals of TCSNs present a mechanism for representing choices of when to execute an activity. For example, for an activity A, we could specify that it should start between times t_1 and t_2, or between times t_3 and t_4. The increased flexibility comes at a price: While the temporal bounds of time points in STNs can be obtained in polynomial time, this is not the case for TCSNs [33]. Disjunctive Temporal Networks (DTNs) generalize TCSNs further, allowing the use of general disjunctive temporal constraints [166]. These constraints are of the form:

$$\vee_n ((X_j - X_i) \in \vee_k [l_k, u_k]) \tag{3.3}$$

The disjunctive constraints in DTNs can be used to represent choices of action. A disjunctive constraint can specify that either activity A should start between times t_1 and t_2, or activity B should start between times t_3 and t_4.

Numerous derivative works have expanded the core temporal constraint propagation functionality of temporal networks. The full family family of temporal networks we describe in this section is summarized in Table 3.1. Controllability criteria for temporal networks were developed to reason about uncertainty in activity durations. The basic recognition is that during the execution of a schedule, there is no firm control over how long activities will take, leading to uncertainty in their finish times. Time points in the temporal network are separated into controllable points (the start times of the activities), and uncontrollable points (the finish times of the activities). Schedulers that use controllability aim to create schedules that will remain feasible

	Uncertainty	**Preferences**	**Conditionals**	**Current Time**
STN [33]	STNU [180]	STNP [86]	CSTN [174]	ASTN [76]
TCSN [33]	TCSNU [179]	TCSNP [129]	CTCSN [174]	–
DTN [166]	DTNU [179]	DTNP [129]	CDTN [174]	–

Table 3.1: Family of Temporal Networks

despite these duration uncertainties. Initially, controllability was developed within the context of STNs, producing the Simple Temporal Networks with Uncertainty (STNUs) [180]. Later, the same techniques were adapted to TCSNs and DTNs, producing the Temporal Constraint Satisfaction Networks with Uncertainty (TCSNUs) and Disjunctive Temporal Networks with Uncertainty (DTNUs) [179]. Section 3.2.2 expands the discussion on controllability criteria, and explains how these criteria can be used to increase schedule robustness within the context of STNs.

Recent work has considered preference optimization as an extension to the temporal network's bounds propagation. In essence, these techniques transform the original constraint satisfaction problem of determining feasible bounds for time points into the constraint optimization problem of computing the optimal time within the feasible bounds for the time points. As with STNUs, these techniques were originally developed within the context of STNs, producing the Simple Temporal Networks with Preferences (STNPs) [86]. Later, these algorithms were extended to TCSNs and DTNs, producing the Temporal Constraint Satisfaction Networks with Preferences (TCSNPs) and Disjunctive Temporal Networks with Preferences (DTNPs) [129]. There have been additional efforts to combine STNUs and STNPs, producing the Simple Temporal Networks with Preferences and Uncertainty (STNPUs) [178], and to provide probability distributions for the duration uncertainty, producing the Simple Temporal Networks with Preferences and Probabilities (STNPPs) [115].

A framework to explicitly specify conditionals has also been developed within the temporal network family [174]. The technique is equally applicable to STNs, TCSNs and DTNs, producing the Conditional Simple Temporal Networks (CSTNs), the Conditional Temporal Constraint Satisfaction Networks (CTCSNs) and the Conditional Disjunctive Temporal Networks (CDTNs). The conditional statements allow the definition of logical propositions that are attached to constraints. Depending on the value of the proposition, different branches of the network are activated. The binary domain of the propositions restricts the usefulness of this technique to define conditional branches, as a maximum of two can be obtained per proposition. While a greater number of branches can be achieved through proposition combinations, the resulting increase in the number of necessary propositions increases the complexity of the algorithm.

As mentioned in section 2.1, a recent technique adds a *now* time point to an STN to keeps track of the current time, producing the Augmented Simple Temporal Network (ASTN) [76]. The ASTN representation is useful in continuous scheduling

domains, where awareness of the passage of time is important.

3.1.2 Continuous Scheduling

Classical scheduling research in Artificial Intelligence (AI) and Operations Research (OR) has treated scheduling as a static process that produces a solution to be executed at a later time. AI research has traditionally viewed scheduling as a special case of planning where the activities to be performed are known and all that is left is to order them [150], while OR research has largely focused on optimization processes to order a set of activities so that an objective function (e.g. the makespan of the schedule or the weighted tardiness of the activities [132, 18]) is maximized. However, viewing scheduling as a static, puzzle-like ordering task is an unfortunate over-simplification of the scheduling process. Under most practical conditions, scheduling must be a *continuous* process that is tied to the execution of the scheduled activities, and must react to changes in the environment by revising the existing schedule.

Many state-of-the-art scheduling systems follow the continuous scheduling paradigm, and have been applied to a wide variety of domains using different architectures and approaches. Although fixed-times schedules, which assign exact start and finish times to scheduled activities, are in general more brittle to execution-time deviations, they simplify some conflict analysis procedures since exact times for activity execution are available, and they have found application in some continuous scheduling systems. In manufacturing, the OPIS scheduler provides reactive mechanisms for maintaining production schedules in the presence of environmental dynamics [162]. Constraint analysis is used to prioritize and resolve problems as they arise, incrementally repairing existing solutions rather than formulating new schedules from scratch. In robotic search and rescue teams, the COCoA framework provides a Mixed Integer Linear Programming (MILP) approach to continuously allocate and schedule activities to a team of robots searching for victims in hazardous environments [89, 88]. In space exploration, the CLARAty framework incorporating the CASPER continuous planner has been used in the Deep Space Four probe. CASPER uses a hierarchy of planning layers, each responsible for a level of plan detail, going from long term coarse planning to short term detailed planning [24]. Subsequent works have used CASPER to create a continuous coordination framework for self-interested agents [26], and to implement a procedure for *live-task modification*, where a centralized continuous planner monitors the execution state of activities and adds resources to them if they are falling

behind schedule [146, 147].

Other researchers have leveraged flexible-times scheduling, which sequences scheduled activities but does not assign them exact start and finish times, to design continuous scheduling systems. A recently deployed high-speed manufacturing system uses constraint-based scheduling and state-space planning. The planner uses a variant of STRIPS extended with action durations and resources to solve a job-shop scheduling problem where several jobs arrive every second [140]. In an aerial reconnaissance domain, a recent extension to the Kirk Planner [87] has introduced a flexible-times planner that uses incremental repair to resolve execution time conflicts. The framework has been tested in a scenario involving aerial vehicles coordinating to take synchronized images [72]. The LAAS architecture, coupled with the IXTET-EXEC planner/scheduler, has been used to govern robotics systems. IXTET-EXEC maintains a temporally flexible plan for a robot, using this flexibility to accommodate failures during execution [93]. When the nominal plan becomes invalid, a plan repair procedure attempts to use the flexibility in the plan to incrementally reformulate the plan. This procedure may insert activities to produce needed resources, or insert sequencing links between activities in the plan. If the plan repair fails, full re-planning is attempted. Several flexible-times continuous scheduling systems have been developed by NASA to govern exploratory vehicles. The Remote Agent program has designed a continuous planner/scheduler for the Deep Space One NASA probe. The probe uses temporally flexible plans to coordinate the activities of its different components, and obtain scientific data without human supervision, operating over extended periods of time. Given the need for long-term autonomy, the program emphasizes schedule robustness [118]. More recently, the approaches used in the Remote Agent program were unified under the IDEA framework and the EUROPA continuous planner [116]. The IDEA framework has been used to control the Gromit exploration rover [46]. However, unlike the class of systems studied in this thesis, these continuous schedulers use centralized solvers to obtain solutions for their problems.

3.1.3 TAEMS Schedulers

TAEMS (Task Analysis, Environment Modeling and Simulation) [35, 71] was developed as a representation for Generalized Partial Global Planning (GPGP) [34], an extension of Partial Global Planning (PGP) [43]. PGP assumes that tasks are inherently decomposed, and that agents might not be aware of what tasks others are

planning to perform, or how these tasks relate to each other. Agents attempt to gain knowledge of the goals of the team, and construct a partial plan that can be integrated into the global plan to fulfill the team mission. PGP was developed in the context of a Distributed Vehicle Monitoring (DVM) application. GPGP generalized the coordination techniques of PGP, making them domain-independent. TAEMS was designed to serve as the modeling language underlying GPGP. TAEMS extends the HTN formulation [45] by providing semantics to describe complex interrelationships between activities, environmental uncertainty, models of activity utility and cost and resource consumption. C_TAEMS, the language of the Coordinators problems, inherits the basic TAEMS infrastructure and philosophy, while redefining some TAEMS constructs, adding new ones, and removing others that are not pertinent to the Coordinators domain.

The TAEMS framework has been used to develop various scheduling techniques, including continuous scheduling systems. TAEMS scheduling techniques have focused on satisficing, rather than optimal, solutions, given the high complexity of the TAEMS scheduling problem. Design-to-Time (DTT) was the first scheduler designed to use TAEMS [56]. The focus of DTT is anytime scheduling. Consider SUM tasks: As the children of the task accrue quality, the quality attained by the parent task increases. This property mirrors the anytime scheduling notion where performing an activity for longer durations leads to higher utility. Unlike other anytime scheduling frameworks [37, 98, 142, 185], where the utility of activities increases in a continuous fashion, the utility jumps of TAEMS tasks are discrete. Shifting the focus from anytime reasoning, the Design-to-Criteria (DTC) scheduler concentrated on issues such as balancing the quality of the taskgroup against the likelihood that the provided solution will succeed [183, 182]. DTC provides *knobs* for users of the scheduler to specify the criteria the scheduling process should emphasize. Knobs for emphasizing higher quality, lower duration, increased robustness, etc. are provided. Versions of DTT and DTC have been used in TAEMS-based applications for information gathering [96], hospital patient scheduling [36], intelligent home environments [69], distributed sensor allocation [70] and others.

The Coordinators program has produced several techniques that address the distributed scheduling and schedule management problem from different perspectives. Extending DTC, a Multi-Agent Design-to-Criteria (MADTC) scheduler was developed [149], but suffered from scalability problems due to the soft real-time nature of the DTC scheduler. A second system uses a multi-agent MDP (Markov Decision

Process) approach that *unrolls* the C_TAEMS structure into policies [119]. Given the size of the state space, it is infeasible to perform a complete exploration. The search is heuristically guided to unroll promising sections of the state space and construct a policy. The use of heuristics, while necessary, prevents the discovery of the optimal policy, the major incentive behind the use of MDPs. If the agents fall off policy during execution, a greedy scheduling approach is used. A third approach focuses on robustness, centering on the goal of avoiding mission failure [170]. The agent's scheduler is divided into a deliberative component and an opportunistic component. The opportunistic component is a reactive layer that responds to updates by making local modifications to the schedule. When changes cannot be managed locally, the deliberative component coordinates with other agents to find a new schedule. Schedules are chosen based not only on the quality they can accrue, but on their likelihood of success. Schedules whose success probability falls below an acceptable threshold are discarded. Coordination of agents' schedules is done through partial centralization. In some cases, a single agent may schedule for the whole team. Another recently developed system, the Criticality-Sensitive Coordination scheduler [101], makes use of a set of domain-specific metrics to guide the scheduling process. Metrics were designed to (1) calculate the likelihood that an activity can increase (or decrease) the quality of the taskgroup, (2) obtain the probability that an activity will attain positive quality, (3) determine a "target" quality that a scheduled activity is expected to achieve, and others. By propagating these metrics to the different team agents, the agents are able to form a cohesive picture of the state of the schedule. A set of managers add or remove activities from the schedule based on these metrics.

3.1.4 Explanation-based Constraint Programming

Classical constraint solvers do not provide any guidance to the user (whether human or automated system) when a solution is not found to a system of constraints. This omission is all the more regrettable given the fact that the constraint solvers have the knowledge of why a solution was not found, even though this knowledge is usually not explicitly kept. *Explanation-based* constraint programming introduces techniques that provide the user with the reasons that the solver was unable to obtain a solution to a constraint system [163, 85, 84, 32]. Two types of explanations have been presented: *contradiction* explanations, and *eliminating* explanations [145, 85, 84]. Contradiction explanations highlight a subset of constraints in the system that are in conflict with

one another. Eliminating explanations justify the removal of a value from the domain of a variable in the system. These latter type of explanations have been used in several constraint solvers. These solvers add *justifications* whenever a value is eliminated from a variable's domain, effectively producing a *trace* of the solver's reasoning [7, 139, 85].

Explanation-based constraint programming strategies have been applied to temporal network technology. Research in STN conflict explanation has focused on providing contradiction explanations. Contradictions, or inconsistencies, arising from a conflict between constraints, manifest as *negative cycles* in the STN's graph. Prior research has developed strategies to exploit the *negative cycles* in the STN's graph to provide an *explanation* of the conflict. As explained in section 2.5.2, the explanation of an *inconsistency* can be found by identifying the edges and nodes involved in the *negative cycle*, and then mapping these edges and nodes to specific temporal constraints and activities using the higher-level domain model of the problem. In the Comirem scheduler [156], a filtering step categorizes the constraints in the conflict and eliminates constraints that cannot be relaxed. The remaining constraints form the *explanation set*. This reduced set of constraints is used to formulate conflict resolution alternatives that are presented to a human user to choose from. A related technique used a "compression" routine to collect all activities in conflict and collapse them into higher-level goals [15]. A human planner is presented with these conflicting higher-level goal activities, and ways to resolve the conflict. A current project, the RoboCARE robotic assistant [21], uses an STN scheduler and conflict explanation to provide verbal cues to elderly or disabled humans to remind them of the tasks they need to perform.

3.1.5 Multi-agent Coordination

Cooperating agents need to to coordinate their behavior. Theories of teamwork, or how teams of agents work together to achieve their goals have been studied and formalized. Three requirements for teamwork have been identified: Mutual responsiveness, mutual commitment to the joint activity and mutual support [13]. A number of models for achieving teamwork have been proposed. These teamwork theories include SharedPlans [62], joint intention [27] and joint responsibility [80] More recently, a robust teamwork approach that scales to large agent teams has been presented [144].

A number of frameworks to facilitate agent teamwork have been developed based on these theories. The RETSINA framework [168] uses reasoning mechanisms based

on SharedPlans to help agents identify relevant pieces of information to be communicated, track interdependencies, detect conflicts and violations, formulate solutions to resolve conflicts and monitor team performance. TEAMCORE uses joint intentions to enable agents to reason about goal commitment and abandonment, information sharing and selective communication [171].

To enable communication between the agents in the team, the agents need mechanisms to transmit or request information. In general, we distinguish two types of mechanisms used for agent coordination: *implicit* and *explicit*. Implicit coordination occurs when agents are endowed with a set of simple rules, and complex group behavior *emerges* from their interactions. Explicit coordination on the other hand is purposeful. Agents intentionally start coordination sessions with other team members to find cooperation opportunities.

Implicit Coordination

Agents using implicit coordination are typically simple entities following a set of straight-forward and usually hard-coded rules of behavior. The agents self-organize, synchronizing their activities based on these rules of behavior that let them anticipate the actions of others and act accordingly [143]. This approach is also often called *emergent* behavior, as global coordinated behavior emerges from local interactions among the agents. The advantages of implicit coordination are its simplicity and inherent robustness: no complex coordination mechanisms are needed, and the behavior of the system is robust, as the agents make purely local decisions. Despite the simplicity of the approach, systems using emergent coordination have been used in a variety of multi-agent and multi-robot systems [3, 172, 167, 40, 124], and have demonstrated remarkable feats of coordinated behavior, such as the carrying of a box between multiple robots [17].

Not all domains, however, can rely solely on implicit coordination. In many domains, it is difficult or impossible to design the local behavior of agents in such a way that the desired global behavior is attained. Consequently, a more focused approach to explicit negotiation is required, although some form of implicit coordination is not precluded.

Explicit Coordination

In explicit coordination schemes, agents intentionally negotiate to achieve a common goal. Negotiation approaches between agents have been used in different applications to provide solutions to planning, allocation or scheduling problems. A popular approach uses market-based frameworks. These techniques are based on the contract-net protocol [152]. Agents submit bids based on the utility they gain, or cost they incur, for performing an action. The agent submitting the winning bid receives the contract to perform the action it placed a bid on for the team. Market-based frameworks provide an efficient way of coordinating teams of agents and have been used in a variety of systems [38, 196, 60, 57, 91, 78]. Many other explicit coordination approaches have also been used, such as organizational mechanisms (i.e. blackboards) [94], potential fields for obstacle avoidance[175], levels of "impatience and acquiescence" [126], persuasion [148], etc. The Generalized Partial Global Planning (GPGP) protocol, designed in conjunction with the TAEMS planning language, is one explicit coordination approach for multi-agent scheduling that provides mechanisms for explicitly coordinating the actions of agents [95].

3.2 Previous Research Related to Thesis Contributions

In this section, we focus on previous research that is closely related to the contributions made by this thesis. First, we present strategies that have been developed to manage temporal uncertainty. We describe both *reactive* approaches to recover from unexpected events during schedule execution, and *pro-active* techniques to prevent failures before they occur. Then, we present approaches that have been used to drive a search procedure based on conflicts in constraint-based systems.

3.2.1 Recovering from Temporal Inconsistencies

This section provides an overview of *reactive* techniques to overcome inconsistencies encountered during schedule execution due to environmental uncertainty. We first present Truth Maintenance Systems, which study the management of inconsistent beliefs in agent systems, and how agents can *react* to these inconsistencies and rec-

oncile them. Then, we describe how inter-agent negotiation has been used to resolve inconsistencies in scheduling applications using distributed STNs.

Truth Maintenance Systems

The management of the beliefs of an agent entity has been formally studied in truth maintenance systems (TMSs). TMSs propose that there should exist a module in the agent architecture that reasons about the consistency of the beliefs held by the agent [107]. Three different types of TMSs have been proposed: Justification-based TMSs (JTMSs), Assumption-based TMSs (ATMSs) and Logic-based TMSs (LTMSs). JTMSs use labels that are attached to beliefs and denote their state. A belief that is consistent is labeled as IN. A belief that is found inconsistent is labeled as OUT. OUT beliefs can be removed. ATMSs introduce the concept of *contexts*. Beliefs can be valid in one context, but not in others. LTMSs can use logic clauses to justify a belief [164].

It has been shown that all CSP problems can be translated into the logic propositions used by TMSs, enabling the use of TMS approaches in agents using constraint representations [29]. Given that temporal networks are a special form of CSPs, we can use the concepts developed in TMS research to maintain the beliefs encoded in the temporal network. TMSs have been previously used in multi-agent applications by expanding the set of labels associated with beliefs to INTERNAL, EXTERNAL and OUT. INTERNAL beliefs are those that are believed because they are locally justifiable. EXTERNAL beliefs are those that an agent believes because it was told their validity by another team member. Finally, OUT beliefs have the same semantics as for single-agent TMSs [73].

In an STN-based scheduling agent (such as the cMatrix agent described in section 2.3), the temporal beliefs of the agents are represented by temporal constraints in the STN. Temporal beliefs regarding local activities can be considered INTERNAL, beliefs regarding remote activities EXTERNAL, and inconsistent beliefs are OUT and retracted from the STN. Nonetheless, deciding which belief should be labeled as OUT can be difficult. A negative cycle in the STN indicates the presence of a set of conflicting temporal constraints. While the STN can be brought back to consistency by simply removing constraints from the conflicting set, discarding beliefs at random is not a practical solution. Discarding an INTERNAL belief as OUT corresponds to making a change to a local activity. EXTERNAL beliefs, though,

require extra reasoning. When two EXTERNAL beliefs are in conflict with each other, what status should they be assigned? One suggestion in the TMS literature is that agents can *agree to disagree*, and have inconsistencies across multi-agent beliefs [73]. Using this approach is reasonable in domains where the shared beliefs do not have a clear "owner". However, in domains where the authority to modify beliefs is assigned to specific agents, it seems unreasonable to allow the local agent to disagree with the information given to it by the "owner" of the EXTERNAL belief. While Bayesian Belief Networks [128] and adding possibilistic models to TMSs [42] can take into account uncertainty in beliefs, they require the availability probability distributions or possibility measures that describe the uncertainty in specific beliefs.

Using Coordination to Resolve Inconsistencies in Distributed STNs

Previous distributed scheduling applications that employ temporal network technology have developed coordination-based approaches to recover from the temporal inconsistencies that arise during schedule execution. One approach is to view the distributed temporal network as a single global network, where constraint propagation is performed in exactly the same manner as with a centralized temporal network, with the difference that when a constraint links two agents together, bounds are transmitted back and forth [19, 20, 187] between the agents. While simple, this approach only considers the transmission of local decisions to other agents. Coordinated actions where joint decisions have to be made involves centralization of the problem in a single agent.

A second, more sophisticated approach, relies on inter-agent negotiation to resolve temporal conflicts that link the activities of different agents. The Multi-Agent Planning System (MAPS) uses STNs to represent the schedules of a team of cooperating agents [63]. This system overcomes the difficulties of information inconsistencies by introducing coordinating agents that control the use of shared resources and prevent conflicts between the agents using them. The Multi-agent Planning Language (MAPL) framework instead assumes that inconsistencies in the states of different agents will be resolved through coordination sessions [14]. While logical, this approach leaves the STN in an inconsistent state while waiting for the resolving information, rendering it useless for the interval. Further, in a scenario with imperfect communication, an agent may never receive information that resolves the conflict, leaving the STN in an inconsistent state permanently.

3.2.2 Preventing Temporal Inconsistencies

Increasing the robustness of schedules to execution uncertainty has been an area of research since early scheduling applications. PERT, a technique that can be traced back to the 1950's, extended the Critical Path Model (CPM) approach to incorporate durational uncertainty reasoning, and it's still in use today [92]. A recent overview of how schedulers manage environmental change divided the approaches into three categories: Schedulers that use uncertainty models to reason about and prevent potential failure points, schedulers that attempt to maximize stability and flexibility of solutions given the direct correlation of these metrics to robustness, and self-scheduling systems that schedule activities on the fly as necessary to keep pace with execution [153].

In this section, we survey *pro-active* strategies that attempt to prevent temporal inconsistencies *before* they arise, frequently using a model of durational uncertainty for the plan activities. First, we briefly summarize Markov Decision Processes, a decision-theoretic framework that subsumes most *pro-active* uncertainty techniques. Then, we present contingency scheduling, a group of techniques that reason about potential schedule failures and take action to prevent them. Next, we outline temporal network technologies to prevent temporal inconsistencies during execution. Finally, we present stability and flexibility metrics to increase schedule robustness to failure, and self-scheduling systems, a popular strategy in multi-agent domains that schedules activities on an as-needed basis.

Markov Decision Processes

Markov Decision Processes (MDPs) and related techniques offer a sound decision-theoretic framework that directly or indirectly encompass many other uncertainty management techniques [10]. MDP-based techniques formulate *policies*. These policies specify what action the agent should take in response to any occurrence (coded as the agent finding itself in a *state*). These actions are chosen to maximize the agent's *reward* (or utility). These policies are optimal in the expected-value sense, accruing the maximum reward possible for the agent [135]. While, depending on the events that actually occur as the policies are executed, other actions than those prescribed by the policy may lead to better results, in the average case, following the policy will produce the highest reward for a utility-maximizing agent. MDP policies

are determined using powerful polynomial complexity algorithms. The most popular are *value iteration* and *policy iteration*. Unfortunately, this polynomial complexity is achieved in the number of states. Since the number of states in scheduling applications grows exponentially with the number of features, MDP techniques become problematic in large scheduling problems. Multi-agent problems typically introduce further complications, such as partial observability of the scenario and the need to distribute the reasoning among the agents. These additional characteristics further increase the complexity of using MDP-based approaches. In fact, no polynomial algorithms exist to solve Partially Observable MDPs (POMDPs) and Decentralized MDPs (DEC-MDPs) [61, 6]. To manage the MDP's complexity, some approaches use heuristics to channel the policy search to the most promising regions of the search space. These techniques give up optimality for the sake of computational tractability [44, 23, 121]. One such approach has been utilized for the Coordinators program [119]. It heuristically explores a promising section of the policy space, falling back to a greedy scheduling technique if the agents fall off policy.

Contingency Scheduling

Contingency scheduling [1] uses a given uncertainty model to locate weak points in a schedule and fortify it. Most techniques that reason over uncertainty models can be subsumed within the MDP framework (discussed in the previous section), and in general, it is possible to translate contingency scheduling approaches into MDP terminology. While some contingency scheduling approaches use MDP-based techniques directly as their underlying reasoning engine [30, 9], standard MDP techniques do not scale well to large-scale problems in scheduling domains, and finding the optimal policy becomes intractable.

Contingency scheduling approaches can be divided into two families:

1. *Conformant*: These techniques create "fail-safe" schedules that are robust to any uncertainty anticipated in the model. They assume *no observability* during execution, so the schedule must be able to take into account every eventuality. STN controllability techniques (described in the next section) can be defined as conformant, since they are designed to prevent any failures during execution. Other examples of conformant systems are Probaprob [123] and Conformant

[1] Usually, these approaches are considered part of contingency planning. We will however, use the contingency scheduling moniker to differentiate those relevant to the scheduling field.

Graphplan [151]. Conformant techniques are not well-suited to oversubscribed scheduling domains, where failure to schedule and execute some of the plan activities is acceptable, and the typical goal is to maximize a *utility* metric. In these domains, the use of strategies that design "fail-safe" schedules can result in significant missed opportunities for better performance.

2. *Conditional*: These techniques are heuristic, focusing on creating alternative schedules (sometimes called *contingencies*) to the most likely failure points. Some example conditional systems are Just-In-Case Scheduling [41, 31], the Mahinur system [122], the Phocus-HC planner [48], Tempastic [193, 192], ΔDur [105, 106], and Prottle [97].

   JIC Scheduling creates, prior to execution, a robust set of schedules by incorporating *contingencies* that can be used in case of failure [41]. The JIC procedure starts by using a utility-maximizing heuristic to produce an initial schedule. This heuristic selects the activities to schedule, and orders them so that all hard constraints are satisfied and as many preferences as possible respected. Once the initial schedule is produced, the JIC scheduler uses a two-step process to create a set of contingency schedules: First, the initial schedule is analyzed for potential failures during execution. The scheduled activities that are likely to fail are identified, based on an uncertainty model. Then, for each of these potential failure points, an alternative, or *contingent*, schedule that can be used if the failure occurs during execution is generated. This two-step process is repeated for all generated schedules (both the initial schedule and the contingencies) until schedule execution starts. A derivative JIC Scheduling approach focuses on determining the best places in the schedule to create contingency branches [31]. Similarly, the Mahinur system uses a value-based approach that iteratively identifies activities that contribute the most to the "value" of the plan, and designs contingencies that can be used in case they fail [122]. The Phocus-HC planner creates optimistic schedules that assume activities finish quickly, and creates contingency branches for "unsafe" points of the schedule [48]. Tempastic uses a Generalized Semi Markov Decision Process (GSMDP) to generate an initial plan, and iteratively improves the plan by reasoning about failure paths, with the objective of reducing the probability of failure [193, 192]. The ΔDur planning algorithms present a Concurrent Markov Decision Process (CoMDP) to reason about concurrent activities with uncertain durations with the goal of minimizing the makespan of the generated schedules [105, 106]. Prot-

tle uses a heuristic approach to search a probabilistic AND/OR tree and create plans for concurrent activities with uncertain durations [97].

Contingency scheduling techniques have also been examined within the context of the TAEMS framework and the DTC scheduler [138], using strategies similar to the value-based approach of the Mahinur system. These techniques leverage the DTC scheduler and statistical measures of schedule robustness to focus the creation of contingent schedules on activities whose failure would have the most detrimental effect. Full reliability is not guaranteed, instead, this technique attempts to attain an acceptable level of robustness. Our approach bears similarities to this work, but it performs its robustness analysis *continuously* as the schedule executes. A recent technique, *Probabilistic Plan Management (PPM)* [66], creates a similar *continuous* robust scheduling framework for a project scheduling problem. PPM defines an *uncertainty horizon* based on the premise that *pending* activities close to execution have a low probability of changing, while others further in the future have a high likelihood of changing. Pending activities within this uncertainty horizon are analyzed for their probability of successful execution. If this probability is below an acceptable threshold, rescheduling occurs. The approach we describe to monitor the probability of failure of "close-to-execution" activities as the schedule executes bears similarities to the PPM approach. However, PPM has been developed for a centralized domain where all activities have to be executed. Distributed and oversubscribed domains present additional challenges. Actions to reinforce the schedule may require a joint action between several agents. Choice between activities that provide different value to the schedule requires balancing the value of the activities that are in danger of failing against the cost of making the changes to the schedule needed to prevent the failure. An offline version of the PPM algorithm, Probabilistic Analysis Deterministic Schedules (PADS), has been implemented to increase the robustness of C_TAEMS schedules prior to the start of execution [67]. This strategy discovers the activities in the schedule most likely to fail, and reinforces those that are most valuable by scheduling redundancies.

Multi-processor scheduling research for real-time systems has also produced a number of strategies to increase the *fault-tolerance* of a schedule, and guarantee that failures encountered during schedule execution do not lead to system breakage. These techniques broadly fall into two categories: (1) Techniques that increase *resource redundancy* by keeping equivalent resources on *standby*, and (2) techniques that increase *time redundancy* by scheduling backup activities, or increasing idle time between ac-

tivities. In a typical application, fault-tolerance is be achieved by scheduling "replicas" of the activities on the different processors to achieve redundancy in case of failure [1, 58, 59, 64, 65, 104, 120, 136, 190].

Similar fault-tolerant techniques have been used in job shop scheduling applications: *Temporal protection* increases the duration of activities based on a *breakdown* uncertainty model of the resources they are using [54]. The extended duration can be used to cope with a potential breakdown. A similar approach increases the *slack* in the schedule in areas that contain activities prone to failing [28]. Another proposed strategy increases the *idle time* of a machine to hedge against potential failures [108].

Our approach to increasing schedule robustness shares affinities with fault-tolerant systems. Like these techniques, we use *redundant* activities to increase schedule robustness. However, in our oversubscribed domain, our goal is not to design a "fail-safe" schedule, but rather to maximize an objective *utility* criterion by executing the most "valuable" activities. The focus of fault tolerance techniques on guaranteeing robustness to all failures can lead to diminished performance in utility-maximizing oversubscribed domains.

STN Controllability

STN controllability criteria were designed to reason about models of activity duration uncertainty within an STN context [180, 181, 114]. Specifically, controllability works in conjunction with the STNU representation (see section 3.1.1). Different controllability criteria have been defined: A schedule is said to be *strongly controllable* if it is possible to compute an off-line solution that remains feasible regardless of uncertain events. *Dynamic controllability* requires that for any given time forward, a schedule that remains feasible in the face of any execution event can be found. Finally, *weak controllability* requires that for any given time forward, a schedule can be found that can accommodate any possible event, while maintaining feasibility. Approaches that use weak controllability perform a search similar to MDP techniques, where the objective is to ascertain that there is an action that can be taken in response to any event, and produce a feasible schedule. Unlike typical MDP searches, weak controllability is a satisfiability criterion. It only guarantees the existence of a solution, finding the optimal solution is beyond its scope. Strong controllability implies dynamic controllability. Dynamic controllability implies weak controllability. Mixtures of these controllability metrics have also been designed. Waypoint controllability combines

the properties of strong and weak controllability by guaranteeing that certain time points (the waypoints) remain feasible regardless of events [113]. An approach combining strong and dynamic controllability has also been proposed. This technique is very similar to waypoint controllability with the exception that regular time points (not waypoints) should be dynamically controllable, rather than weak controllable [165]. Once a controllable schedule has been produced, it can be reformulated for efficient dynamic execution [117, 184].

While these controllability approaches have been used with success in spacecraft and robotics domains [118, 165], they present challenges in large scheduling domains where the objective is to maximize an objective criterion. Strong controllability enforces the production of off-line guarantees for schedule feasibility under all possible execution events. This is an onerous demand that is hard to satisfy in real-world scenarios. Dynamic and weak controllability are less stringent than strong controllability, but their focus is also solely on schedule robustness to failure. None of these controllability techniques attempts to balance the schedule's robustness against maximizing other utility criteria. As such, they can lead to conservative scheduling policies that can result in significant missed opportunities for better performance.

Temporal Decoupling

A recent approach that avoids information inconsistency in distributed STNs is Temporal Decoupling (TD) [74]. The algorithm starts with an initial global plan (encoded using a global STN) that may involve temporal constraints among activities assigned to different agents. The TD algorithm *decouples* the network into independent sub-networks, each sub-network corresponding to the activities assigned to a single agent in the system. Each agent can then execute its activities without regard to the actions of other agents, as their activities no longer affect each other. The TD process is, in essence, a mechanism that makes each temporal constraint relating two activities of two different agents redundant by superseding it with two time constraints that are instead connected to the *time-zero* point. A drawback of the TD approach is that it imposes extra constraints on the various sub-networks to make links relating the activities of two different agents redundant. These additional constraints can cause the agents to fail to take advantage of subsequent opportunities for better performance, as valid solutions are eliminated. Nonetheless, the approach is promising, and it may be possible to use it to perform partial decoupling of the agents plans based on the

information known to them. A subsequent work has used this idea to partially decouple a global plan into several group plans, where each group could contain more than one agent. The activities in these group plans are then dispatched for execution in a centralized fashion [165]. A second multi-agent scheduling approach that uses TD is described in [11, 12]. Here, agents coordinate to create a schedule that is guaranteed to avoid conflicts over a limited future horizon, and then periodically extend these guarantees using a rolling horizon.

Schedule Stability and Flexibility

A large number of scheduling techniques have used schedule stability and flexibility metrics in lieu of developing uncertainty models to increase the robustness of the schedules. The connection between stability and robustness during dynamic schedule execution is intuitive. First, maintaining schedule stability implies restricting the amount of change to the schedule that execution-time events cause. Incremental scheduling techniques have been designed to implement this idea. Rather than reformulating a new schedule from scratch in response to a change, incremental scheduling approaches attempt to locate the section of the current schedule that was affected by the unforeseen event, and repair it, leaving the rest of the schedule intact. Incrementality (and by extension stability) provides three benefits to scheduling agents operating in dynamic environments: First, the restriction of the scope of the change reduces the complexity of the necessary rescheduling computation, increasing the responsiveness of the system. This feature is specially important when full scheduling is an expensive process (as it often is), or computational resources are scarce. Second, by limiting the amount of change, the amount of re-coordination and renegotiation is also restrained, cutting down on communication needs. Third, and most importantly, many scheduling domains lend themselves well to incremental approaches. As the scheduled activities are executed, commitments are made that are not easy to break, and there is an inherent cost in disrupting the scheduled sequence of events. Several continuous scheduling systems use incremental techniques to maintain schedule stability, and thus increase the robustness of solutions [162, 195, 154, 160, 53, 109, 141, 142, 137, 4, 197].

Similarly, a range of scheduling techniques have used schedule flexibility as a measure of schedule robustness. The key insight behind these approaches is that the larger the flexibility of the scheduled activities (the amount of time activities can

slide earlier or later while still meeting their constraints, or *slack*), the more robust the schedule is to failure. The slack can absorb many small execution-time deviations, and the sequence of scheduled activities does not require change. The ability to absorb changes without rescheduling naturally leads to an increase in schedule robustness. The larger the amount of slack of an activity, the larger the deviations it can absorb, and the more robust the schedule. Exploiting this notion of schedule flexibility naturally evolves into the development of flexible-times scheduling techniques. Flexible-times schedulers eschew committing to a fixed start and finish for scheduled activities. Instead, these approaches only commit to the *sequence* of actions. Rather than a fixed date to start and finish, scheduled activities are given feasible windows of time when they can start and finish, while respecting the constraints of the problem. Several approaches have been proposed to use slack based techniques to enhance the robustness of schedules [161, 155, 133, 134, 28]. Alternative approaches have also sought to increase schedule robustness by measuring the amount of connectivity between activities, that is, the number activities that are related by constraints [133, 2].

STN technologies provide a strong framework for developing both incremental and flexible-times scheduling techniques. The capability of STN constraint propagation to localize constraint violations and pinpoint the constraints in conflict provide strong support for incremental algorithms. With a few exceptions [159, 53, 158], very little work has been done in the development of incremental STN-based schedulers. STN-based schedulers are also inherently flexible-times, as the STN maintains the temporal bounds of activities, rather than fixing scheduled times for them.

Our work bears similarities to a slack-based technique [161] that was developed in the context of job-shop scheduling. In this approach, two components, a scheduler and a dispatcher, share the scheduling responsibility. The scheduler is in charge of long-term scheduling, while the dispatcher is capable of reacting to changes and making some modifications to activities close to execution. The dispatcher is in charge of three activities: The *executing* activity, the *on-deck* activity (scheduled to execute immediately after), and the *in-the-hole* activity (scheduled after the *on-deck* activity). Depending on execution events, the dispatcher can make modifications to these three activities to increase the schedule's flexibility (consequently making it more robust). We extend this idea and define an *on-deck set*: a subset of *pending* activities that immediately follow the *executing* activity. The *executing* activity, together with the *on-deck set* of activities form the set of *horizon* activities. Our robust scheduling

strategies analyze these *horizon* activities for potential failures and attempt to prevent them.

Self-Scheduling Systems

Self-scheduling systems fall in the tradition of reactive systems in AI. The premise under reactive systems is that the behavior of intelligent entities (such as agent programs) is intrinsically tied to the environment they operate in, and they arise or emerge in response to situations [188]. Self-scheduling systems are particularly well suited to scheduling applications where the activities to accomplish appear spontaneously, or models of the uncertainty in the environment are lacking. Scheduling is typically not performed in advance (although some pre-computation of forward schedules is not ruled out). Instead, decisions are made as needed for continued execution. These systems have found application in various domains [16, 100, 112, 25, 127], and a large number have used market-based approaches [125, 39, 57, 79, 60, 196]. However, in collaborative domains where activities are known a priori and uncertainty models are present, scheduling of activities can be crucial component in maximizing the performance of the agent team, and these dispatching techniques have less applicability. Nonetheless, self-scheduling systems are attractive because of their simplicity and inherent robustness. However, they cannot make any guarantees as to global performance [153].

3.2.3 Conflict-Driven Scheduling Search

Scheduling systems must be capable of generating solutions while meeting a broad set of temporal and resource constraints [153]. Managing these potentially conflicting demands require the ability the scheduler to resolve inconsistencies to produce a solution. In other words, inconsistencies during the scheduling process *narrow* the scheduling search, helping to converge to a solution. Focusing a search procedure by exploiting inconsistencies in the current solution is a traditional area of AI research. These approaches have used various names for these inconsistencies, such as *nogoods*, *elimination sets*, *exclusion relations* or *conflicts*. The key commonality behind these techniques is that information provided by the inconsistencies is used to refine a solution until it reaches consistency or the search procedure fails [189]. In the next sections, we examine distributed constraint-based approaches that resolve

conflicts through agent negotiation, and *conflict-driven* temporal network schedulers that analyze conflicts to drive the scheduling process.

Distributed Constraint-Based Systems

Constraint-based systems offer a versatile framework for solving a large number of AI problems [90]. Recently, several approaches have been developed to enable agents to solve constraint problems in a distributed manner. Constraint conflicts (such as conflicting temporal constraints in scheduling systems) are solved using agent negotiation. The two main categories of constraint problems are Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs). CSPs attempt to attain a solution to the problem that satisfies all the constraints. COPs generalize the CSP framework by adding a global objective function to optimize. Generally a COP solver does not attempt to satisfy all constraints, but rather the set of constraints that leads to the highest utility as specified by the optimization function. The Distributed Constraint Satisfaction Problem (DCSP) solves a CSP in a distributed manners. State of the art DCSP algorithms enable a team of agents to coordinate solutions asynchronously and concurrently [191]. Similarly, the Distributed Constraint Optimization Problem (DCOP) has received recent attention. Two DCOP approaches that find the "optimal" solution are ADOPT [111], and OptAPO [103]. ADOPT offers a fully decentralized strategy, while OptAPO uses cooperative mediation approach where a set of agents are designated as "mediators" and solve a subset of the problem for a number of agents. While ADOPT is better suited for domains where privacy of information is a concern, OptAPO can potentially produce faster solutions in domains where some sharing of information is permissible. Two recent "optimal" constraint solvers, DPOP [130] and PC-DPOP [131], present an analogous compromise: DPOP uses a fully decentralized algorithm, while PC-DPOP partially centralizes the information to produce a solution. Given the complexity of scheduling domains, optimal solutions are not always feasible. Both optimal and heuristic distributed constraint-based techniques have been developed and used with success in a variety of scheduling domains, such as the job-shop scheduling problem [169, 99], meeting scheduling [55, 110, 50, 102], sensor net scheduling [102, 194] and others [49, 83, 51, 158].

Conflict-Driven Temporal Network Schedulers

Recent research in temporal network-based scheduling has produced a number of scheduling systems that focus the search process by analyzing the *negative cycles* produced by the temporal network when a conflict arises. The Comirem system [160], a recent version of the MAPGEN planner [15] and the RoboCARE robotic assistant [21] use *conflict explanation* techniques to provide a human user with options to resolve scheduling problems based on *negative cycle* information. The Comirem scheduler operates in a military aircraft application, MAPGEN in a space robotics domain, and the RoboCARE project provides a robotic assistant for disabled people. While these three systems work in a mixed-initiative environment where a human is ultimately responsible for resolving the conflicts, a recent implementation of the Kirk Planner [87] introduces an application with automatic conflict resolution. This work develops the Temporal Plan Network (TPN), a DTN augmented with the Incremental Temporal Consistency (ITC) algorithm. The TPN and the planner work in a tight loop to find a feasible plan: The planner generates a (possibly infeasible) plan which is given to the TPN as input. The TPN translates the plan into temporal constraints that are encoded into the network. If the plan is infeasible (several conflicts may exist), the ITC algorithm is used to locate the conflicts. These conflicts are sent back to the planner which analyzes and resolves them before generating a new plan. The process iterates until plan feasibility is attained [72].

Chapter 4

Experimental Methodology

This chapter starts with a description of the real-time simulated execution environment that was used to run the multi-agent schedules for all the experiments to be described later this thesis. Then, we outline the format of the C_TAEMS plans that we used to test the advantages of the contributions presented in this thesis. Finally, the chapter finishes with an explanation of the format of the experimental results we obtained when comparing the different strategies tested by our experiments.

4.1 The Simulator

All of the experiments for this thesis were run on a cluster of 10 server machines [1]. The servers have dual-processor Xeons running at 3.06GHz, and 2Gb of RAM. They run Fedora Core Linux, with a 2.6.13 64-bit kernel.

The agents operate within a simulated environment built in-house that provides identical functionality to the MASS simulator used by the Coordinators program. All of our software (the simulator and agent) is written in Lisp, and compiled using Allegro Common Lisp (ACL) v8.1. We preferred building an in-house simulator for running problems in the 10-machine cluster for scalability reasons: The MASS requires a lot of processing power and memory, needing one computer per agent for optimal performance. The primary reason for these large system requirements is the use of Java and a separate Java Virtual Machine (JVM) per agent. Bypassing the Java layer enables our software to comfortably run many agents per computer.

[1] Not every server was used for some of the experimental runs.

The simulator has two functions: (1) It keeps track of the passage of time, transmitting the current time to the agents in discrete intervals, called *ticks*. For our experiments, the size of a tick was set to 1 second. (2) The simulator also keeps track of the trace of execution:

- It sums up the accumulated quality for the executing scenario.
- It validates all methods' start and finish times. Agents inform the simulator when they start a method, and the simulator responds with an acknowledgment message. The simulator determines methods' finish times: It selects the actual quality accrued by each method, and its actual duration from the probability distributions (these values can be also selected beforehand and passed on to the simulator).

4.2 Format of C_TAEMS Evaluation Scenarios

The C_TAEMS plans we used to test the contributions made by this thesis were either (1) custom generated or (2) provided by the Coordinators program. The custom generated scenarios were created with a scenario generator built in-house that provided us with increased control over the C_TAEMS plan structures than the scenario generator used by the Coordinators program. Generated scenarios were customized to test each of the three contributions presented by this thesis work. We also tested our techniques on C_TAEMS plans provided by the Coordinators program to validate the advantages of our techniques on an independently generated set of problems.

4.2.1 Structure of Generated Scenarios

All generated scenarios have a simple five-level activity hierarchy (loosely based on the activity hierarchies used by the Coordinators program).

1. First Level: The "scenario" (the taskgroup) is a single task with a SUM QAF.
2. Second Level: The "problems" have a SUM or SUMAND QAF and each has two or more children activities.

3. Third Level: The "windows" define a release and deadline for their descendant activities. They have a SUM QAF. Windows may be chained through *enables* or *facilitates* NLEs.

4. Fourth Level: The "cnodes" are tasks with a MAX or a SYNCSUM QAF. Like "windows", the cnodes may be chained through *enables* or *facilitates* NLEs. The children activities of cnodes are the methods. Each cnode can have up to three children methods: (1) A "primary" method that has the highest expected quality and longest expected duration, (2) a "fall-back" method (owned by the same agent that owns the primary method) with lower expected quality, but shorter expected duration, and (3) a "redundancy" method (owned by a different agent).

5. Fifth Level: This is the last level of activities, and consists of executable methods. Each method has three-point triangular quality and duration distributions. The three points consist of an expected value for the quality/duration, a second value that is lower, and a third value that is higher.

The generated scenarios varied in two key parameters:

1. Deadline Tightness: For a "window", w, the deadline tightness parameter, dt, is defined as the ratio between the size of its time window, $time\_window(w)$, divided by its duration, $duration(w)$.

$$time\_window(w) = deadline(w) - release(w) \tag{4.1}$$

$$dt = \frac{time\_window(w)}{duration(w)} \tag{4.2}$$

where $release(w)$ is the release time of w, and $deadline(w)$ is the deadline of w. $duration(w)$ is computed by analyzing the structure of the descendant activities of the "window" w. When the children cnodes are linked into "chains", the $duration(w)$ corresponds to finding the maximum duration chain.

The set of chains, ch_i, under w is defined as the powerset of all possible chain combinations. Single cnodes that are not part of a chain can be considered as a chain of 1. Then, $duration(w)$ for a window w with a set of K chains under it is given by:

$$duration(w) = max_i^K duration(ch_i) \tag{4.3}$$

The duration of a chain, *ch*, is found by adding the durations of the linked cnodes, and the "delays" of the NLEs linking them. A chain *ch* linking N cnodes, c_i, has a duration defined by:

$$duration(ch) = \sum_{i=1}^{N} duration(c_i) + \sum_{i=1}^{N-1} delay(nle(c_i, c_{i+1})) \tag{4.4}$$

where $nle(c_i, c_{i+1})$ is the NLE linking source cnode c_i and target cnode c_{i+1}. The "delay" of an NLE defines the number of ticks that must elapse between the source activity attaining quality, and the start of the target activity. For example, for a delay of 1, if the source cnode c_i attains positive quality at tick 5, target cnode c_{i+1} can start executing at tick 6. All NLEs in the generated C_TAEMS scenarios had a delay of 1. The duration of a cnode, c, is defined as the expected duration of its primary method child.

Our generated scenarios used three levels of deadline tightness: 1.0, 1.2 and 1.4. Deadline tightness increases as these values decrease: A deadline tightness of 1.0 implies that the time window is the same size as the duration of the window. A deadline tightness of 1.2 indicates that the time window is 20% larger in size, while a deadline tightness of 1.4 indicates a time window 40% larger.

2. Uncertainty: The duration distribution of methods in the generated problems contained three possible durations each: A minimum duration, the expected duration, and a maximum duration. The distribution has a triangular shape, with both the minimum and maximum duration values holding identical probabilities. The uncertainty metric represents the probability that the maximum (or minimum) duration of a method (as opposed to its expected duration) will occur. In our scenarios, we used four levels of uncertainty: 0.05, 0.125, 0.25 and 0.333.

4.2.2 Coordinators Year 2 Evaluation Scenarios

Two of the contributions presented by this thesis were tested on a subset of the C_TAEMS scenarios used during the Year 2 evaluation of the Coordinators program. This set of problems consisted of 56 C_TAEMS scenarios: 32 scenarios involving 25 agents, and 24 involving 50 agents. The structure of these scenarios is significantly more complex than that of the generated problems described in the previous section,

leading to a much greater number of inconsistencies during execution. The scenarios contain three forms of execution-time dynamics:

1. Uncertainty in the quality/duration obtained: Similarly to the generated scenarios, methods can accrue more/less quality than expected or take more/less time than expected.

2. New sets of activities added to the plan during execution: Agents can receive new activity hierarchies from the simulator while executing the original schedule. These new activities present the agents with the challenge of re-coordinating their schedules in the middle of execution to incorporate new activities into their schedules.

3. Modifications to existing activities: The simulator can inform the agents about changes to the initial assumptions on activities in the middle of execution. Two types of modifications were used: Changes to the release/deadline of an activity, and changes to the quality/duration distributions of methods.

The deadline tightness on these scenarios ranges from 1.2 to 1.3, and they include NLEs of every type (both hard and soft) linking activities at every level of the hierarchy.

4.3 Format of Experimental Results

Given that we are comparing heuristic techniques on scenarios for which it is not possible to obtain optimal scores, experimental results presented in this thesis are computed by comparing the values obtained by the different strategies against each other. For each scenario in a given set of problems, the result obtained by a strategy s was computed as the *ratio*, N_s^r, where

$$0 \leq N_s^r \leq 1$$

The ratio N_s^r is obtained using the equation

$$N_s^r = N_s / N_{max} \tag{4.5}$$

where N_s is the "raw" value achieved by strategy s on the scenario, and N_{max} is the maximum "raw" value achieved by any of the strategies under comparison on the same scenario. We report the results obtained by a strategy s on a set of scenarios as the average N_s^r on all the scenarios in the set.

The quality of the executed schedules is a common comparison metric in the experimental evaluation of all the strategies developed in this thesis. The quality comparison between the different strategies was obtained using quality ratios of the form shown in equation 4.5. The "raw" quality Q_s obtained by strategy s on a scenario is divided by Q_{max}, the maximum quality obtained by any of the tested strategies. Their ratio, Q_s^r, the quality ratio for strategy s, is given by the equation.

$$Q_s^r = Q_s/Q_{max} \tag{4.6}$$

Other comparison metrics used in this thesis test delays in information transmission, number of bytes transmitted during inter-agent coordination, and time elapsed during coordination sessions. All these comparison metrics also follow the format of equation 4.5.

Further details of the experimental analyses conducted in this thesis work are presented later in the relevant chapters of this document.

Chapter 5

Recovering from Inconsistencies

Managing agent beliefs in uncertain execution environments is an active area of research. Dynamics during execution that change the initial assumptions made by the agents can introduce inconsistencies between agent beliefs: (e.g. an agents receives a message with new or updated information that conflicts with a previously held belief.) When an agent is faced with two (or more) inconsistent pieces information, it has to make a decision to reconcile this conflict in a way that minimizes the deleterious effects of the inconsistency on its own work. In multi-agent scheduling domains, agents typically face inconsistencies between temporal beliefs, (e.g. the start/finish times of two inter-dependent activities are inconsistent with one another). The root cause of these dynamics is uncertainty about activity durations. This uncertainty results in some activities finishing later than expected, deviating from scheduling-time assumptions. When activities are inter-dependent, these deviations can have cascading repercussions on the rest of the agents' schedules. While STN-based scheduling agents produce *flexible-times* schedules capable of absorbing many unforeseen changes, flexibility is ultimately a function of the problem constraints and scheduled load of the agent, and some deviations can introduce an inconsistency that cannot be absorbed by the STN.

Approaches that deal with execution uncertainty are of two basic types: *pro-active* and *reactive*. Pro-active approaches make use of models of uncertainty to reason about possible future failure events and decide what action to take *before* these events occur. Reactive approaches, on the other hand, rely on recovery mechanisms to find a new solution *after* a failure event occurs. Pro-active and reactive approaches are not mutually exclusively. They can work in tandem to provide a combined framework

for uncertainty management. This chapter presents a reactive approach to recovering from inconsistencies within a STN-based multi-agent application; the following chapter will complement these techniques with pro-active strategies that minimize the number of inconsistencies that arise. The reactive techniques we have developed enable STN-based scheduling agents to reconcile conflicts between beliefs during execution. Such techniques have an added importance for applications that use STNs: An STN with an inconsistency is *broken*. STNs are a graph-based temporal CSP solver, and inconsistent temporal beliefs translate into inconsistent constraints in the STN. Like other constraint solvers, an STN with inconsistent constraints provides *no solution*. The inconsistency must be resolved for continued STN operation. In scheduling applications that rely on STNs to manage temporal information, an inconsistent STN prevents the scheduler from querying the STN for information about the temporal bounds of activities, paralyzing its ability to operate.

Our work expands on previous research in *conflict explanations* within the context of STN scheduling [156, 15, 21]. We analyze the *negative cycle* present in the STN graph when an inconsistency occurs to find the cause of the inconsistency, and decide on an action to resolve it. While previous research has focused on centralized domains, we address the additional complexities in resolving STN inconsistencies in distributed applications: A centralized solver has full knowledge about all activities in the plan, along with the ability to make scheduling decisions on any of these activities when a temporal conflict arises. Scheduling agents, on the other hand, may not have full knowledge of the plan, and they may lack the ability to make scheduling decisions for activities that other agents are executing. Previous work in STN-based scheduling agents has mostly concentrated on pro-active techniques that design conservative schedules capable of accommodating any execution-time deviations. The absence of reactive mechanisms to recover from inconsistencies during execution restricts the applicability of these approaches: An uncertainty model must exist, and it must be highly accurate if inconsistencies are to be avoided during execution. Further, the necessity of accounting for worse-case scenarios can lead to lost opportunities for better performance.

This chapter describes recovery techniques that enable agents to restore consistency to their STN when a conflict arises. We take into consideration the limited freedom of action of scheduling agents, and define two types of explanation-based conflict resolution techniques that an agent can use to restore its STN to consistency when a conflict arises: *local* and *non-local*. Local actions are akin to those

used by a centralized solver, involving scheduling decisions on *local* activities (those being executed by the agent). Non-local actions, on the other hand, involve making *assumptions* about *remote* activities (those being executed by other agents) to restore STN consistency, while waiting for updated information on these activities. The conflict resolution techniques we present concentrate on restoring a consistent STN state to enable the agent's scheduler to move forward. Further improvements to the schedule can be made subsequently once consistency has been achieved. We validate our conflict resolution techniques by comparing a team of agents that uses these resolution strategies against a second team of agents using a conservative approach that creates fail-safe schedules (using an uncertainty model). Our results show that an agent team using the explanation-based revision strategies we have developed can significantly outperform conservative approaches in domains where scheduling conservatively leads to lost opportunities for higher performance.

A second set of experiments tests the performance of a team of agents using a *delayed* conflict resolution approach. Agents using this delayed approach discriminate between conflicts caused by execution events (i.e. the start or finish of an activity), and non-execution events (related to pending activities). Rather than resolving non-execution related conflicts as soon as they arise, a small wait period is introduced to see if the conflict disappears on its own as new information arrives. Our results show that the introduction of this delay can significantly reduce the number of conflict resolution actions taken by the agents, thus avoiding unnecessary changes to the agents' schedules.

5.1 Restoring STN Consistency

Our approach to conflict resolution builds upon previous research in STN *temporal conflict explanations* [156, 15, 21]. As described in section 2.5.2, these techniques map the edges and nodes involved in an STN conflict (i.e. those involved in the *negative cycle*) to specific constraints and activities in the domain model. Once the constraints and activities in the conflict have been identified, the types of the constraints in the conflict (i.e. release/deadline constraints, duration constraints, etc.) are used to *explain* the situation that causes the conflict. While research in STN conflict explanations techniques has focused on centralized domains, multi-agent applications introduce additional complexities. Agents may not possess a full picture of the team

plan, preventing the conflict analysis from providing a complete explanation of the root of a conflict. Further, agents cannot take scheduling decisions for activities being executed by other agents: Agents can only make scheduling decisions involving the activities that they "own".

We have developed a three-step approach to resolving conflicts as they arise during schedule execution:

1. *Explain the Conflict*: In this first step, we use conflict explanation techniques to analyze the *negative cycle* in the STN and understand why the conflict arose. We *categorize* the constraints in the cycle to determine which can be modified.

2. *Remove the Conflict*: In this second step, we decide on an action to remove the *negative cycle* from the STN, and restore consistency. We have designed two types of conflict resolution actions: *Local* actions, involving scheduling actions on activities the agent "owns", and *non-local* actions, which involve making assumptions about about the continuing validity of information previously reported about remote activities by other agents. The goal of these conflict resolution actions is simply to restore STN consistency. Taking these actions typically involves a *loss of schedule quality*.

3. *Reschedule*: In this final step, the scheduler (see section 2.3.2) is invoked to attempt to recover (at least partially) the lost quality.

The initial two steps form the core of our conflict recovery approach. These steps restore consistency to the agent's STN, while ensuring that the scheduled local activities satisfy all known constraints that apply to them. Once STN consistency has been achieved, the third step is applied to improve the agent's schedule. The remainder of the section focuses on the first two steps. We start by providing a motivating example that extends the scenario outlined in Section 1.2. We will use this revised scenario to portray the types of conflicts that occur in agents' STNs during schedule execution. We then provide an overview of how the we categorize the constraints in a negative cycle, and decide on a conflict resolution action to restore consistency based on the type of conflict.

5.1.1 Motivating Scenario

In the original scenario described in section 1.2, teams Charlie and Bravo were tasked with rescuing hostages held in locations Alpha and Gamma. The teams had to synchronize the time of their attack, and then each team had to eliminate the terrorists at their assigned site, and finish by rescuing the hostages held there.

We now focus on the part of the mission that team Charlie has to accomplish and provide further details on the activities to be performed: Team Charlie is composed of three units, a command center Delta, an infantry squad Echo and an air-support unit Lima. Lima starts the mission by commencing the attack on Alpha, softening the enemy resistance for the ground team. When the enemy fire has been mostly subdued, Echo can move in to finish the remaining resistance. After eliminating the remaining terrorist threat, Echo finishes the mission by rescuing the hostages held.

Team Charlie's C_TAEMS Plan

The more detailed plan for team Charlie can be codified into a refined C_TAEMS plan. Team Charlie's updated plan, consisting of activities to be executed by Echo and Lima is shown in Figure 5.1. The higher-level tasks are managed by the commanding unit Delta. The key difference with the coarser plan shown in Figure 2.13 is that the method *ET_Alpha* (where team Charlie eliminates the terrorist threat) is now a higher-level task with a SUMAND QAF (indicating that all children have to be attained for the tasks to be successful). *ET_Alpha* divides into two children activities:

1. *S_Alpha*, a method assigned to Lima, represents the initial aerial softening of the enemy resistance.

2. *FT_Alpha*, a task with a SYNCSUM QAF, stands for the finishing up of the remaining resistance. *FT_Alpha* is itself composed of two children methods: (1) *G_Alpha*, assigned to Echo, represents the mopping up of the remaining threat by the ground team. (2) *A_Alpha*, assigned to Lima, describes the continuing aerial support provided to the ground team after Echo moves in.

An Unforeseen Delay During Execution

The STNs and schedules for Echo and Lima are shown in figures 5.2 and 5.3. Black solid arrows denote the *problem* constraints, such as the hierarchical structure of the

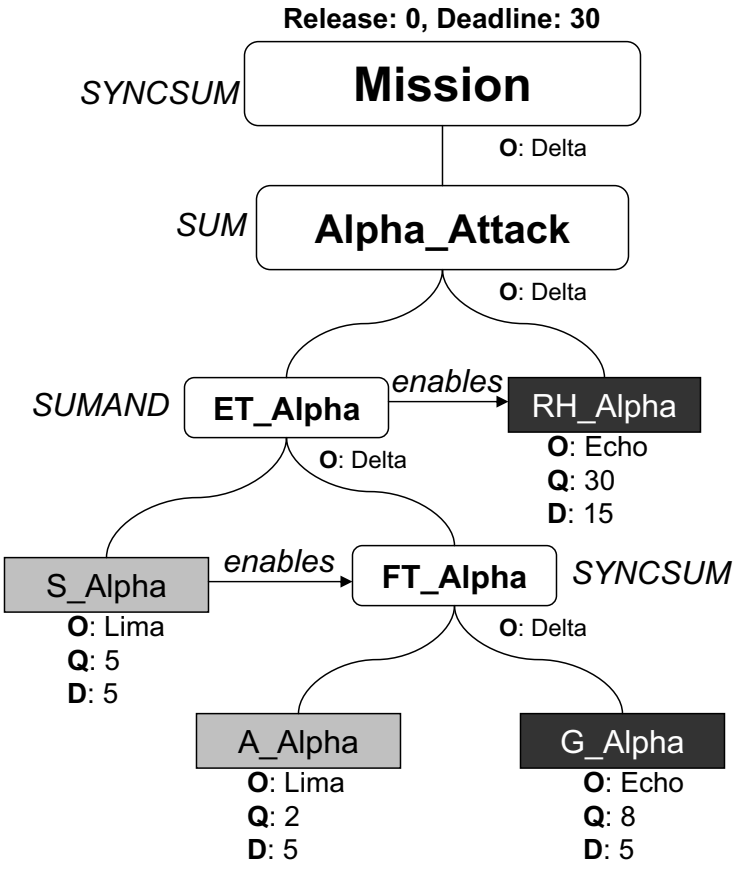

Figure 5.1: A more detailed C_TAEMS problem for Team Charlie.

activities, the release and deadline on the mission, and the *enables* constraints serializing the execution of the linked activities. Purple dashed lines show the *remote* constraints. These constraints enforce information about "remote" activities as related by their owner. This information consists of the duration of the remote activity, and its feasible time window (its EST/LFT). Green dotted lines show the *scheduler* constraints. These are sequencing constraints between scheduled local methods that the scheduler posts to order the activities on the agent's timeline.

The schedules of Echo and Lima are interrelated by the *enables* constraint linking *S_Alpha* and *FT_Alpha*. Even in this small problem, unexpected delays can introduce problems that the agents cannot easily resolve. Let's take Lima's part of the mission, for example: Lima encountered much stiffer resistance than anticipated. The terrorists at *Alpha* were equipped with shoulder-based anti-aircraft missiles. One of Lima's Apache helicopters was shot down, and the remaining helicopters were forced to with-

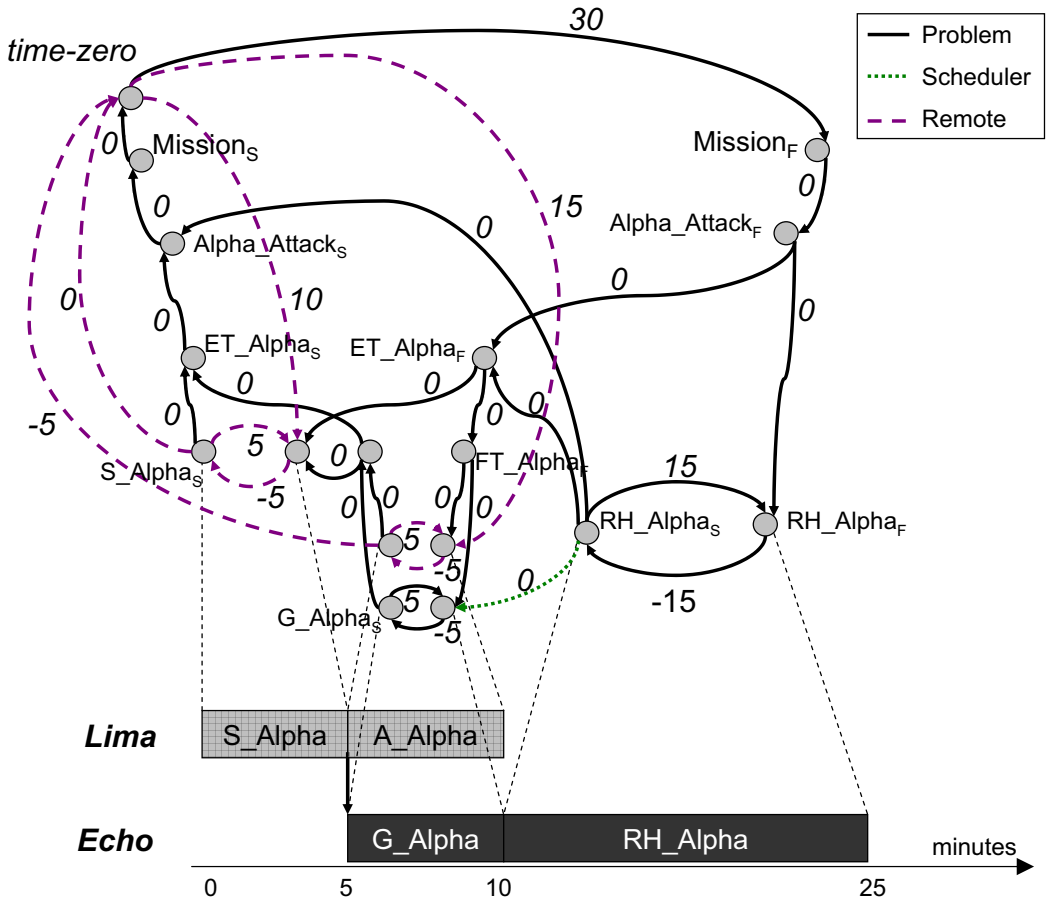

Figure 5.2: The STN and Schedule for Echo.

draw from close-range. While fighting the insurgents, their short-range weapons (such as their heavy machine-gun) became useless, forcing them to rely on their heavier and longer-range weaponry. Rather than subduing the resistance in 5 minutes as anticipated, Lima was only able to eliminate the enemy 11 minutes into the start of the mission. Given the unforeseen delay, Echo's activities have been pushed back to the point that the 30 minute deadline can no longer be met.

Lima's failure to complete its method *S_Alpha* on time introduces a temporal inconsistency in both agents' schedules. The negative cycle in Echo's STN is shown in Figure 5.4. The corresponding negative cycle in Lima's STN is shown in Figure 5.5. The inconsistency in Echo's STN involves its pending methods *G_Alpha* and *RH_Alpha*. Echo can restore consistency to its STN (and its schedule) by taking a "local" action, such as aborting the rest of the mission (given that it can no longer

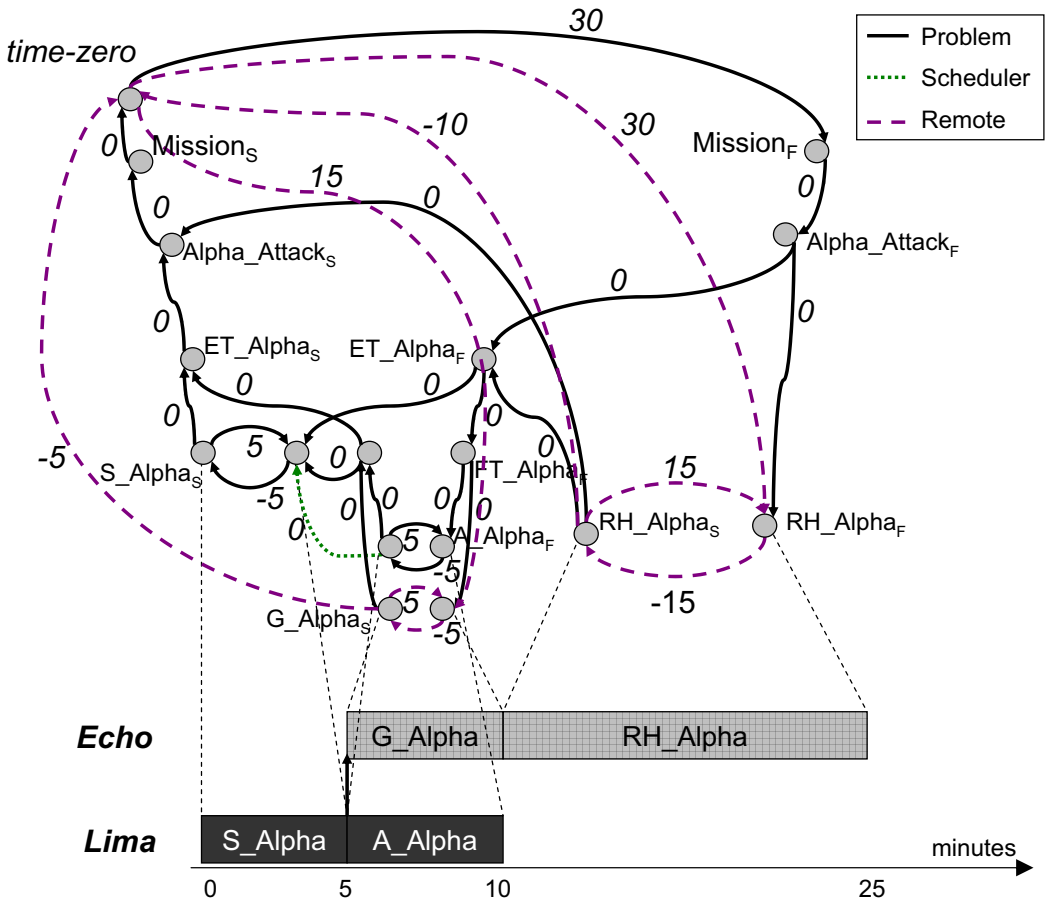

Figure 5.3: The STN and Schedule for Lima.

be finished in time).

The negative cycle in Lima's STN, on the other hand, does not involve any of Lima's pending methods, but rather Echo's method *G_Alpha*. Unlike Echo, Lima cannot take a "local" action by make scheduling decisions on this method. Instead, it has to wait to hear from Echo about what it decides to do. While it waits for this updated information, Lima's STN is inconsistent, and cannot be queried for information on activity bounds until its consistency is restored. If any scheduling decisions need to be made during this period, the STN has to be "repaired". Conflict resolution actions that do not depend on Lima's ability to make "local" scheduling decisions are necessary in this eventuality [1].

[1]While the example is contrived (Lima has no pressing scheduling decision to make, and could just wait to hear from Echo without taking any action to resolve the STN conflict), the purpose is to

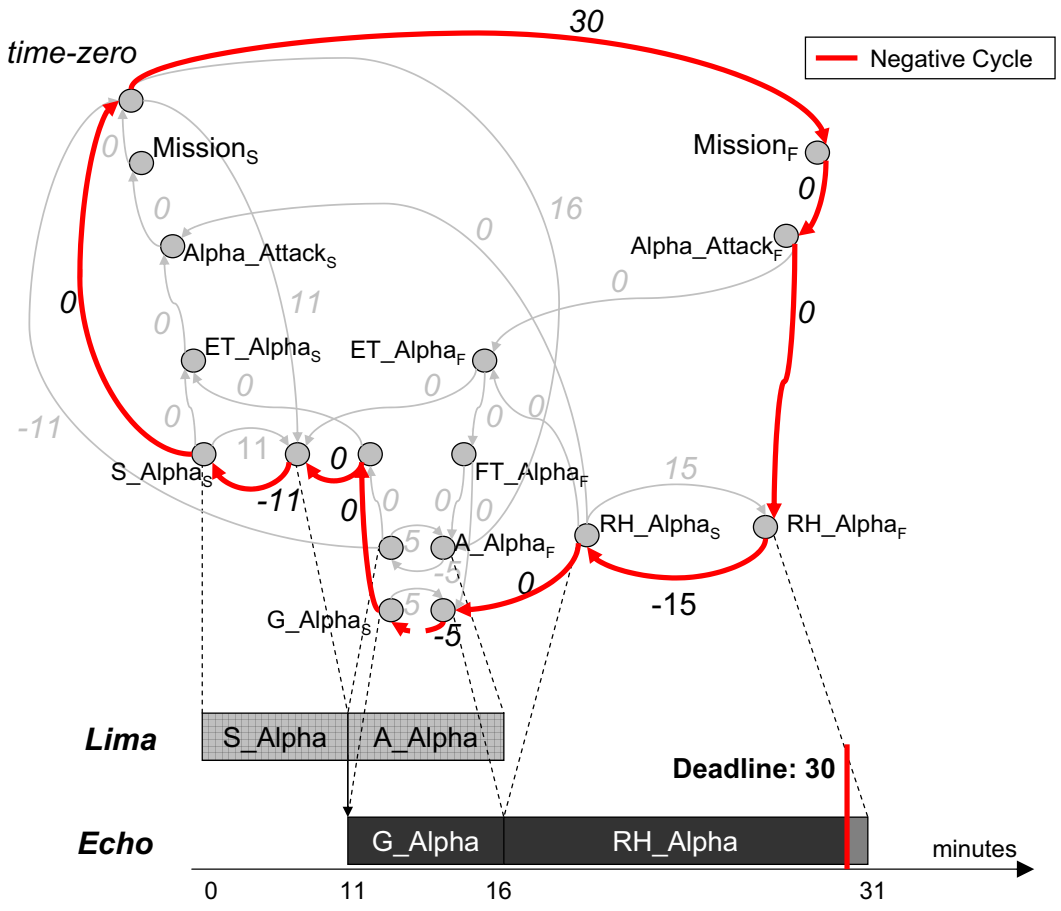

Figure 5.4: The Negative Cycle in Echo's STN.

5.1.2 Categorizing Constraints in the Conflict Set

When an STN conflict arises, the first step we take to explain the conflict is the mapping of the set of edges in the negative cycle to the corresponding set of domain level constraints. We refer to this set of domain level constraints as the *Conflict Set*. The constraints in any given Conflict Set can be partitioned into three non-overlapping subsets:

- *Retractable*: These constraints represent local scheduling decisions tied to events in the future (i.e. sequencing constraints that order the pending methods on the agent's timeline). The scheduler is free to retract or modify these constraints

illustrate a situation where the local agent (in this case Lima) cannot take a local action to resolve its STN conflict.

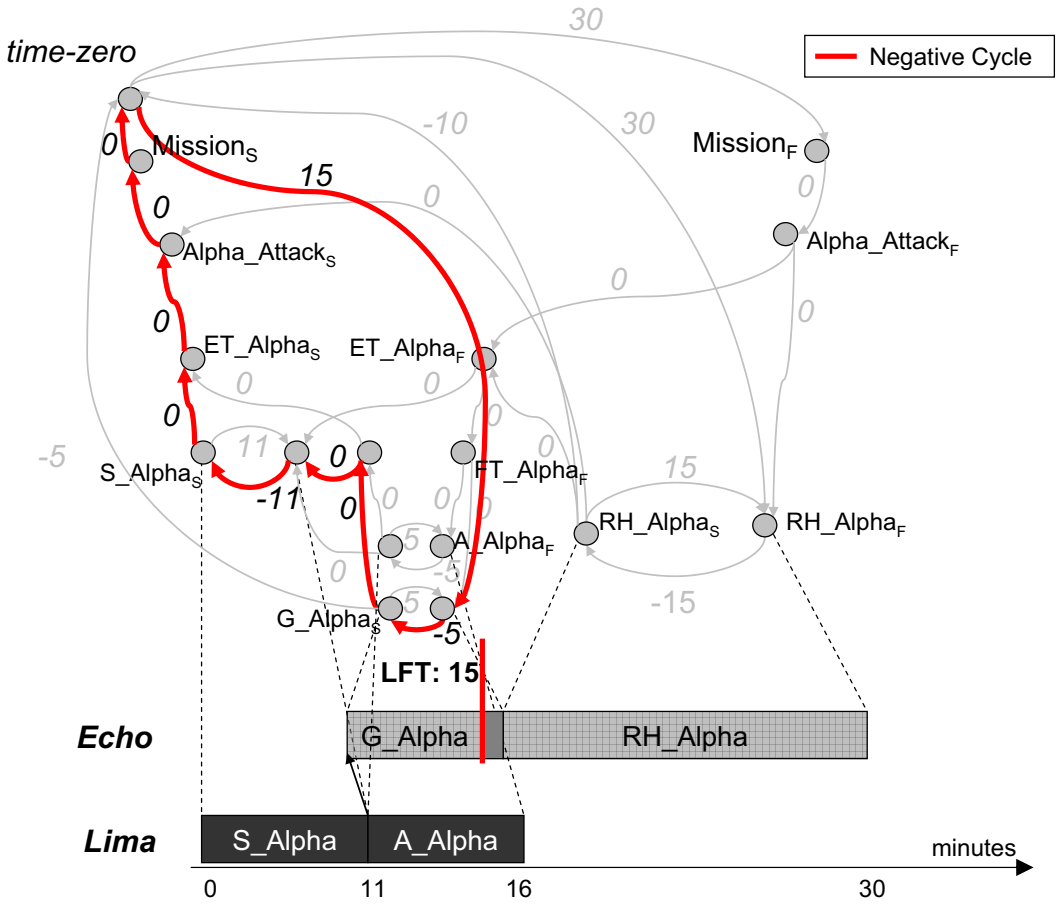

Figure 5.5: The Negative Cycle in Lima's STN.

as necessary to resolve a conflict. Retracting sequencing constraints results in the unscheduling of methods from the agent's schedule.

- *Suspendable*: These constraints include the problem constraints (i.e. release and deadline constraints, or structural links establishing the hierarchy of activities) tied to future events, and constraints enforcing reported information about pending remote activities. These constraints cannot be arbitrarily retracted, but their enforcement can be suspended when no retractable constraint can be removed or modified.

- *Immutable*: These constraints represent past events. The agent needs to resolve the conflict leaving these constraints untouched.

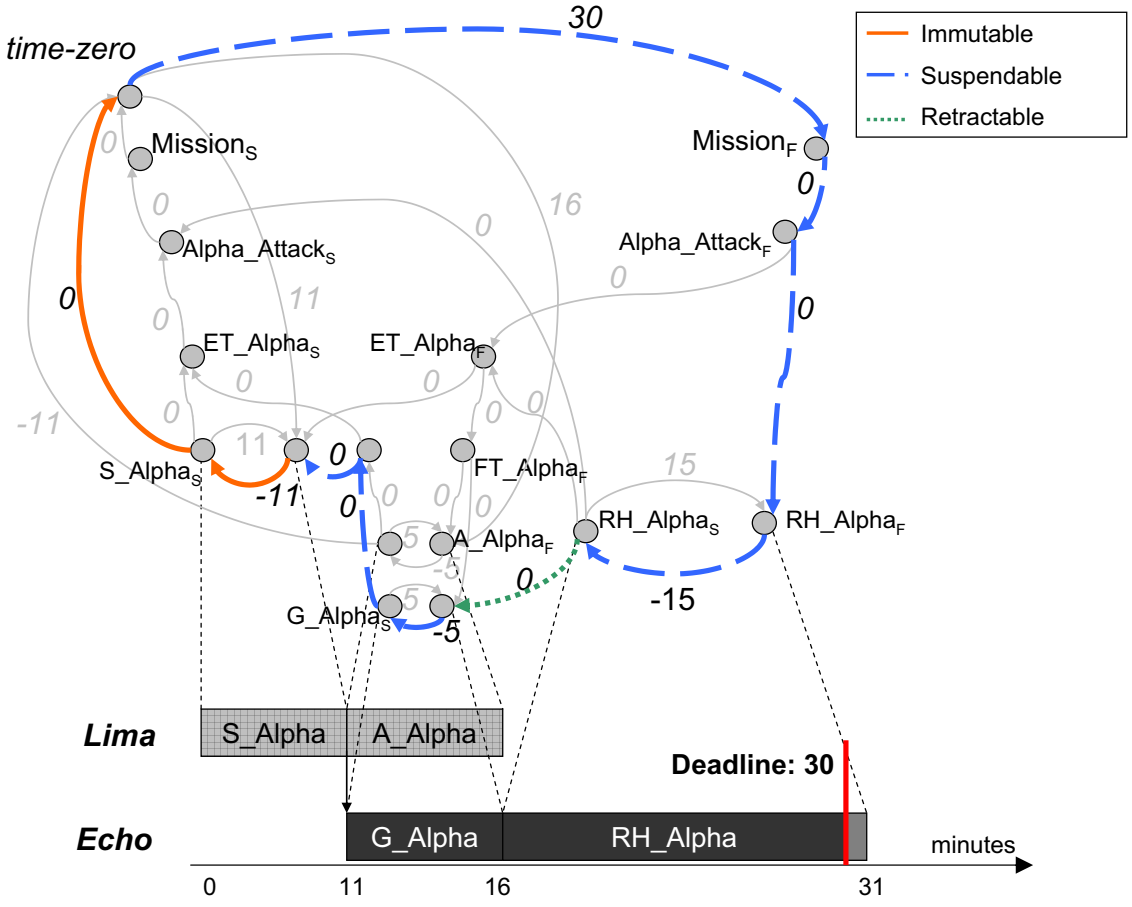

Figure 5.6: Categorizing the Constraints in the Conflict in Echo's STN.

The three sets of conflict constraints for the negative cycles in Echo and Lima's STNs are shown in figure 5.6 and 5.7 respectively. The STN edges not involved in the negative cycle are grayed out in the background. Solid orange lines represent the edges that correspond to *immutable* constraints. In this example, these constraints represent the information reported by Lima on the executed method *S_Alpha*. The edges correspond to the constraints enforcing the EST of *S_Alpha* and the duration of *S_Alpha*. Dashed blue lines represent the edges that correspond to *suspendable* constraints. These constraints include the problem constraints on pending activities (e.g. the durations of the pending methods), the hierarchical constraints linking them to higher-level activities, the NLEs between them, and the deadline constraint they inherit from the taskgroup. Finally, dotted green lines represent the edges that correspond to *retractable* constraints. These constraints are the sequencing constraints

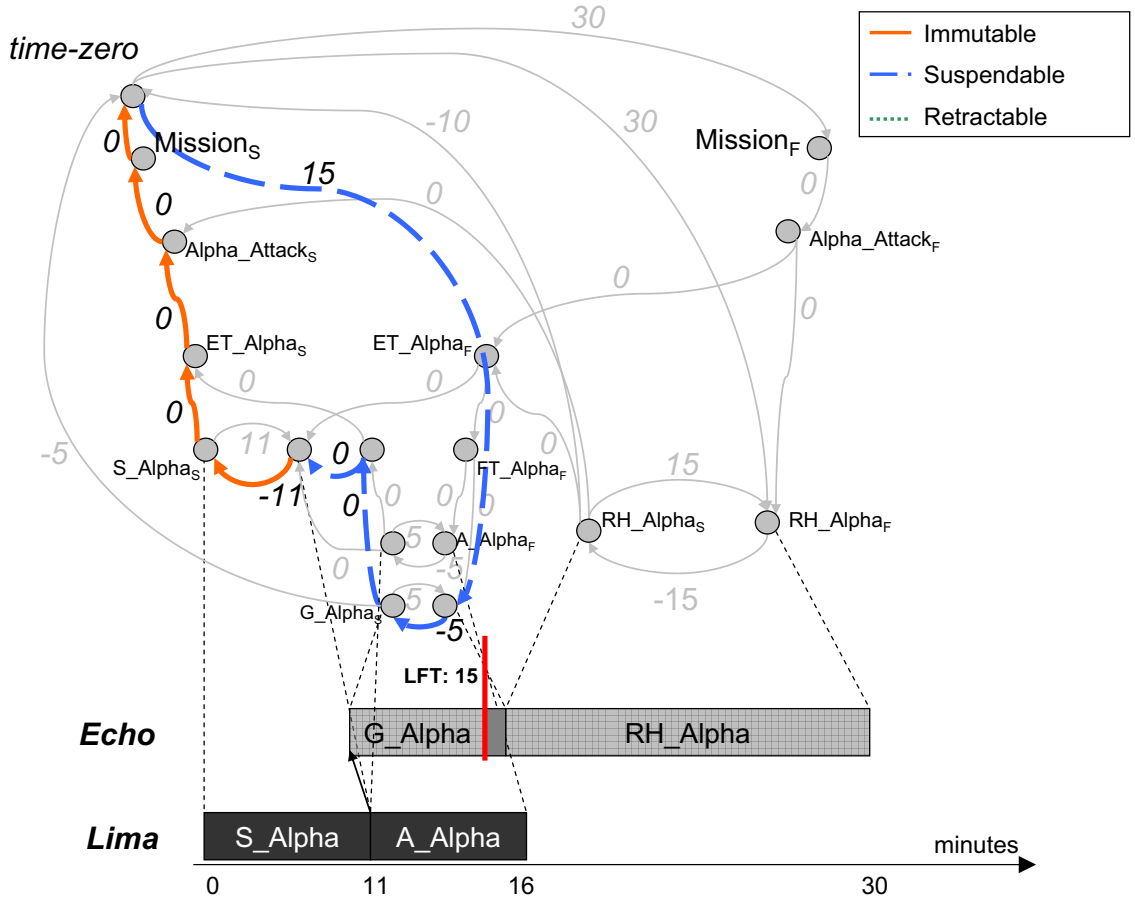

Figure 5.7: Categorizing the Constraints in the Conflict in Lima's STN.

linking the pending methods in the schedule.

The two sets of modifiable constraints (retractable and suspendable) encompass all constraints tied to future events. We call the constraints in these two sets the *Modifiable Conflict Set*. Any temporal inconsistency arising during execution involves at least one constraint in the *Modifiable Conflict Set*. Our conflict resolution actions remove inconsistencies that arise during execution by retracting or suspending some of the constraints in this set.

5.1.3 Conflict Resolution Actions

Our approach to conflict resolution defines a set of actions that can be used to restore STN consistency. The agent chooses one of these actions by examining the constraints

in the *Modifiable Conflict Set* to decide which action is most appropriate. Our conflict resolution actions are of two types: *Local* and *non-local*. The local actions involve steps an agent can take that entail changes to its "own" scheduled activities (over which the agent has full decision-making control). These types of actions are akin to those undertaken by a scheduler in a centralized domain, where the scheduler can make self-contained decisions regarding modifications to the schedule to resolve a temporal conflict. The "local" moniker, however, does not imply that these actions taken have no effect on other agents' schedules (e.g. the unscheduling of a local activity that is a prerequisite for another agent's activity will force this other agent to unschedule its dependent activity in turn). If an agent takes a "local" action that affects another agent's schedule, this second agent will in turn need to take appropriate action to modify its schedule as necessary, once it is informed of the change.

Taking non-local actions, ultimately involves making *assumptions* about "remote" activities. A non-local action *suspends* a "remote" constraint to restore consistency (e.g. an NLE involving a remote activity, or the constraint enforcing the duration of a remote activity). The *suspended* constraint is reinstated (possibly in a modified form) after receiving updated information from the pertinent remote agents.

For our problems of interest, we have identified four conflict resolutions actions that an agent can use when an inconsistency arises. Two of them are local actions: unscheduling of local activities and retracting a deadline constraint. The other two are non-local: suspending an NLE and suspending the reported duration of a remote activity. While some simple conflicts allow for only one possible resolution action, more complex conflicts can be resolved in different ways. In these situations, the selection of a conflict resolution action involves the use of a matching strategy that gives preference to one resolution action over another, based on the constraints in the conflict. This matching strategy is inherently application-dependent, as different domains present varying characteristics that make some actions more (or less) desirable than others. The cMatrix agent uses a simple case-based matching strategy that orders the four conflict resolution actions from most to least desirable, and selects the most desirable conflict resolution action that applies to the conflict.

The four conflict resolution actions used by the cMatrix agent are listed below in the order of preference in which they are used:

1. *Unscheduling of local activities* (Local)
2. *Suspending an NLE* (Non-local)

3. *Suspending the reported duration of a remote activity when enough local information exists to overwrite it* (Non-local)

4. *Retracting a deadline constraint* (Local)

The characteristics of the Coordinators domain are the main determinant factor behind this preference order. The first action, unscheduling of a local activity, is the preferred conflict resolution action, when applicable. This action does not involve consultations with any other agent, or making any assumptions. The fourth and last action, retracting of a deadline constraint, although local, is the least desirable, because deadlines in the Coordinators domain are "hard". An activity that violates its deadline does not accrue quality. Therefore, this is an extreme action that is only used when there is no other alternative to resolving the inconsistency. The second and third actions are both non-local, implying that assumptions about "remote" activities are needed. Suspending an NLE is given higher preference because, while it affects a "remote" activity (i.e. a temporal constraint that affects this activity is removed), it does not directly alter information received from the "owner" of the activity (i.e. its duration). The next sections describe each of these four conflict resolution actions in greater detail.

Unscheduling of local activities

Certain conflicts can be resolved by the simple unscheduling of one or more methods from the agent's timeline. These conflicts are characterized by the presence of one or more duration constraints of local *pending* activities in the Modifiable Conflict Set, and are of two types: (1) Conflicts involving an NLE that targets a local pending method, and (2) conflicts where a local pending method is in conflict with its release or deadline. The first type of conflict indicates that the NLE targeting the local method can no longer be enforced. Removing this method from the schedule resolves the conflict [2]. The second type of conflict indicates that a local pending method is not able to meet its release and/or deadline constraints. Typically this happens when the feasible time window of the method is compressed during schedule execution (e.g. a deadline is made tighter). The conflict can be removed by unscheduling the activity.

[2]Of course, the scheduler may later attempt to reschedule the method so that the NLE is respected; again our concern here is restoring a consistent state so that the agent's scheduler can move forward.

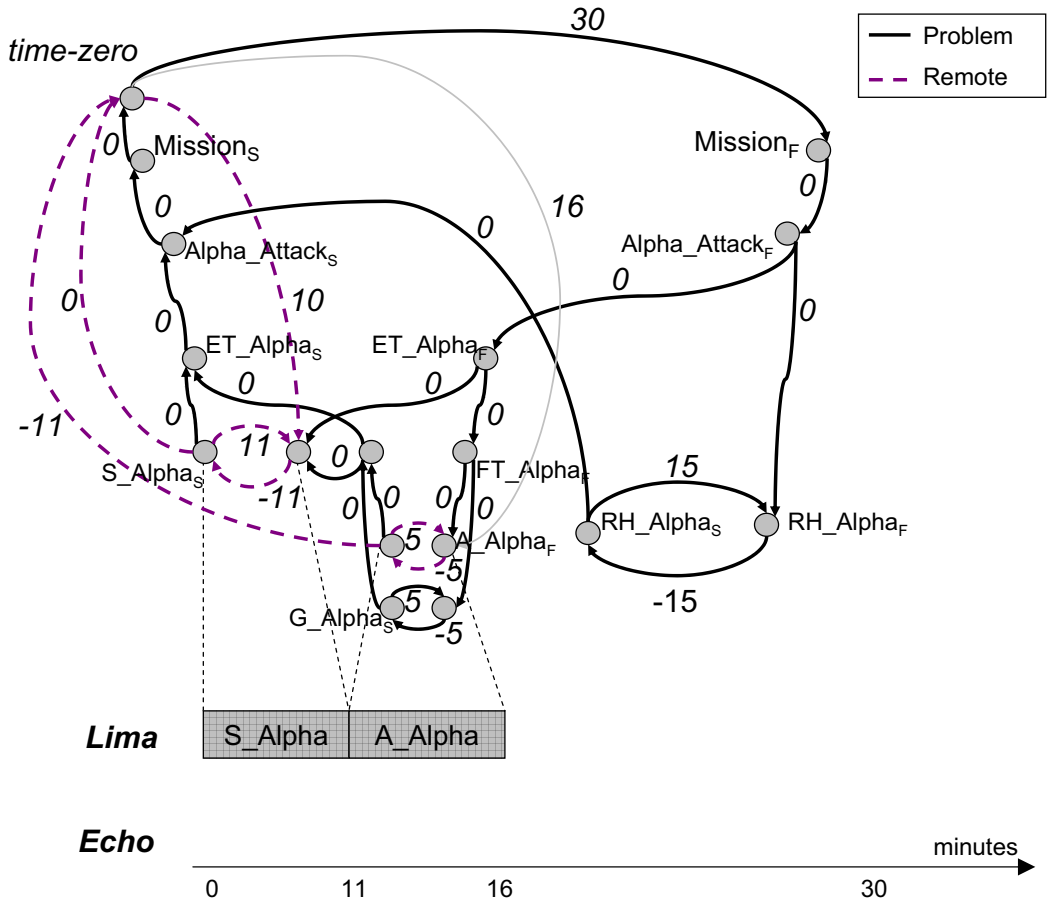

Figure 5.8: Action 1: Unscheduling Local Activities

The inconsistency in Echo's schedule (shown in Figure 5.6) is an example of the first type of conflict. After receiving notification from Lima that the completion of method *S_Alpha* has been delayed, Echo realizes that the enables NLE between *S_Alpha* and *FT_Alpha* can no longer be respected. Echo can resolve this conflict by aborting its part of the mission (unscheduling *G_Alpha* and *RH_Alpha*). The resulting conflict-free STN is shown in Figure 5.8 [3].

[3]Besides retracting the sequencing constraints that serialize the methods on the timeline, the unscheduling process also deactivates any NLE that targets the newly unscheduled methods to avoid spurious negative cycles involving these constraints.

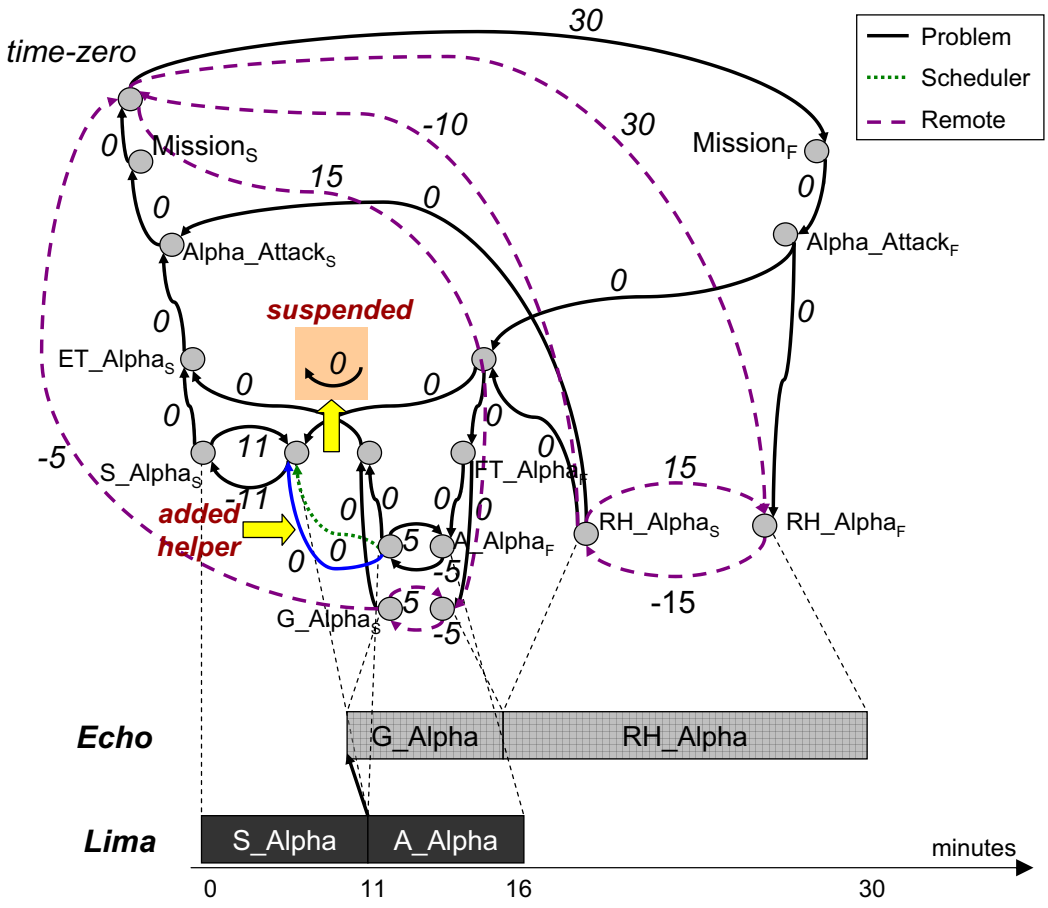

Figure 5.9: Action 2: Suspending an NLE - The Conflict-Free STN

Suspending an NLE

These types of conflicts are characterized by the presence of an NLE targeting a remote activity in the Modifiable Conflict Set. The local agent is unable to unschedule this activity to resolve the conflict, so it chooses the next best option: It *suspends* the enforcement of the NLE by temporarily retracting it from the STN, keeping it aside for future reinsertion when updated information on the remote target activity is received that resolves the conflict (e.g. when the agent hears that the start time of the remote target activity has been moved forward in time).

The inconsistency in Lima's schedule (shown in Figure 5.7) is an example of such a situation. After notifying Echo that it has been unable to complete *S_Alpha* in time, Lima (like Echo) realizes that the *enables* constraint between *S_Alpha* and *FT_Alpha*

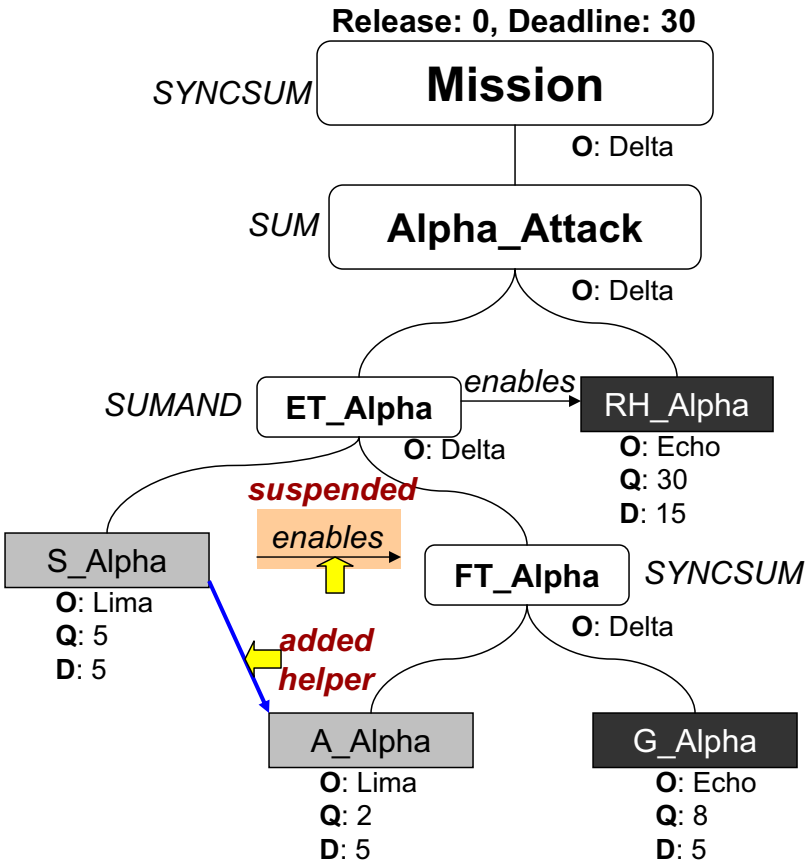

Figure 5.10: Action 2: Suspending an NLE - The modified C_TAEMS Plan

can no longer be respected. However, Lima is unable to unschedule Echo's method *G_Alpha*, and has to wait for Echo's updated information. In the meantime, to restore consistency to its STN, and allow its scheduler to move forward, Lima *suspends* the enables NLE between *S_Alpha* and *FT_Alpha*. This temporary retraction restores STN consistency. The STN with the suspended NLE constraint is shown in Figure 5.9. The suspended NLE can be reinserted once Lima receives the updated information on Echo's activities.

Suspending an NLE can have extended repercussions on the feasibility of the local schedule. If the *remote* target activity of the NLE is a task, then all descendant activities "inherit" this NLE, and its suspension affects their bounds. To ensure that *local* descendant activities continue to respect the suspended NLE, new *helper* NLEs are created between the source activity and all local methods that are descendants of the remote target task. Any local methods that are inconsistent with enforcing

the NLE are unscheduled. In our example, when Lima *suspends* the NLE between *S_Alpha* and *FT_Alpha*, its pending method *A_Alpha* (which inherits the NLE from its parent *FT_Alpha* is no longer bound by this interrelationship. To avoid a potentially invalid schedule, Lima instantiates a helper enables NLE directly between the delayed method *S_Alpha* and *A_Alpha*. The helper enables ensures that Lima's scheduler continues to be aware of the relationship between these two methods [4], and can be removed when updated information allows the original NLE to be reactivated. The modified C_TAEMS problem with the original NLE between *S_Alpha* and *FT_Alpha* suspended, and the helper NLE between *S_Alpha* and *A_Alpha* in its place is shown in Figure 5.10. Figure 5.9 shows the inserted helper in the STN.

Suspending the reported duration of a remote activity when enough local information exists to overwrite it

When no suspendable NLE is found in the Modifiable Conflict Set, yet the Modifiable Conflict Set includes the duration of a remote activity, it can be inferred that the reported duration for this remote activity is in conflict with the time window of an ancestor task. Since the activity is remote, the agent cannot unschedule it to restore feasibility. However, since the duration of the activity is infeasible, the agent can deduce that its reported duration is no longer valid. The agent thus *suspends* the duration of the activity by reducing it to fit within the time window of its ancestor task, while waiting for updated information from its owner.

We illustrate this conflict resolution action by further developing our hostage rescue scenario: Assume the commanding unit Delta has assessed the new situation (after Lima's unexpected delay), and has reached the conclusion that the level of damage inflicted by Lima on the insurgents at Alpha (due to the longer time of bombardment and higher use of heavier weapons) is higher than initially planned at the end of *S_Alpha*, and the ground team Echo is likely to encounter less resistance when it goes in. To achieve the mission within the 30 minutes time window, Delta updates the original plan, and informs Echo that its method *G_Alpha* is likely to last 3 minutes (instead of the original 5). The parent task *FT_Alpha* is given a hard deadline at minute 15 (to leave enough time to accomplish the rescuing step *RH_Alpha*). Echo incorporates these new instructions, revising its STN by posting the new and updated

[4] In our example, the schedule feasibility is not endangered, given the existing sequencing constraint linking the two methods. However, this situation is not generally the case.

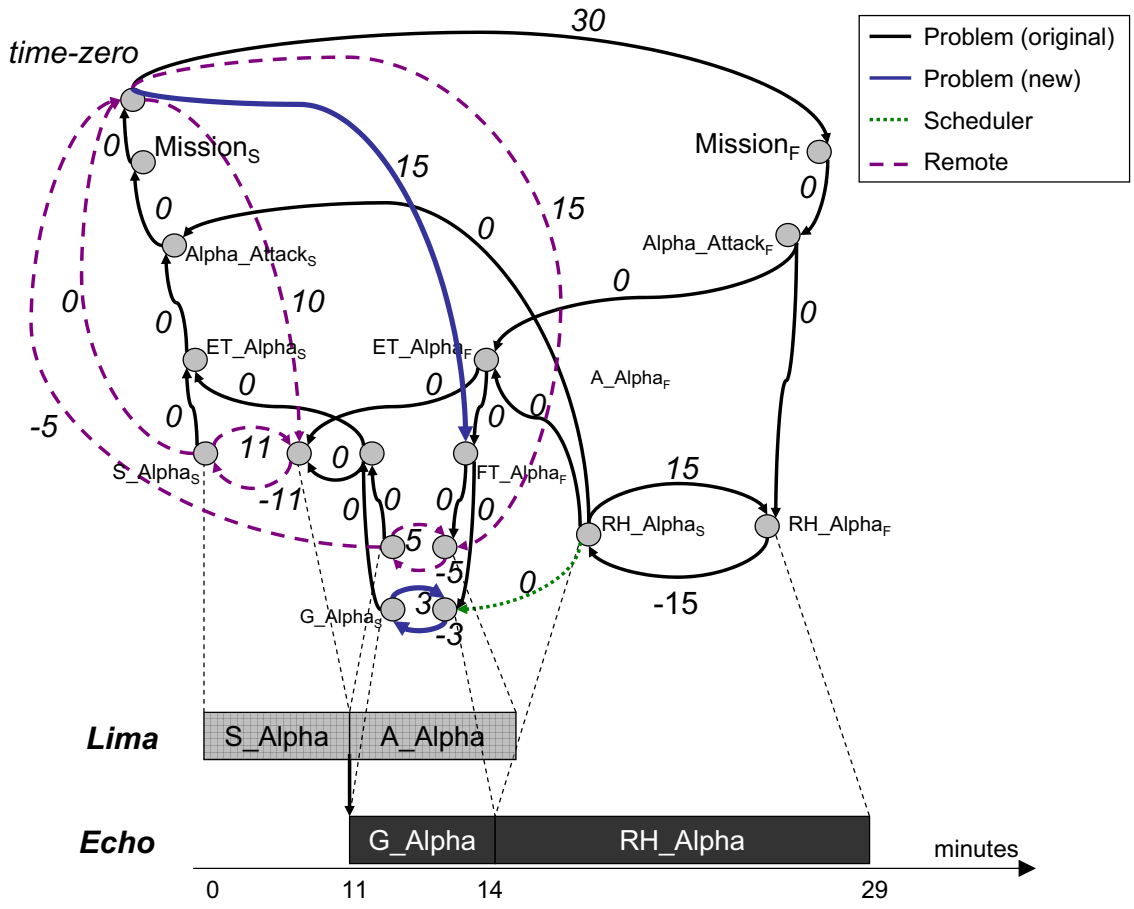

Figure 5.11: Echo's STN after Receiving Updated Constraints

temporal constraints. The revised STN is shown in Figure 5.11 with the new deadline and the updated duration constraint for *G_Alpha* in blue. However, the updated instructions produce a temporal inconsistency in Echo's STN. This new inconsistency is caused by Echo's outdated information about Lima's method *A_Alpha* (its duration as reported by Lima is 5 minutes). The categorized constraints in the STN's negative cycle are shown in Figure 5.12. At this point, Echo can conclude that Lima's reported duration for *A_Alpha* is no longer valid. It is a safe assumption that Lima has also received updated instructions from Delta, and will provide updated information for *A_Alpha* shortly. Echo then *suspends A_Alpha*'s invalid duration, restoring consistency to its STN. The conflict-free STN is shown in Figure 5.13.

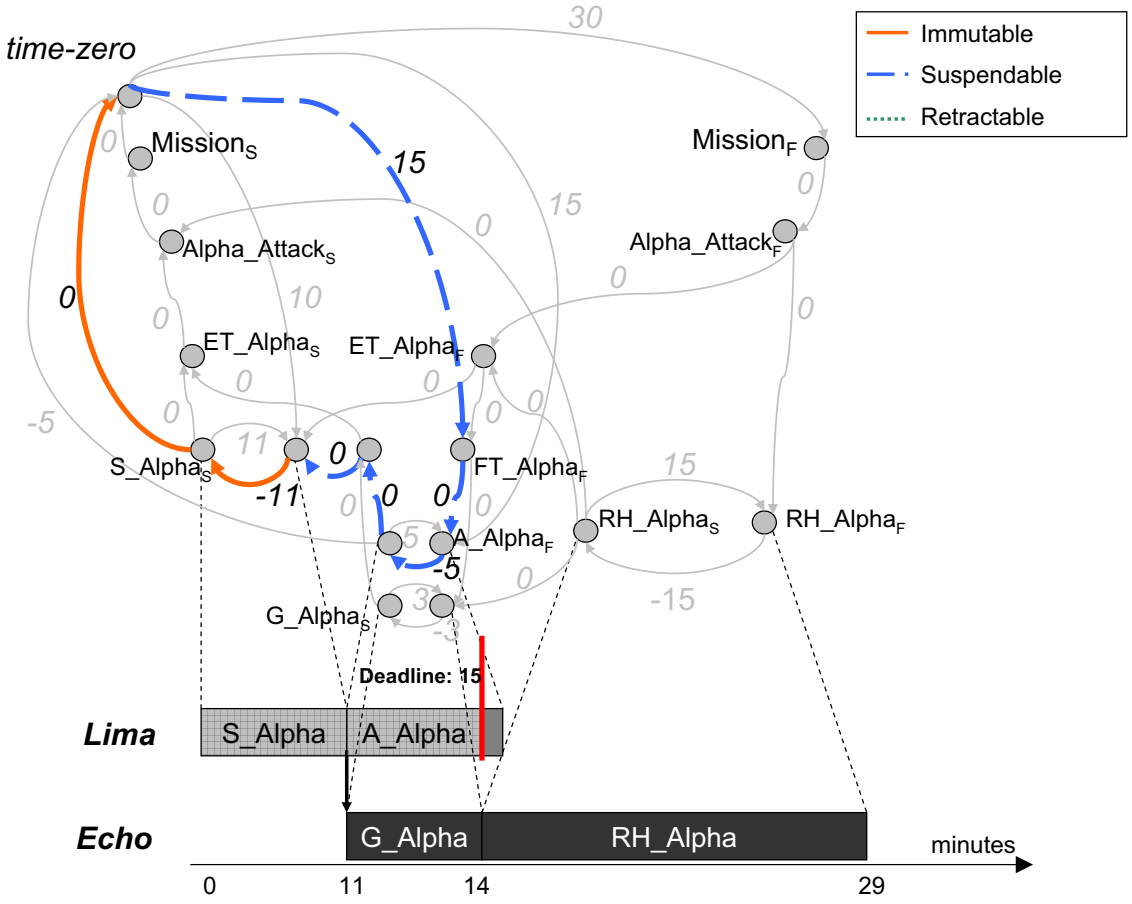

Figure 5.12: The Negative Cycle in Echo's STN after Receiving Updated Constraints

Retracting a deadline constraint

If this case is reached, the Modifiable Conflict Set contains neither an NLE nor a remote activity duration to suspend. The conflict involves an executed activity that has violated its deadline constraint (or that of an ancestor task) [5]. Since the activity has been executed, its duration is immutable and the activity cannot be unscheduled any longer (first strategy above). The agent's only choice to restore consistency in the STN is to retract the deadline constraint. We note that if the violated deadline is "inherited" from an ancestor task, it is sufficient to break the structural constraints linking the method to its parent task. Decoupling the late method from its parent

[5]The case where the executed activity has violated its release constraint can be trivially avoided prior to execution by not starting a method before its release. The cMatrix agent adopts this strategy, and we do not look further into this possibility.

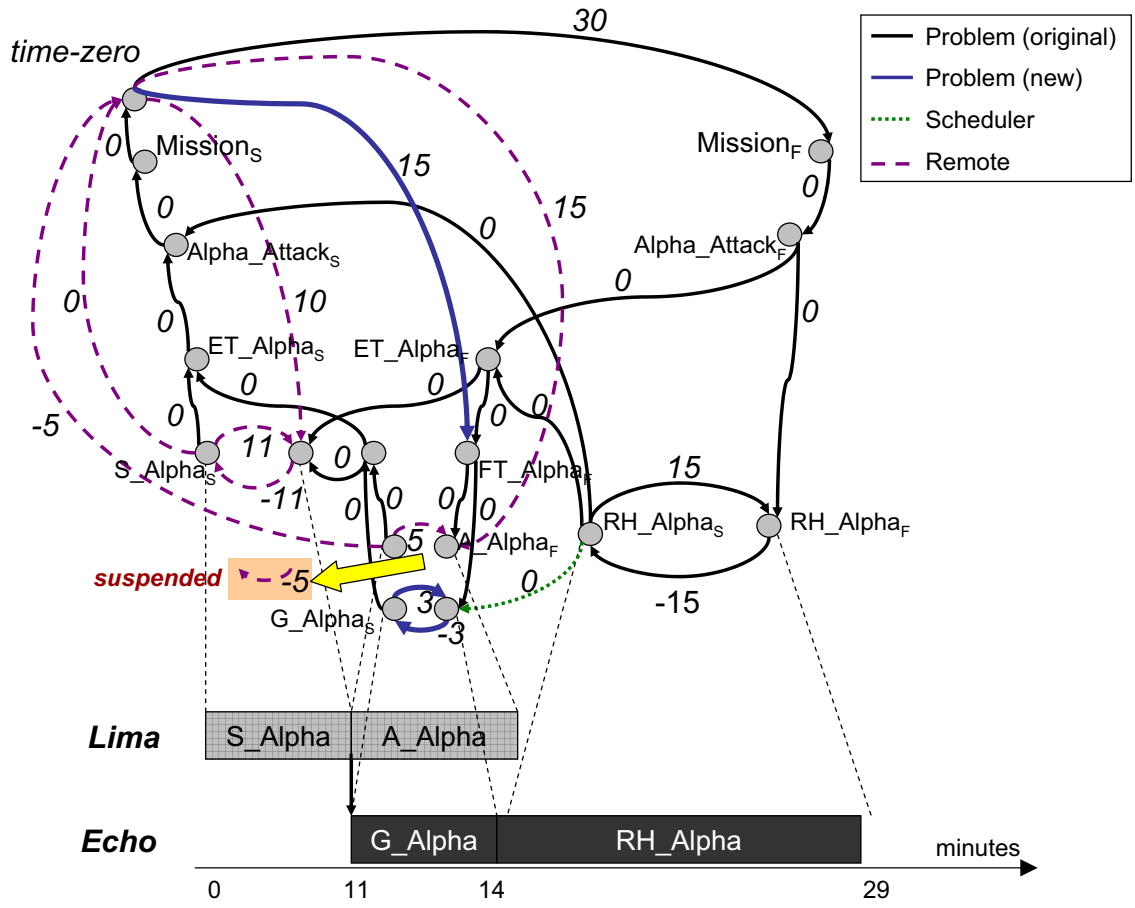

Figure 5.13: Action 3: Suspending a Remote Activity's Duration

minimizes the impact of the constraint suspension. Retracting the deadline constraint itself would instead affect all activities in the hierarchy below the deadline-holding ancestor task.

An example of this situation occurs as Echo moves in to engage the insurgents at Alpha. Despite the heavy aerial bombardment they had received from Lima (which had led the commanding unit Delta to believe that resistance would be minimal), Echo encounters fierce resistance from remaining terrorist elements hiding in the wreckage. What was believed to be a simple mop-up operation scheduled to last 3 minutes turns out into a more protracted battle lasting 8 minutes. The delay in completing *G_Alpha* violates the deadline imposed on the parent task *FT_Alpha* to finish at minute 15. Instead, the operation is only completed at minute 20. The violated deadline translates into an inconsistency in the STN. The categorized constraints in the negative

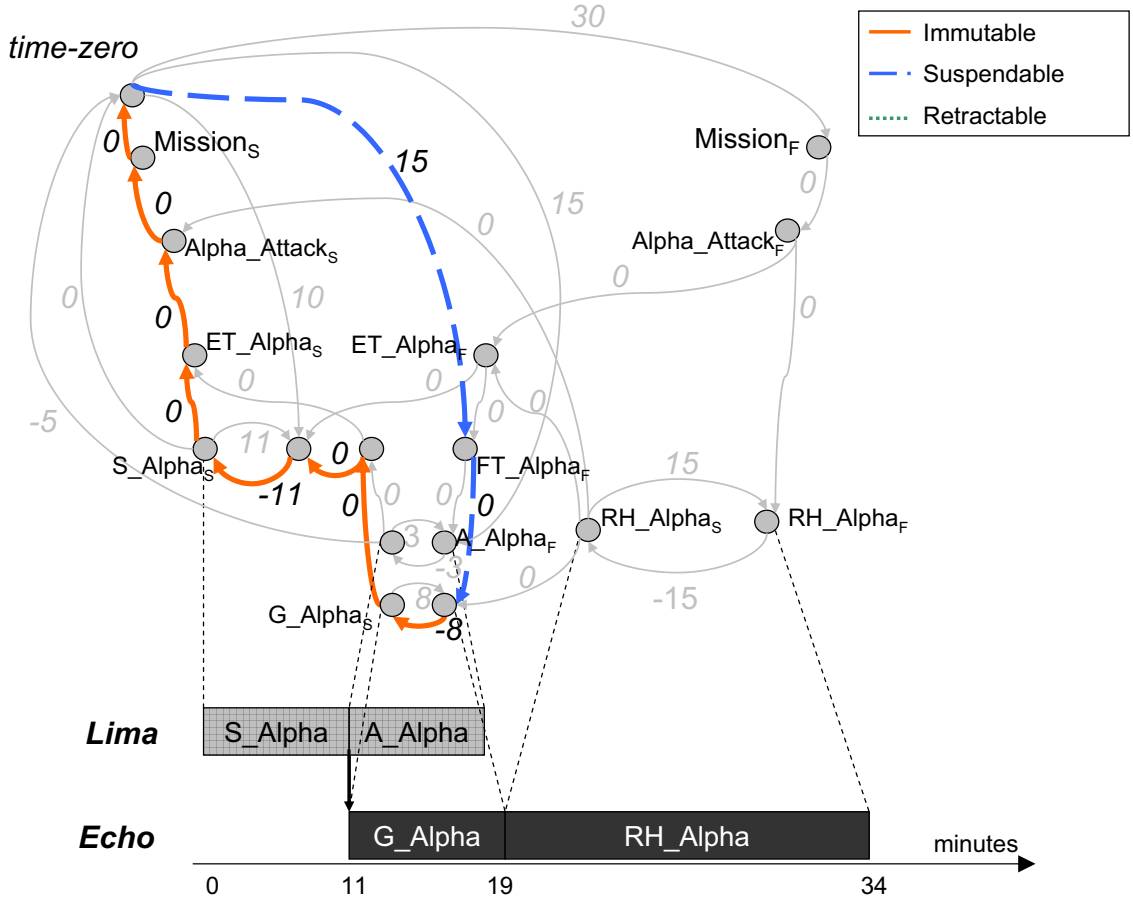

Figure 5.14: The Negative Cycle in Echo's STN After Finishing *G_Alpha* late

cycle are shown in Figure 5.14. The Modifiable Conflict Set consists of only two constraints: The deadline constraint on *FT_Alpha*, and the structural constraint linking the finish points of *FT_Alpha* and *G_Alpha*. Through this hierarchical constraint, *G_Alpha* "inherits" *FT_Alpha*'s deadline. To restore the STN to consistency, Echo has to stop the enforcement of this deadline on the late method *G_Alpha*. While Echo could accomplish this by removing *FT_Alpha*'s deadline constraint from the STN, this action does not only affect *G_Alpha*, but also Lima's method *A_Alpha* (which also inherits the deadline constraint from *FT_Alpha*). Instead, Echo can achieve the same effect by simply removing the aforementioned structural constraint, an action that does not affect *A_Alpha*. The resulting STN with this conflict removed is shown in Figure 5.15.

After an agent applies the selected conflict resolution strategy to a temporal con-

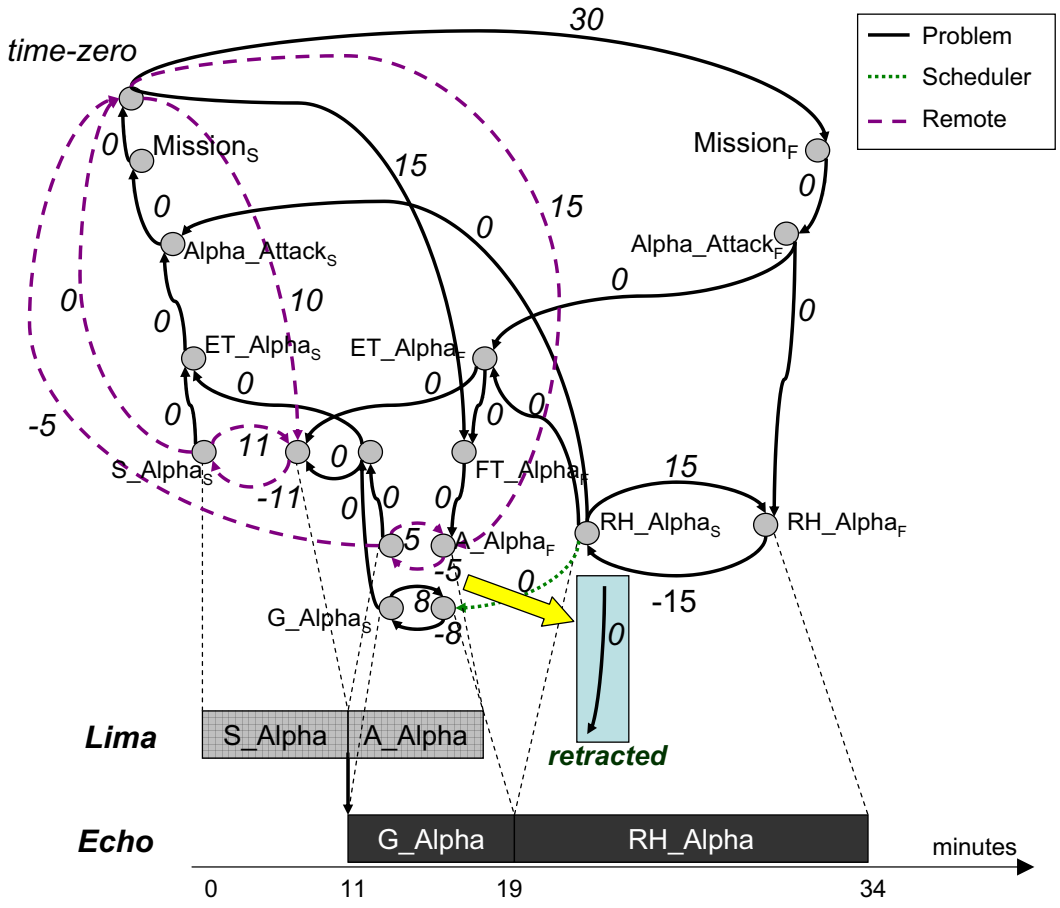

Figure 5.15: Action 4: Retracting a Deadline Constraint

flict, this conflict is resolved. In some cases, however, the resolution of a given conflict will unmask further temporal inconsistencies. An example of this situation occurs in the previous situation involving *G_Alpha*'s deadline. Retracting the deadline constraint from the STN does not restore it to a consistent state. Instead, once this conflict is removed, a second conflict is exposed: The delay in completing *G_Alpha* on time pushes the next method on the timeline, *RH_Alpha*, and it can no longer meet its deadline. This second STN conflict is shown in Figure 5.16. Echo needs to address this second conflict to restore the STN to a consistent state. In such situations, the agent repeatedly applies the conflict resolution strategies described above until the STN is conflict-free.

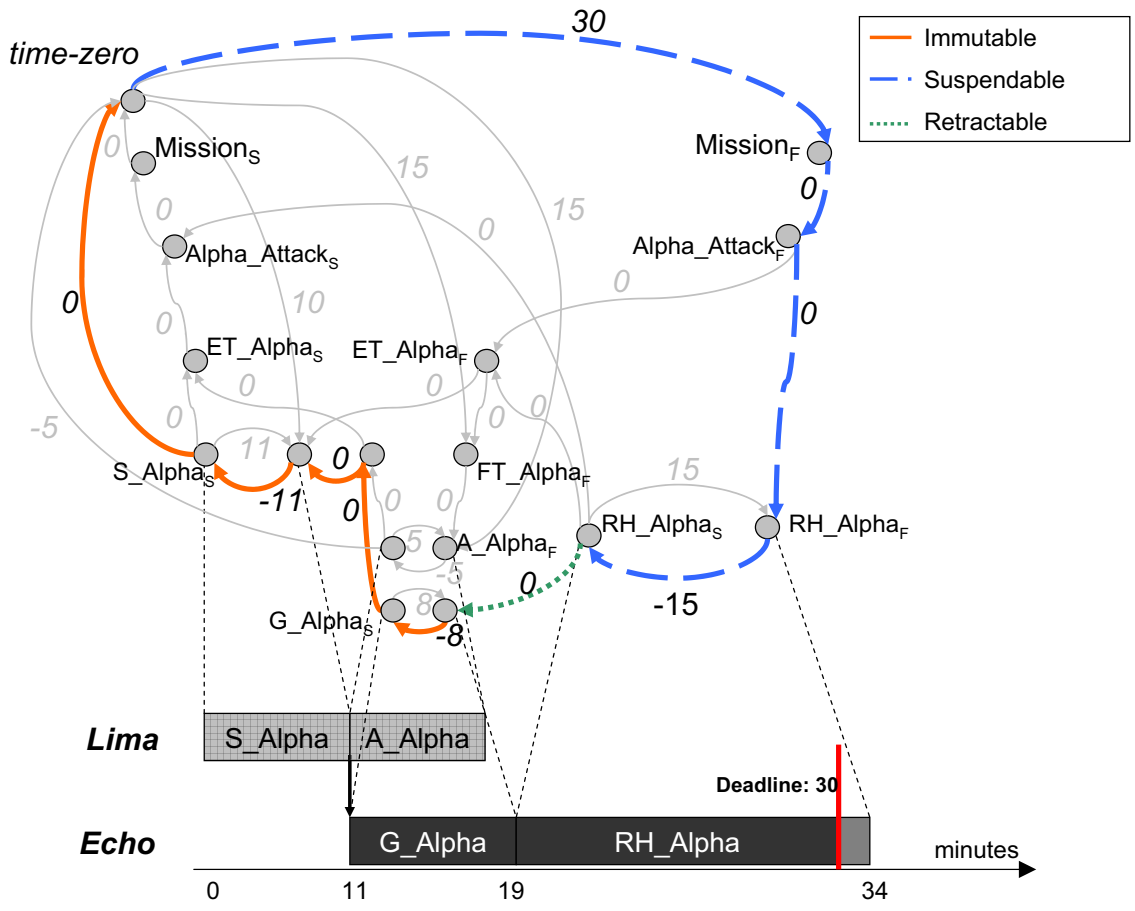

Figure 5.16: The Negative Cycle in Echo's STN After Retracting *G_Alpha*'s Deadline

5.1.4 Modulating Reactivity to Inconsistencies

Some inconsistencies that arise during the execution of the agents' schedules occur due to the asynchronicity of message communication. A case in point is illustrated by one of our example in the previous section: When Echo receives updated instructions from the commanding unit Delta updating the duration of *G_Alpha* and imposing a deadline on *FT_Alpha*, a negative cycle appears in Echo's STN. This inconsistency is caused by outdated information on Lima's method *A_Alpha*, and only occurs because the message from Lima containing updated information for this method has not yet reached Echo.

These spurious inconsistencies resolve themselves upon the arrival of the necessary inter-related update. In this sense, conflict resolution actions that repair these

inconsistencies are unneeded. If an action is taken to resolve the conflict before the arrival of the connected information, this action will be undone once this information arrives. These "do-undo" steps simply waste resources by making unneeded changes to the agents' schedules (with potential further repercussions if these actions affect other agents). To distinguish whether a conflict is potentially spurious, we categorize STN conflicts into two types, depending on the event that causes the conflict:

1. Execution-related conflicts: These conflicts are caused by actual *execution* events (i.e. the start or finish of a method). The local agent receives an update from the simulator (directly, or indirectly when another agent relays the information) that informs it of the start or finish of a method. When it attempts to incorporate this information, an STN conflict arises. These conflicts are always "real", in the sense that they are caused by an event that has already taken place.

2. Non execution-related conflicts: These conflicts are caused by *scheduling* events (i.e. remote agents changing their schedules). The local agent receives new information from another agent with updated temporal bounds for a remote pending activity. However, these updated bounds are inconsistent with other temporal constraints in the local agent's STN. Sometimes, these conflicts are "fictitious": their appearance is simply caused by the asynchronicity of message arrival. The updated information causes a temporal conflict because the agent has not yet received a second (connected) piece of information (which is on its way). When the linked information arrives, the apparent conflict automatically resolves itself.

A conflict resolution action is applied immediately for execution-related conflicts. The information that caused this conflict is an actual event that has taken place, and it needs to be incorporated into the agent's belief state without delay. Non execution-related conflicts, on the other hand, are caused by temporal conflicts between pending activities. Therefore, incorporating the information that causes the conflict is not as pressing. Further, as was explained previously, the conflict may be "fictitious", caused by the order of information arrival, and may resolve itself once additional information pieces arrives. To restore the STN to consistency, we have then two choices: (1) Apply one of the conflict resolution actions (as described in the previous section), and incorporate the information, or (2) Retract the new information (restoring the previous consistent STN state), wait for new information to arrive, and attempt to reincorporate the information at a later point. In the case of spurious inconsistencies,

the second strategy can provide a means of modulating the agent's reactivity to inconsistencies, preventing it from making unnecessary changes to its schedule.

We have developed a *delayed* conflict resolution strategy that minimizes unnecessary conflict resolution actions when potentially spurious conflicts arise. When the agent receives an update to the temporal bounds of a remote pending activity A, and these update bounds cause an inconsistency, the bounds are temporarily set aside (restoring the previous ones). The agent attempts to incorporate A's new bounds over a delay period. This delay period can be specified in two ways:

1. Maximum number of updates: This delay is measured in the number of new updates that the agent receives since A's new bounds arrive. For example, if the delay is 2, the agent will attempt to incorporate A's new bounds over the next 2 updates.

2. Maximum elapsed time: This delay is measured in the time that elapses since the agent receives A's new bounds. For example, if the delay is 2, the agent will attempt to incorporate A's new bounds over the next 2 time units.

If the delay period expires, and the new bounds have not been incorporated (i.e. an STN conflict occurred whenever the agent attempted to update A's bounds), the inconsistency is assumed to be "real". The agent then incorporates A's new bounds by applying a conflict resolution action on the ensuing inconsistency.

To illustrate a situation when *delayed* conflict resolution can be used, we can reuse the situation depicted in Figure 5.11. Briefly recapping the situation, Lima has finished *S_Alpha* late, and Delta has updated Echo's model by changing the duration of *G_Alpha* to 3, and imposing a deadline on *FT_Alpha* at time 15. This new deadline conflicts with Lima's reported duration of 5 for method *A_Alpha*. The new deadline for *FT_Alpha* was incorporated into the agent's belief state by applying a conflict resolution action that *suspended* the duration constraint of *A_Alpha* to restore the STN to a consistent state. The resulting conflict-free STN is shown in Figure 5.13.

Instead of incorporating *FT_Alpha*'s new deadline by *suspending A_Alpha*'s duration, an alternative course of action would be to *delay* incorporating the deadline into the agent's belief state while we wait and see if some new information causes the conflict to disappear on its own. If Lima sends updated information for its method *A_Alpha* during the delay period, the deadline can then be incorporated without causing an STN conflict.

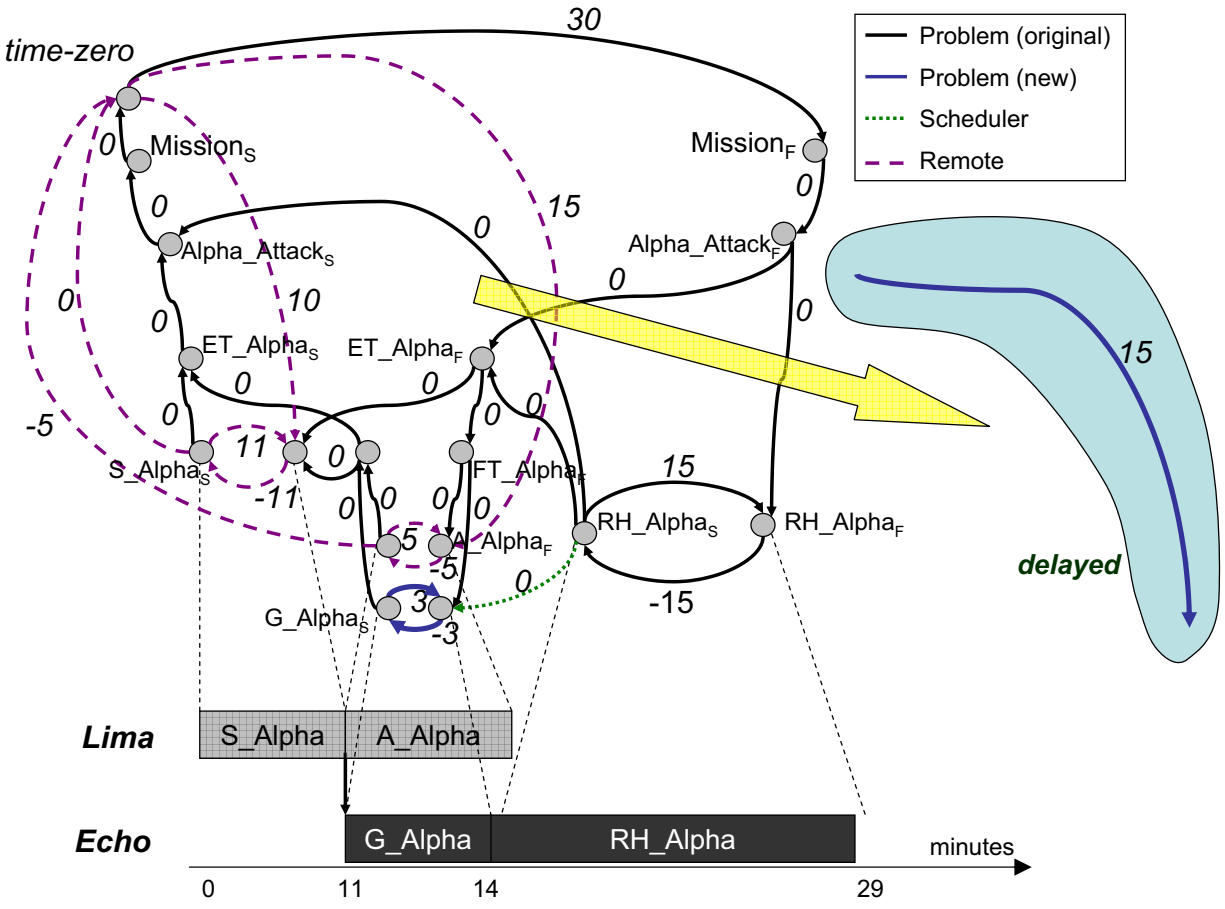

Figure 5.17: Delaying Incorporation of *FT_Alpha*'s Deadline

We develop this alternative scenario to illustrate a successful application of *delayed* conflict resolution: After assessing the situation caused by Lima's unexpected delay in completing *S_Alpha*, Delta relays to Lima a revised expected duration for *A_Alpha* of 3 minutes (from 5 minutes). In the meantime, Echo has received *FT_Alpha*'s new deadline from Delta, and it conflicts with the outdated information about *A_Alpha*'s duration. Echo momentarily delays incorporating the deadline, while it awaits more information. The resulting (conflict-free) STN with *FT_Alpha*'s new deadline is shown in Figure 5.17. While Echo was dealing with this conflict, Lima has sent a message to Echo with the new duration for *A_Alpha*. The sequence of actions Echo takes upon receipt of this message depends on the time of its arrival. There are two possibilities:

1. The message reaches Echo *before* the delay period for incorporating *FT_Alpha*'s new deadline is over. Echo updates the duration of *A_Alpha* to 3 minutes. Then

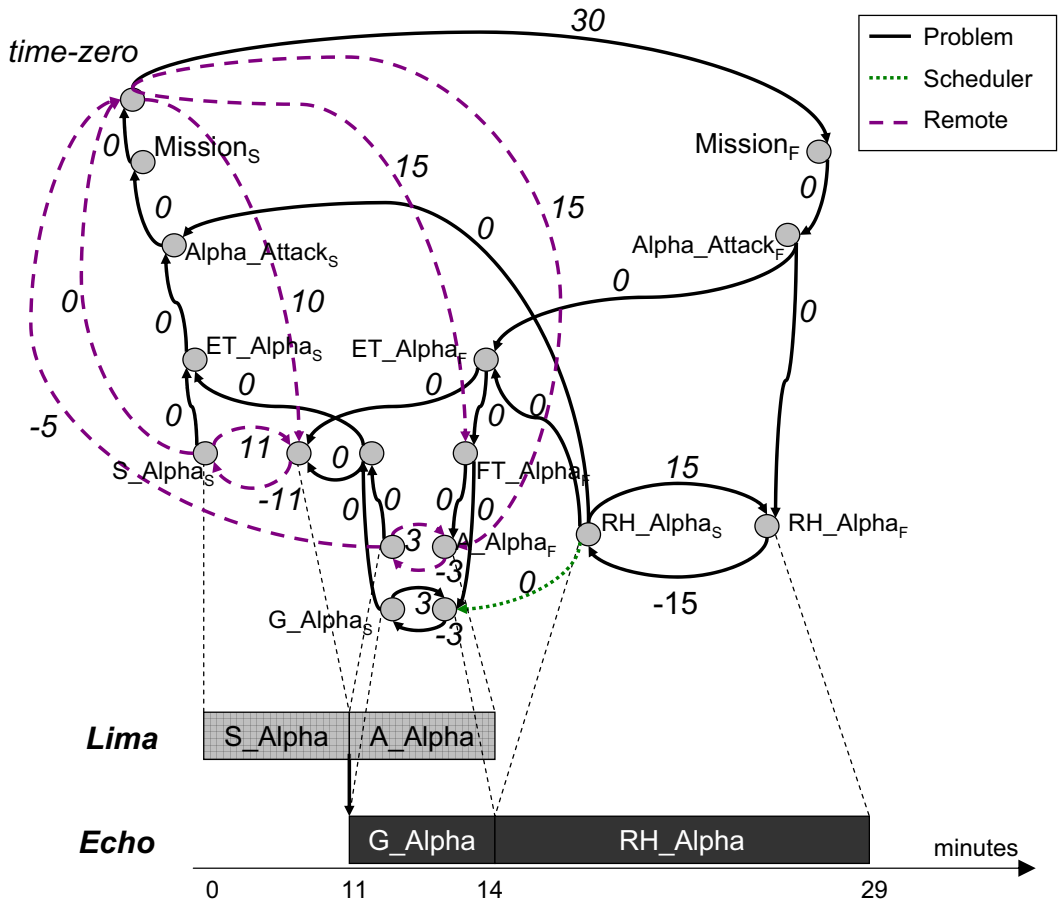

Figure 5.18: After Receiving *A_Alpha*'s Updated Duration

it attempts again to update *FT_Alpha*'s deadline. The new attempt succeeds without producing a conflict.

2. The message reaches Echo *after* the delay period for incorporating *FT_Alpha*'s new deadline is over. Once the delay period has ended, Echo incorporates *FT_Alpha*'s new deadline. It resolves the ensuing conflict by *suspending A_Alpha*'s duration constraint. When the message arrives, Echo updates the duration constraint to 3 minutes, and attempts to reinsert it into the STN. The attempt succeeds.

For both cases, the final resulting STN is shown in Figure 5.18 [6].

[6]In the example situation, the introduction of the delay does not lead to significant savings in computational expense. Other situations, however, specially those involving unscheduling actions, are more disruptive, and can produce significant savings.

5.2 Experimental Design and Results

To test the validity of our approaches, we conducted experiments that compared the performance of two agent teams: (1) A team of "risk-taking" agents using our conflict resolution actions to resolve inconsistencies when unexpected events occur, and (2) a "risk-averse" team of agents using a conservative scheduling strategy that prevents conflicts from arising during execution. A second set of experiments validated the reduction in scheduling time encountered by agents using *delayed* conflict resolution. Some initial results for the strategies developed in this chapter were presented in [52].

5.2.1 Testing Conflict Resolution Actions

To compare the team of "risk-taking" against the team of "risk-averse" agents, we designed several sets of randomly generated C_TAEMS scenarios. The scenarios were constructed to highlight the lost opportunities incurred by risk-averse agents. Their design emphasized two features: (1) Scheduling conservatively misses opportunities for higher performance, and (2) Agents cannot easily recover from conservative decisions, regaining lost opportunities.

An example 10-agent C_TAEMS scenario generated for these experiments is shown in Figure 5.19. The activity hierarchy is arranged left-to-right, rather than top-to-bottom for easier viewing. Scheduled activities are shown in blue, unscheduled activities in red. Higher-level tasks have a rounded-edge rectangular shape, methods a straight-edge rectangular shape. The text inside tasks shows: (1) The name of the task, (2) its QAF (such as SUM or SUMAND), (3) the task's release and deadline if any, and (4) the scheduled quality of the task (by applying the QAF on its children activities). The text inside methods shows: (1) The name of the method, and (2) the quality/duration it will obtain if executed (pre-selected before execution and given to the simulator).

We generated 10-agent, 25-agent, and 50-agent scenarios (scenario generation is described in section 4.2.1). The 10-agent scenarios have 2 "problem" tasks. The 25-agent scenarios have 5 "problem" tasks. The 50-agent scenarios have 10 "problem" tasks. Problem tasks have a SUMAND QAF, and have two window children each. Each window has either two cnode children with a MAX QAF linked by an enables NLE, or a single cnode child with a SYNCSUM QAF. Each cnode has three children methods: The "primary" method, a "fall-back", and a "redundancy". Under a MAX

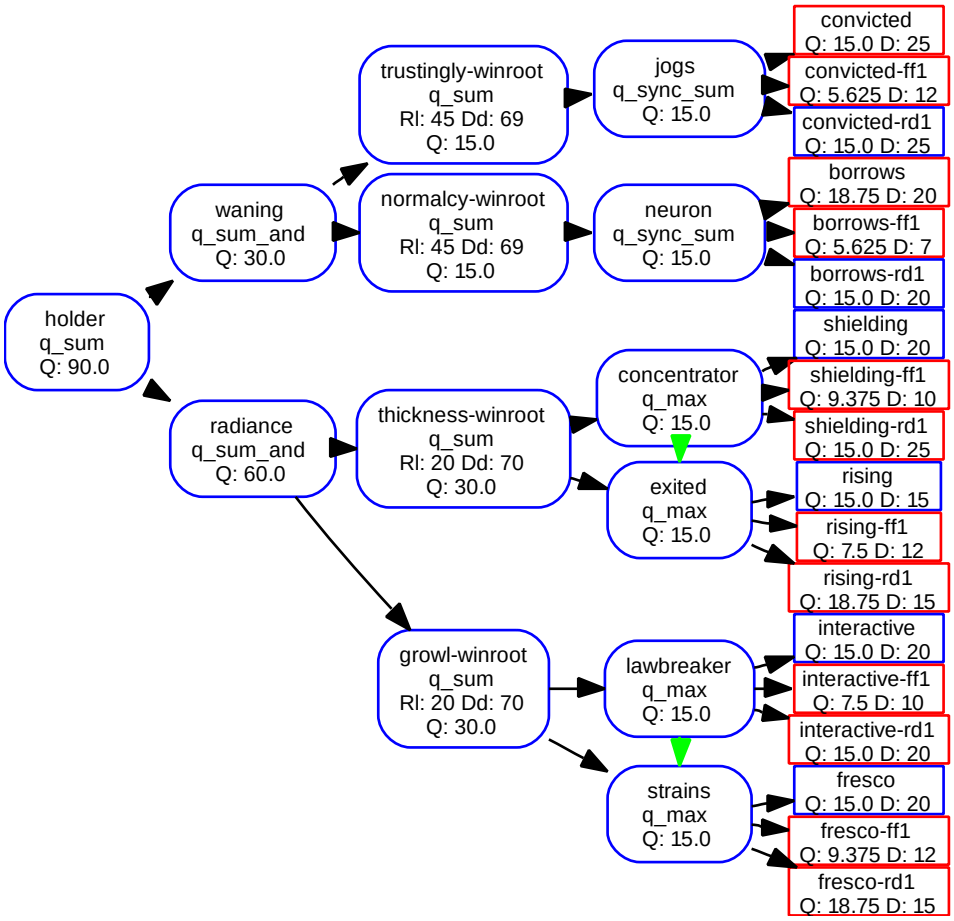

Figure 5.19: Example C_TAEMS Scenario for Testing Conflict Resolution Actions

task, both fall-back and redundancy methods have half the expected quality and duration as the primary method. Under a SYNCSUM task, the fall-back method has half the expected quality and duration of the primary method, while the redundancy method has the same expected quality and duration as the primary method. The primary methods were given an expected quality of 15, and an expected duration of 20 time units. The three points in a method quality/duration distribution consisted of an expected value, a second value 25% lower, and a third value 25% higher.

For each scenario, we compared the performance of two strategies for managing execution-time inconsistencies, one "risk-averse" and the other "risk-taking":

1. Conservative (MD): This strategy uses a "risk-averse" scheduling approach. A centralized solver generates an initial schedule that assumes methods will ex-

ecute their maximum duration. The agents' schedulers use the same assumption when rescheduling during execution. This conservative strategy is akin to the preventative approaches used in previous STN-based multi-agent applications that construct fail-safe schedules to avoid inconsistencies during execution [180, 74].

2. Conflict Resolution (CR): This strategy uses a more "risk-taking" scheduling approach. A centralized solver generates an initial schedule that assumes methods will execute their expected duration. The agents' schedules also schedule using expected durations. When inconsistencies arise, the agent uses a conflict resolution action (described in section 5.1.3) to restore its STN to consistency. For these experiments, the agents used no delay (conflicts were resolved right away).

We created a total of 9 sets of 150 scenarios (each containing 50 10-agent scenarios, 50 25-agent scenarios, and 50 50-agent scenarios). The nine sets differed in two key parameters: *deadline tightness* and *uncertainty*. Deadline tightness defines the ratio between the size of the time window of a "window" task, and its duration. Uncertainty indicates the probability that the maximum duration of a method will occur. Both parameters are described in greater detail in section 4.2.1. We hypothesized that increasingly "tighter" windows enhance the advantage of "risk-taking" strategies, as conservative techniques lose opportunities to schedule methods that do not fit to their maximum duration, and that the advantage of "risk-taking" strategies would increase as the probability that the maximum duration of methods will occur decreases.

Table 5.1 displays the results of applying the two inconsistency management strategies, MD and CR, on the generated scenarios. The results validate our initial hypotheses. We run both strategies on each of the 9 sets of scenarios, and divided the results for each set into three parts, based on the number of agents in the problem. For each scenario, we computed the ratio Q_s^r using Equation 4.6, where Q_s^r is the quality ratio achieved by strategy s (either MD or CR), The results in the table show *deadline tightness* varying in the horizontal direction, and *uncertainty* changing in the vertical dimension. For each one of the 9 combined settings of *deadline tightness* and *uncertainty*, the average ratios Q_s^r were computed on the 10, 25, and 50 agent scenarios. These values are shown side-by-side, with the p-value (obtained using the Student's t-test) comparing the results for both strategies below them. The size of the scenario proved important: 10-agent scenarios were too small to derive any

			tighter $\Longleftrightarrow$ *looser* **Deadline Tightness**					
		Ag	1.0		1.2		1.4	
			MD	CR	MD	CR	MD	CR
Uncertainty	0.125	10	0.7663	0.8537	0.8329	0.8926	0.9245	0.9689
			$p = 0.1205$		$p = 0.1718$		$p = 0.1377$	
		25	0.7168	0.9979	0.8079	0.9955	0.9454	0.9847
			$p = 0$		$p = 0$		$p = 0.0018$	
		50	0.6310	1.0	0.7454	1.0	0.9709	0.9966
			$p = 0$		$p = 0$		$p = 0$	
	0.25	10	0.8004	0.8245	0.8724	0.8721	0.9028	0.9346
			$p = 0.6833$		$p = 0.9945$		$p = 0.4004$	
		25	0.7857	0.9848	0.8478	0.9759	0.9521	0.9749
			$p = 0$		$p = 0$		$p = 0.0487$	
		50	0.6939	1.0	0.7816	0.9997	0.9806	0.9891
			$p = 0$		$p = 0$		$p = 0.1214$	
	0.333	10	0.8135	0.8098	0.9292	0.8000	0.9046	0.9400
			$p = 0.9512$		$p = 0.0092$		$p = 0.3602$	
		25	0.8455	0.9650	0.8776	0.9542	0.9666	0.9601
			$p = 0$		$p = 0.0011$		$p = 0.6398$	
		50	0.7536	0.9980	0.8704	0.9941	0.9848	0.9865
			$p = 0$		$p = 0$		$p = 0.7063$	

Table 5.1: Quality Ratios Obtained by Inconsistency Management Strategies

significant conclusions, but the larger 25-agent and 50-agent scenarios show significant differences ($p < 0.05$) between the two strategies. The larger 50-agent scenarios show the greatest differences. As we anticipated, scenarios where the deadlines are tightest (1.0) and uncertainty is lowest (0.125) produce the largest discrepancies in performance between the two strategies. For the 50-agent problem set, the "risk-taking" agents using strategy CR outperform the "risk-averse" agents using strategy MD in every scenario: The average Q_s^r for CR is 1.0, while the average Q_s^r for MD is 0.631. The p-value between the two sets is 0. At the other extreme, when the deadline is the loosest (1.4) and uncertainty is highest (0.333), both strategies perform equally well: The average Q_s^r for CR is 0.9865, and the average Q_s^r for MD is virtually the same at 0.9848. Intuitively, loose deadlines provide the agents enough time to schedule methods to their maximum duration, decreasing the performance loss incurred by conservative strategies. Higher uncertainty increases the likelihood that the method will in fact execute its maximum (rather than expected) duration.

This implies a higher likelihood that "risk-taking" techniques will run into trouble. Nonetheless, even in these scenarios, "risk-taking" agents can perform as well as conservative agents when equipped with techniques that enable them to recover from inconsistencies that they may encounter.

5.2.2 Testing Delayed Conflict Resolution

We tested our delayed conflict resolution approach by comparing the performance of two teams of agents, one that aggressively resolves conflicts when they arise, and a second that waits for a delay period before resolving the conflict:

1. Aggressive Repair (AR): The agents "repair" an STN as soon as a conflict arises by applying a conflict resolution action to resolve the inconsistency.

2. Delayed Repair (DR): The agents delay "repairing" possibly a *fictitious* conflict in the STN, and wait for the next update to see if it contains information that resolves the conflict. If the next update received by the agent contains no new information that removes the conflict, a conflict resolution action is applied to resolve the inconsistency [7].

The strategies were tested on a set of C_TAEMS scenarios used during the Year 2 evaluation of the Coordinators program. These scenarios are described in detail in Section 4.2.2. Table 5.2 shows the results obtained by both strategies, AR and DR, on the selected scenarios. The first two columns display the average quality ratios, Q_s^r, for both strategies. Results for the 25 and 50 agent sets are shown separately.

[7] Since the simulator sends an update to all agents every tick (updating the current time), this simulator update creates an upper bound (of 1 tick) on the delay before a conflict resolution is applied.

	(higher is better)		(lower is better)	
Agents	**AR** (quality)	**DR** (quality)	**AR** (repairs)	**DR** (repairs)
25	0.9092	0.9281	0.9121	0.7960
	$p = 0.6290$		$p = 0.0114$	
50	0.8993	0.9610	0.9540	0.8112
	$p = 0.0809$		$p = 0.0038$	

Table 5.2: Quality Ratio for Inconsistency Management Approaches

Below the Q_s^r values, the p-value comparing the results for both strategies is shown. The results show that both strategies performed equally well, quality-wise. The small difference between the values is not significant. The last two columns show the average number of conflict resolution actions ("repairs") normalized as a ratio, R_s^r, given by the equation:

$$R_s^r = R_s / R_{max} \tag{5.1}$$

where R_s is the number of conflict resolution actions taken on a given scenario by strategy s (either AR or DR), and R_{max} is the maximum number of conflict resolution actions taken by either of the two strategies. The delayed conflict resolution strategy DR shows a marked drop in the number of conflict resolution actions taken by the agents. This decrease in the number of conflict resolution actions taken was achieved without sacrificing quality. The results validate our initial hypothesis that introducing a small delay that prevents over-reactivity to fictitious conflicts can lead to a decrease in unnecessary changes to the agents' schedules, enabling agents to make better use of their resources.

5.3 Summary

STNs present a powerful framework for reasoning about temporal constraints, and have found increasing acceptance in scheduling domains. However, their use in multi-agent domains is complicated by the need to manage inconsistency in individual agent models. Previous work in inconsistency management for distributed STNs has mainly focused on *pro-active* techniques that aim to create schedules that avoid any inconsistencies during execution. Unfortunately, designing "fail-safe" schedules frequently leads to conservative choices that miss opportunities for higher performance.

In this chapter, we present a *reactive* framework that enables scheduling agents to make riskier, but potentially advantageous decisions, and recover from any inconsistencies that may arise. We use STN *conflict-explanation* techniques to interpret STN conflicts (manifesting as negative cycles in the STN graph), and categorize the constraints in the conflict into three sets: *Immutable*, *suspendable* and *retractable*. Immutable constraints represent past events and cannot be changed. Suspendable constraints stand for problem constraints and constraints enforcing information conveyed by other agents. Retractable constraints represent sequencing constraints that are imposed by the agent's scheduler. Conflicts must be resolved by modifying sus-

pendable and retractable constraints. These constraints form the *Modifiable Conflict Set*.

We have designed four conflict resolution actions that an agent can use to restore its STN to consistency. The best-suited action is selected by examining the constraints in the Modifiable Conflict Set. Two of these actions involve local scheduling actions, akin to techniques used in centralized domains. The other two actions work on conflicts that involve remote activities that the agent cannot arbitrarily modify. Our strategy starts by attempting to unschedule local activities to resolve the conflict. However, if no change to the local schedule can resolve the conflict, the agent attempts to *suspend* a constraint in the suspendable set, either an NLE involving a remote activity, or the duration constraint of a remote activity. The constraint remains suspended pending arrival of new information that resolves the conflict. When a constraint is *suspended*, the agent will ensure that its local activities respect all constraints acting on them, so that its schedule remains feasible. Finally, when no constraint can be *suspended* to restore consistency, a method has finished past its deadline. The violated deadline constraint is retracted from the STN.

We tested our conflict resolution framework on a set of scenarios generated to compare the performance of "risk-taking" agents (scheduling to expected duration, and recovering from inconsistencies using our conflict resolution actions) against that of "risk-averse" agents (scheduling to maximum duration). We found that as the flexibility in the schedule decreases (time windows are tight, leading to lost opportunities for risk-averse agents), and the uncertainty decreases (the likelihood that the maximum duration will occur is low), the performance gains obtained by the risk-taking agents significantly increase. Nonetheless, at the other extreme, when flexibility and uncertainty are high, the risk-taking agents continued to perform as well as the conservative team.

While conflicts caused by execution events (i.e. the start and finish of methods) need to be resolved with the information incorporated, other conflicts involving pending remote activities are caused by outdated information and may resolve themselves once updated information arrives. We proposed and tested a *delayed* conflict resolution strategy that introduces a small wait period before resolving these potentially "fictitious" conflicts. Our results show that the use of this delay leads to marked drop in the number of conflict resolution actions taken by the agent team, without adversely affecting the quality obtained. The delay modulates the reactivity of the agents to potentially "fictitious" conflicts, enabling them to make better use of their

resources.

Chapter 6

Just-In-Time Backfilling

In this chapter, we continue our study into the management of durational uncertainty within scheduling applications that use distributed STNs. Chapter 5 discussed reactive techniques that enable STN-based scheduling agents to recover from inconsistency once it occurs (an inevitability in dynamic, uncertain environments). This chapter presents complementary pro-active techniques that use a model of uncertainty to locate potential inconsistencies that may arise as the schedule executes, and attempt to prevent them, or at least minimize their effect.

Previous research in pro-active techniques to manage duration uncertainty in STN-based scheduling applications has focused on designing schedules that do not encounter inconsistencies during execution. Besides a few exceptions [177, 67], the typical domains of interest involve plans where every activity has to be scheduled and executed (a common goal is to minimize the makespan of the schedule). Instead, we deal with over-subscribed distributed scheduling problems, where agents can choose different combinations of activities in the plan to schedule and execute. The goal is to maximize the overall quality of their combined schedule (see section 2.2). In our domains of interest, plans include interchangeable "alternative" activities that are not mutually exclusive. Our schedule robustness techniques concentrate on identifying valuable activities in *danger* of failing as the scheduled *executes* and scheduling alternative activities in the plan that reduce the risk to the schedule if a valuable activity is lost. We take a *Just-in-Time (JIT)* approach to increasing the robustness of the schedule. Rather than focusing on the full set of *pending* activities, we define a set of *horizon* activities that are close-to-executing, and identify potential deviations from their expected durations by using the modeled durational uncertainty.

Then, we *hypothetically* introduce these (unexpected) durations into the STN. *Problem* activities that may introduce an inconsistency if they execute longer than their expected duration are identified when these extended durations are rejected by the STN. An analysis of the resulting negative cycles reveals the *endangered* activities (those in danger of failing), and the repercussions of the potential inconsistencies on the agents' schedules. The negative cycle analysis enables the agents to determine whether the disruptions to the team schedule caused by the potential inconsistency warrant *preventative* action.

We have designed *passive* and *aggressive* actions to increase the schedule robustness to durational uncertainty by installing (or "backfilling") alternative activities. In passive actions, the "owner" agent of a valuable *endangered* activity informs the agents that own *alternative* activities (i.e. those that can substitute for the endangered activity) of the potential repercussions the failure. When these agents receive the information, they can try to schedule an alternative activity if their current commitments permit. In aggressive actions, the "owner" agent of a valuable *endangered* activity can *coerce* the owner of an alternative activity to execute it, even if it causes this other agent to uninstall some of its scheduled activities to "make room" for it. Aggressive actions are taken only if the *endangered* activity is the next activity scheduled to execute, and no assistance has been obtained from other agents through passive requests.

We have tested our robustness techniques by comparing the performance of a team of "pro-active" agents that use these strategies, against a second team of "reactive" agents that uses the recovery techniques developed in Chapter 5 to overcome failures once they occur. Our results show significantly increased performance for the team of "pro-active" agents, validating the advantages of our robust scheduling strategies.

6.1 The Problem Domain

Our domain of interest involves multi-agent plans that present the agents with interchangeable ways of accomplishing their objectives, in the form of *alternative* activities. These alternative activities provide:

1. Different ways an agent can accomplish a given objective: The agent "owns" one or more "fall-back" activities (that are short-duration and accomplish the objective in a quick-and-dirty way). This fall-back activities can be scheduled

to replace (or in addition to) the "primary" activity (that is longer-duration, more detailed-oriented and produces higher quality).

2. Different agents that can accomplish the same objective: One or more agents own "redundancy" activities that can be scheduled to replace (or in addition to) the "primary" activity (owned by a different agent).

Alternative activities are not mutually exclusive: An agent can choose to execute both its primary and fall-back activities (e.g. the agent initially executes the fall-back to get something up and running quickly, and then, time permitting, it executes the primary activity to attain higher quality). Similarly, different agents can choose to independently execute their alternative activities (e.g. one agent executes the high-quality, but high-risk primary activity, while a second agent executes a redundancy activity so the objective is met even if the primary activity is not successful).

6.1.1 Alternative Activities in C_TAEMS Plans

C_TAEMS plans represent *alternative* activities as MAX tasks (see Section 2.2.1 for a description of C_TAEMS) that expand into a set of methods. These tasks are labeled *cnodes* (to distinguish them from other MAX tasks higher in the activity hierarchy). *Cnodes* represent alternate, non-exclusive ways to accomplish an objective. A *cnode* typically expands into a primary method, assigned to a given agent, and a set of fall-back and redundancy methods. Fall-back methods are owned by the same agent as the primary method and accrue less quality, but are shorter duration. Redundancy methods are owned by different agents, and can be executed instead of, or in addition to, the primary method to achieve a higher level of robustness.

Figure 6.1 illustrates a sample C_TAEMS hierarchy, highlighting the *cnodes* and their children methods. *Cnode* T_1, for example, decomposes into a primary method M_{1p} owned by agent A_1, a fall-back method M_{1ff} also owned by agent A_1 that is shorter duration but accrues lower quality, and a redundancy method M_{1rd} owned by agent A_2. To accomplish the objective of *cnode* T_1, it is sufficient for one of these three methods to execute successfully (accruing positive quality). To increase the likelihood that T_1 executes successfully, one option is for agents A_1 and A_2 to parallelly execute their respective methods, M_{1p} and M_{1rd}. T_1 accrues quality if either method succeeds. A second option is for A_1 to execute the lower-quality, but "safer" fall-back method

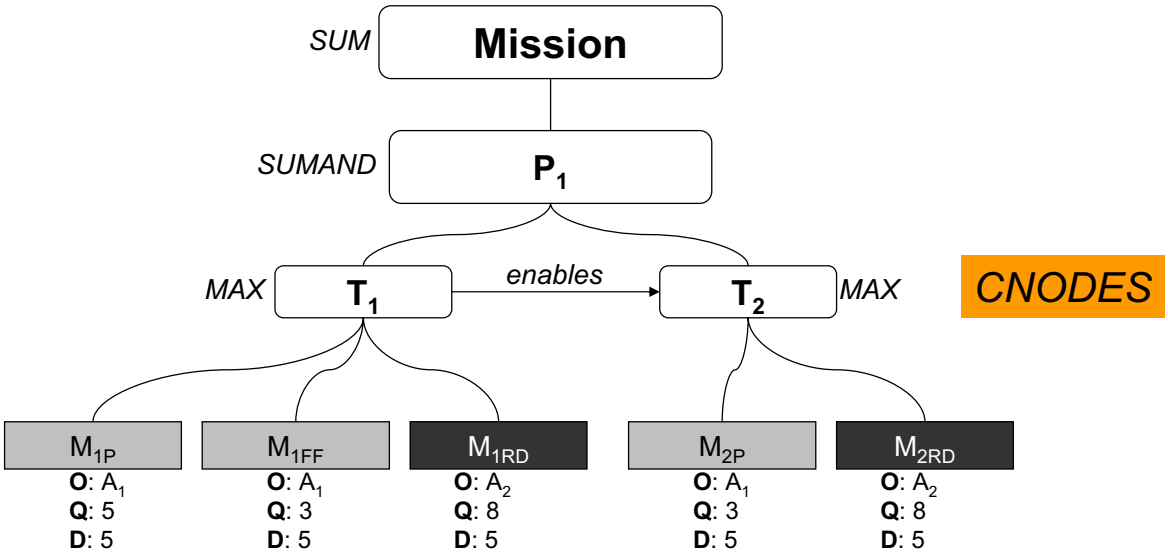

Figure 6.1: A C_TAEMS Plan with Alternative Activities

M_{1ff}, if it deems the likelihood of failure of the primary method M_{1p} too high. In this case, T_1's objectives are met, but the quality accrued is lower.

Restrictions to the Structure of C_TAEMS Plans

To facilitate the design of the robust scheduling techniques we develop in this chapter, we introduce a few restrictions to the structure of the C_TAEMS plans that are valid for the application of our strategies:

- A corollary consequence of viewing *cnodes* as the "lowest-level" activities in the hierarchy representing a plan objective (with their children methods only representing *alternative* ways to accomplish this objective), is that there are no NLEs at the method level. For the robust scheduling work presented in this chapter, NLEs are restricted to link activities at the *cnode* level or higher.

- We restrict the types of NLEs in the plans to hard NLEs (*enables* and *disables*) and soft NLEs (*facilitates* and *hinders*) that do not modify the duration of methods. Soft NLEs that cause a *shortening* (facilitates) or *lengthening* (hinders) in the *executed* duration of a method (i.e. how long the method actually lasts when executed), can result in a method's duration falling outside of the

duration range specified in the uncertainty model. Naturally, as the accuracy of the durational uncertainty model decreases, so does the performance of any technique that relies on this model to increase the robustness of the schedule. Since this additional, hard to quantify, variable can introduce distortions in the evaluation of our techniques, we exclude them from consideration. The techniques we develop can still be used in scenarios with these excluded NLEs, but their performance could be diminished.

6.2 Using STN Inconsistencies in JIT Backfilling

An unexpected activity duration during schedule execution can potentially introduce an inconsistency. In STN-based schedulers, these inconsistencies manifest as negative cycles in the STN graph (see section 2.1). Execution dynamics are a reality in a large percentage of scheduling domains, and STN-based continuous schedulers (see section 3.1.2) need strategies to recover from these inconsistencies, and formulate a new schedule. Chapter 5 developed strategies to *recover* from inconsistencies in distributed scheduling applications. However, waiting for an inconsistency to occur, and then *reacting* to it *after* it has happened can inflict irrecoverable losses to the schedule. Taking *pro-active* action *before* the inconsistency occurs to either prevent it from happening, or reducing its deleterious consequences adds robustness to the schedule, potentially warding against catastrophic failures. Given a durational uncertainty model for the plan activities, this model can be used to find weak points in the schedule, and strengthen it against potential failures.

In this section, we describe *Just-in-Time* Backfilling (JIT-BF), a strategy to distributedly monitor a multi-agent schedule for potential impending failures, and strengthen the schedule against these failures by scheduling redundant activities (or "backfilling"). An agent using JIT-BF leverages the built-in conflict analysis tools of its STN to examine its *pending* scheduled activities that are close-to-execution. The agent introduces *hypothetically* durations for these activities that deviate from their expected value (based on the durational uncertainty model). Resulting inconsistencies signal a potential impending failure, and their analyses detect which activities are likely to fail, and how the failure would affect the multi-agent schedule. The agent can use this information to coordinate with other agents the scheduling of redundant activities that can be executed instead of, or in addition to, an activity that is likely

to fail. JIT-BF can be divided into four steps:

1. *Determining the Horizon Activities*: Starting from the current executing activity (or last executed activity if none is currently executing), an agent determines the set of *horizon* activities that will be examined for potential failures. These *horizon* activities consist of the union of the *executing* activity and the *On-Deck Set*, a subset of the *pending* activities that are close-to-execution.

2. *Discovering Schedule Weaknesses*: Mining the durational uncertainty model, an agent collects potential duration values for the *horizon* activities (and their predecessors) that deviate from their expected durations. These "unexpected" durations are introduced into the STN. A rejection by the STN of one of these unexpected durations (denoted by a *negative cycle*), represents a potential failure point, indicating what would occur if the durations that were introduced into the STN actually materialize during execution. The agent collects all the *negative cycles* produced by the STNs for each of these potential failures.

3. *Analyzing Hypothetical Inconsistencies to Discover their Effect on the Schedule*: The agent analyzes the collected negative cycles to discover the activities in the schedule that are affected by the potential inconsistency. Once this set of *failing* activities has been obtained, the agent calculates (1) the potential effect to the schedule of the potential failure (the quality loss incurred by forfeiting the *failing* activities), and (2) the probability that the durations rejected by the STN will materialize during execution (from the uncertainty model). By understanding how the schedule is affected by the potential failure, and how likely is the failure, the agent can decide whether the repercussions of the failure warrant the undertaking of preventative action to reinforce the schedule.

4. *Backfilling to Strengthen the Schedule*: When the agent determines that a potential failure should be avoided, it takes action to reinforce the schedule, possibly in combination with other agents. Actions taken to strengthen the schedule are intrinsically domain-dependent: In our domain of interest, schedule strengthening actions include the scheduling of redundant activities, and the swapping of at-risk activities with safer alternatives.

JIT-BF presents a strategy that scheduling agents operating in a dynamic, uncertain execution environment can use to safeguard their schedules against duration

uncertainty (other forms of uncertainty, such as quality uncertainty in C_TAEMS, are outside the scope of JIT-BF). JIT-BF makes use of the key insight that that in dynamic, uncertain environments, it is *ineffectual* to attempt to increase the robustness of the whole *pending* schedule. Given that the latter portions of the schedule are likely to change as execution deviates from the expected trajectory, the schedule strengthening actions required for these later activities also changes as their time of execution approaches, and it makes little sense to take these actions too far in advance. This JIT approach to strengthening the multi-agent schedule helps to maintain tractability by avoiding a potential exponential search. A comprehensive approach that attempts to examine all possible duration combinations of all the *pending* activities in the schedule can easily bog down in an exponential explosion of possible combinations, and is not feasible in general. Take, for example, a schedule with 20 *pending* activities, each with a discrete duration distribution specifying 3 possible durations [1]. The number of potential duration combinations is $3^{20} = 3.49 \times 10^9$. Even if some tricks are used to reduce the search space (e.g. not trying a larger duration for an activity whose shorter durations are known to be infeasible), such an algorithm is not practical in a continuous scheduling application.

The next sections present a motivating scenario that shows the potential benefits of JIT-BF, and describe each of the four steps of the JIT-BF process in greater detail.

6.2.1 Motivating Scenario

The scenario developed in section 5.1.1 involved a military team working together to rescue hostages held by terrorists in two locations, *Alpha* and *Gamma*. The example focused on team Charlie, tasked with rescuing the hostages in *Alpha*. Team Charlie is divided into two groups: *Lima*, in charge of aerial "softening" of the target, and *Echo*, the ground team in charge of entering *Alpha*, eliminating the terrorists and rescuing the hostages. During the execution of the mission, Lima ran into problems while softening the resistance: The terrorists were in possession of surface-to-air (SAM) missiles and were able to shoot down one of Lima's helicopters and force the remainder to withdraw to a safer distance. Lima was unable to finish softening the target in its expected time, endangering the remainder of the mission.

We now make note that the reason that Lima's lateness endangered the mission

[1] A continuous PDF can be discretized to any desired granularity. We assume discrete distributions for the work in this chapter.

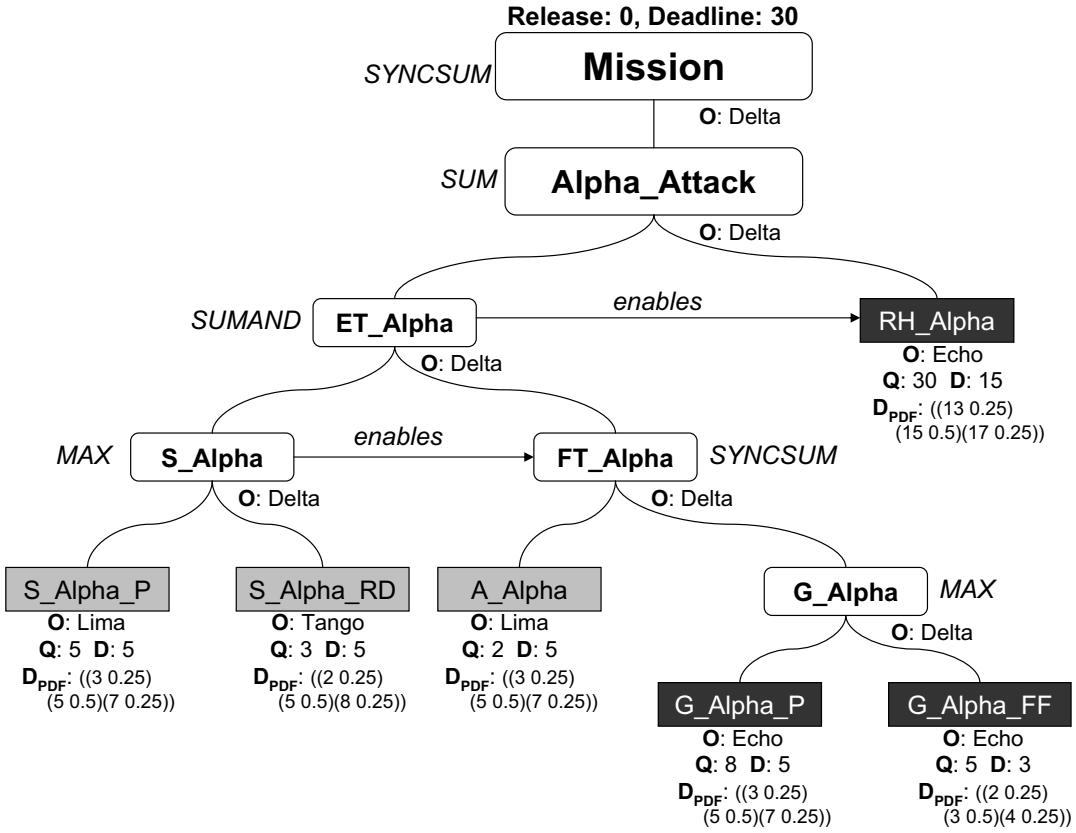

Figure 6.2: The Redundant C_TAEMS Plan for Team Charlie.

is that there was *no backup* action for the case that Lima ran into difficulties. The mission assumed that Lima would succeed in performing its activity by its prescribed time, and made no provisions for the case it didn't. We now consider the case where the plan includes *redundancies*, or backups for potential failures points: Suppose team Charlie includes a light artillery team, *Tango*, that can direct mortar fire at the insurgents. If Lima is unable to provide the planned aerial bombardment, Tango can step in and fire mortar rounds to suppress the insurgents' fire. While not as accurate, and more risky to the hostages, a mortar bombardment can effectively diminish the insurgent threat and enable Echo to rescue the hostages.

Team Charlie's C_TAEMS Plan

An updated plan for team Charlie incorporating *alternative* activities is shown in Figure 6.2. This new plan incorporates both fall-back and redundancy activities.

S_Alpha, the softening of the target that needs to occur prior to Echo's engagement of the terrorists is now a *cnode*. Its children correspond to alternative ways of accomplishing the "softening". *S_Alpha* decomposes into two children methods:

1. *S_Alpha_P*: The primary method. This is the preferred way of accomplishing the softening of the target. It represents Lima's aerial bombardment of Alpha using Apache helicopters.

2. *S_Alpha_RD*: The redundancy method. It represents an alternative way of accomplishing the softening of the target, and involves Tango aiming mortar fire at Alpha.

Similarly, *G_Alpha*, Echo's mopping up of the remaining terrorists after the softening action is now also a *cnode*. *G_Alpha* decomposes into two children methods:

1. *G_Alpha_P*: The primary method, and preferred way of mopping up the terrorists. It involves Echo entering *Alpha* with small arms to minimize the risk to the hostages.

2. *G_Alpha_FF*: The fall-back method. It involves Echo using heavier weaponry to enter *Alpha* to minimize the operation's time, with potentially greater collateral damage.

The STNs and schedules for Echo, Lima and Tango at the start of the mission are shown in figures 6.3, 6.4 and 6.5. (Tango's schedule is not shown since it is not yet an active participant). As before, black solid lines denote *problem* constraints, purple dashed lines represent *remote* constraints, and green dotted lines are *scheduler* constraints. Time points corresponding to scheduled activities are shown in gray, while time points corresponding to unscheduled activities are shown in orange.

How Alternatives Can Save the Mission

As Lima's Apache helicopters approach Alpha, they are attacked by anti-aircraft missiles. After 2 minutes of battle, one of Lima's helicopters is shot down, and the other two are forced to withdraw to a safer distance. Given the new situation, Lima estimates that while it is likely that it can subdue the insurgents in 10 minutes (rather than the original 5 minutes anticipated), there is a 25% probability that it may take as long as 12 minutes.

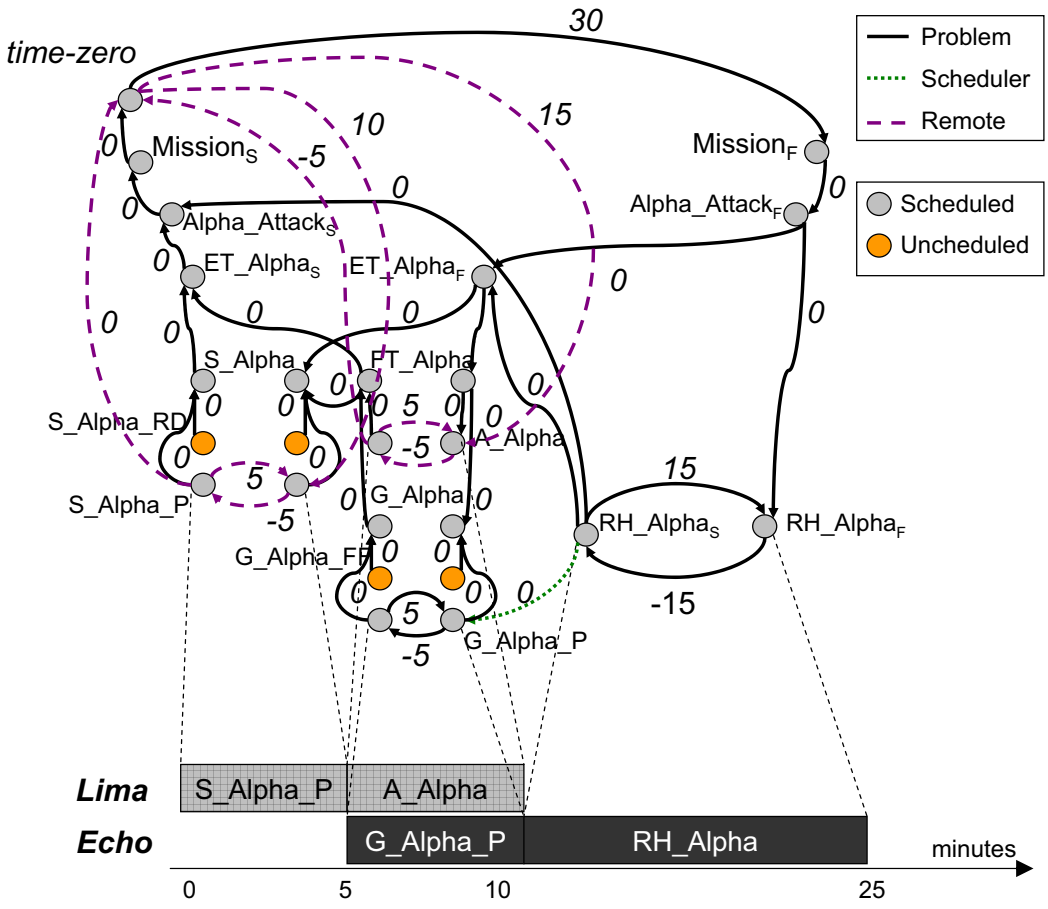

Figure 6.3: The STN and Schedule for Echo at the start of the mission.

At this point, we will present two alternative scenarios. The first scenario involves Lima continuing its attempt at subduing the enemy resistance on its own. The second scenario presents Tango jumping into the fray when it realizes that Lima may not be able to complete its part of the mission in a timely manner:

1. *Scenario 1*: Lima continues aerial bombardment of Alpha, but the insurgents are well entrenched, and better prepared than anticipated. A second helicopter suffers damage from enemy fire 7 minutes into the mission. With one helicopter down, and a second damaged, Lima is forced to abort, causing team Charlie to fail its rescue operation at Alpha. Echo's STN and the current schedule of the team at minute 8 is shown in Figure 6.6 (the STN's of Lima and Tango are almost identical to Echo's).

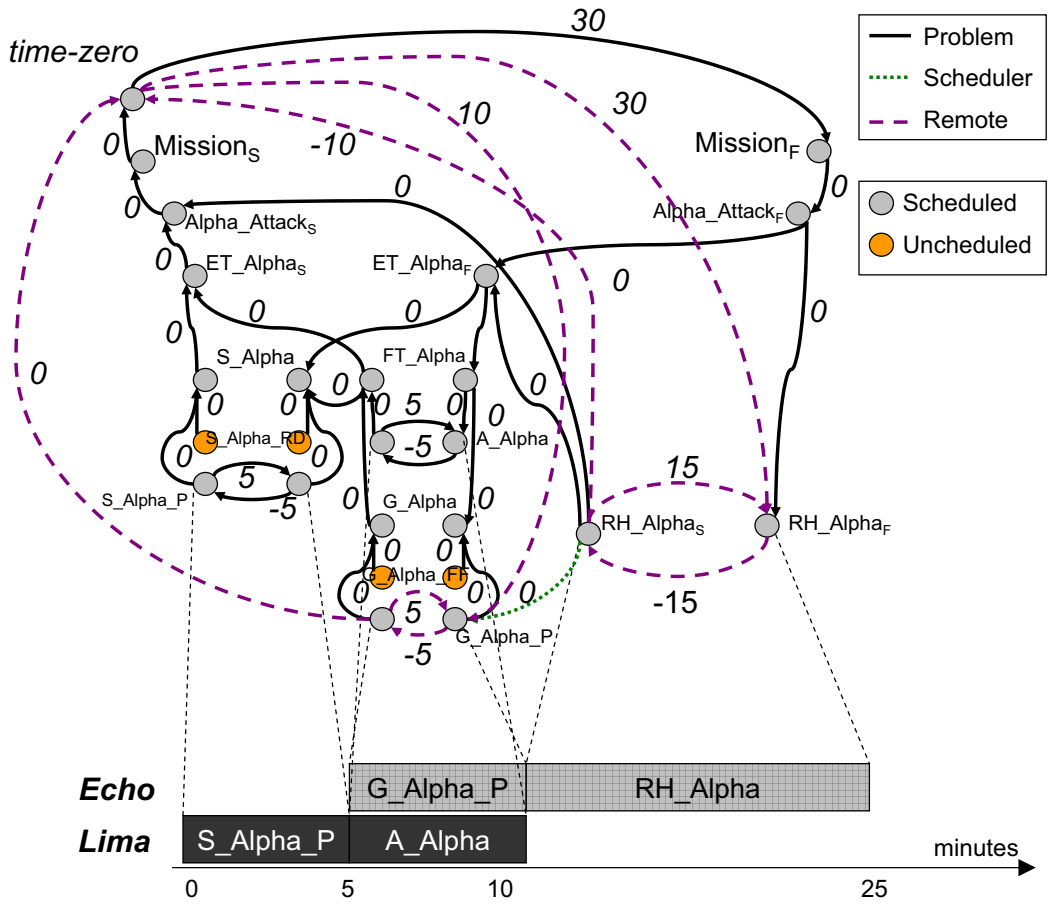

Figure 6.4: The STN and Schedule for Lima at the start of the mission.

2. *Scenario 2*: Lima relays the new situation to Tango. When Tango receives this updated information, it recognizes the danger to the mission. The slack in the original plan can accommodate Lima's extended bombardment of the insurgents in Alpha if it takes 10 minutes. However, if Lima takes 12 minutes to soften the resistance to an acceptable level for Echo to engage, Echo will no longer have enough time to rescue the hostages by the prescribed deadline, risking a hostile mob impeding its retreat. In the face of the new situation, Tango starts bombarding the insurgents. Lima continues the aerial bombardment of Alpha, but 7 minutes after starting engagement, a second helicopter gunship suffers damage due to continued anti-aircraft fire. Lima is forced to abort and withdraw. Nonetheless, 10 minutes after the start of the mission, Tango's sustained mortar rounds manage to subdue the insurgents, enabling Echo to proceed. Echo's

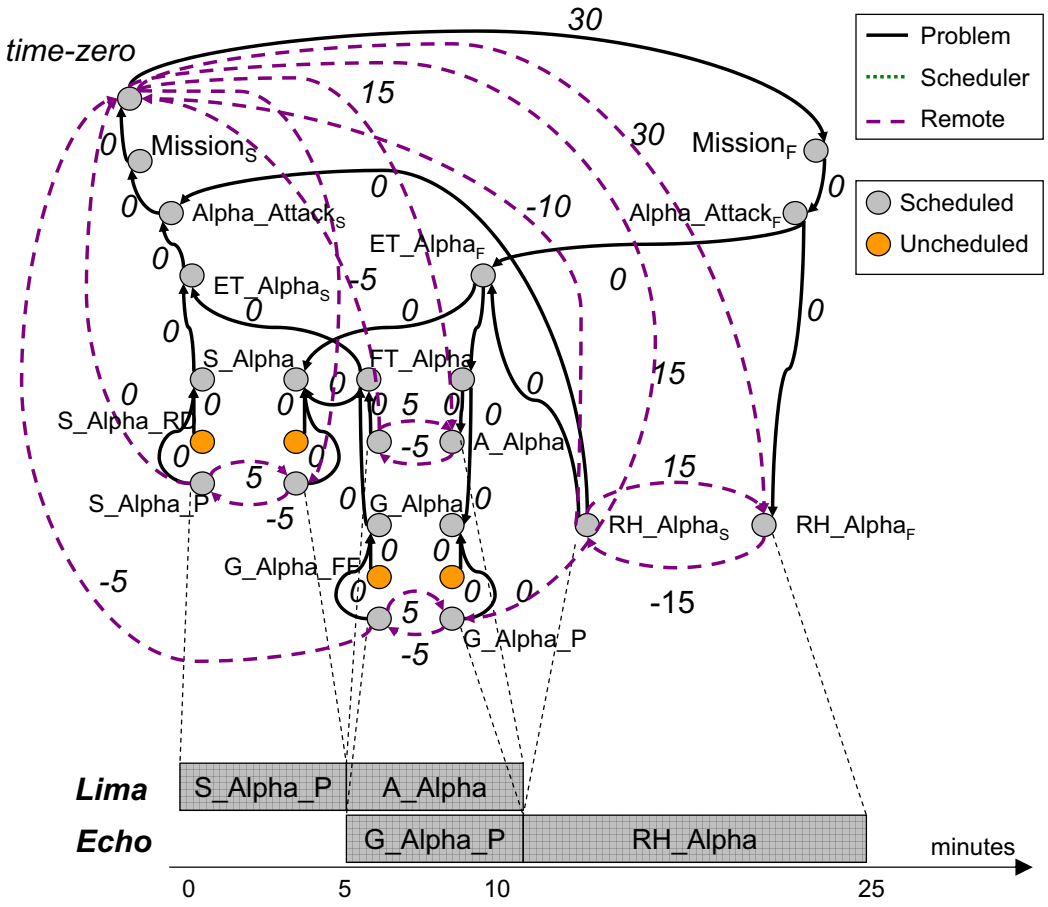

Figure 6.5: The STN and Schedule for Tango at the start of the mission.

STN and the current schedule of the team at minute 8 is shown in Figure 6.7.

Comparing these two scenarios illustrates the potential importance of redundant activities. In scenario 1, the mission failed after Lima was unable to complete its part of the mission. In scenario 2, on the other hand, Tango is able to take up the bombardment of Alpha from Lima, and enable the mission to proceed.

6.2.2 Determining the Horizon Activities

An agent using JIT-BF monitors the potential failures of its *horizon* activities. The *horizon* activities continuously change as execution progresses, and consist of the union of two sets, (1) the *Executing Set* and (2) the *On-Deck Set*. The *executing set* is composed of a single activity, the one currently executing (if any). The *on-deck*

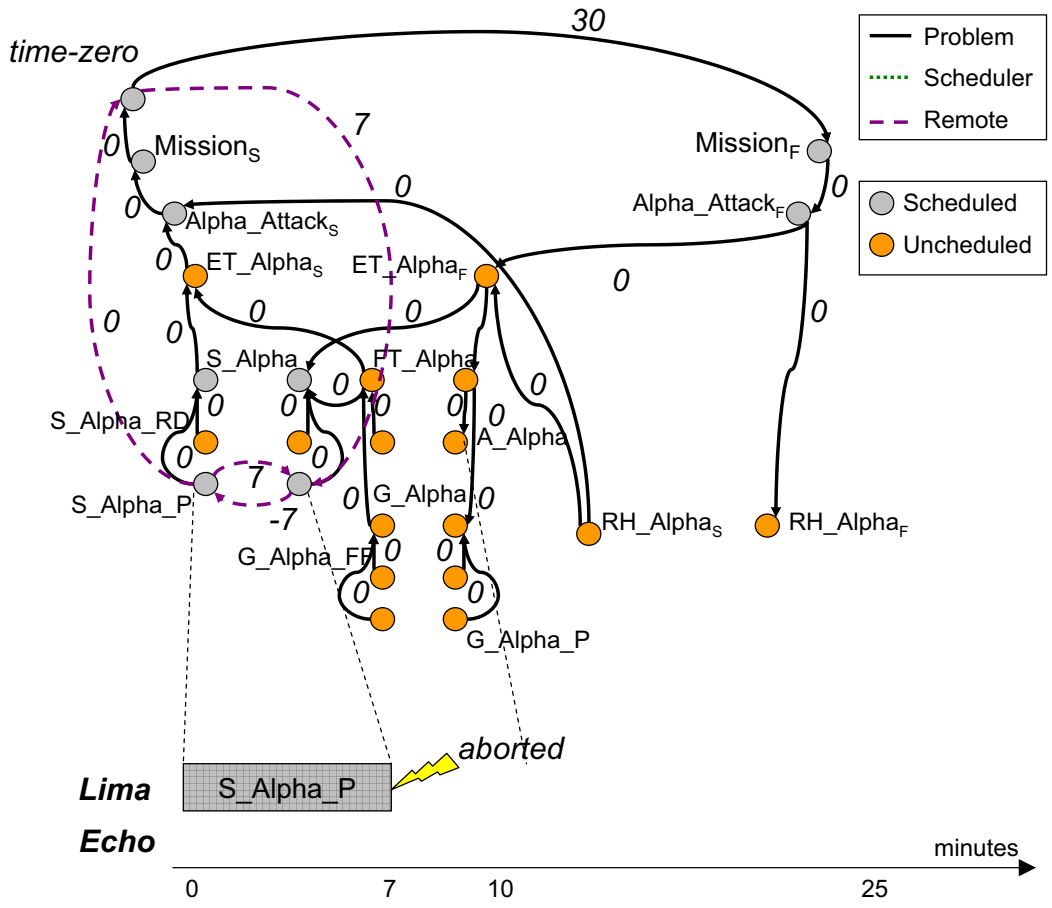

Figure 6.6: The STN and Schedule for Echo at minute 8.

set is composed of a subset of *pending* activities that are close to execution. This approach extends a previous technique [161] that labels a single *pending* activity (the one immediately follows the currently *executing* activity) as *on-deck* and attempts to increase its chances of successful execution.

The agent determines which *pending* activities belong in the *on-deck set* by using two parameters: $OD_{MaxActs}$ and $OD_{MaxTime}$. The first parameter, $OD_{MaxActs}$, limits the maximum number of activities in the *on-deck set*. The second parameter, $OD_{MaxTime}$, excludes from the *on-deck set* any *pending* activities whose *earliest start time (EST)* is greater than $now + OD_{MaxTime}$, where *now* is the current time. The *on-deck set (OD)* is given by the intersection of the sets obtained using these two parameters.

Figure 6.8 shows an example of the set of *horizon* activities (subdivided into

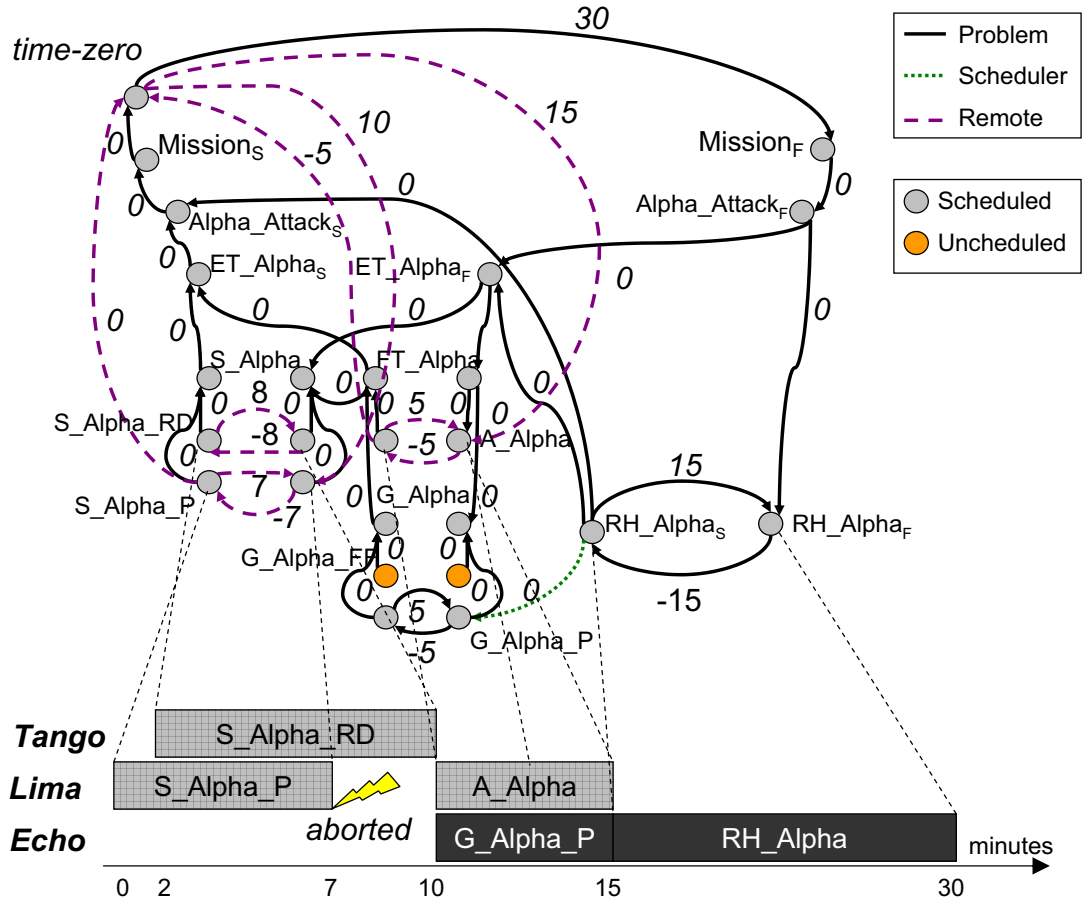

Figure 6.7: The STN and Schedule for Echo at minute 8.

the *executing set* and the *on-deck set*) for a section of agent's schedule consisting of five methods M_0, M_1, M_2, M_3 and M_4. For this example $OD_{MaxActs} = 3$ and $OD_{MaxTime} = 40$ minutes. The current time is 75 minutes. Four methods, M_1, M_2, M_3 and M_4 start within 40 minutes of the current time, falling within $OD_{MaxTime}$. However, given that $OD_{MaxActs} = 3$, the *on-deck* methods are M_1, M_2 and M_3. Since M_0 is executing, the *horizon* activities are M_0, M_1, M_2 and M_3.

Once the agent has determined the set of *horizon* activities, its next step is to find the activities in the *related set* of each *horizon* activity. This set includes the *horizon* activity itself and a subset of its predecessors [2]. The *related set* of a *horizon* activity corresponds to the activities whose durations are modified when calculating the likelihood that a *horizon* activity will fail. All activities excluded from the *related*

[2]The predecessors are the activities that are constrained to execute *before* the *horizon* activity.

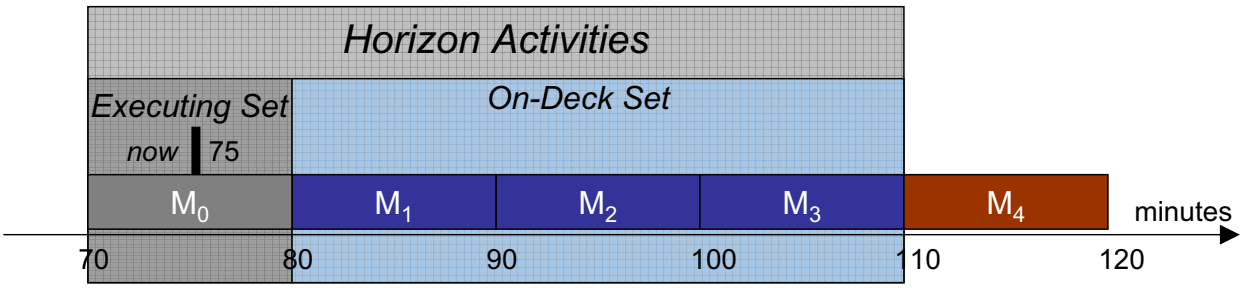

Figure 6.8: The Set of Horizon Activities

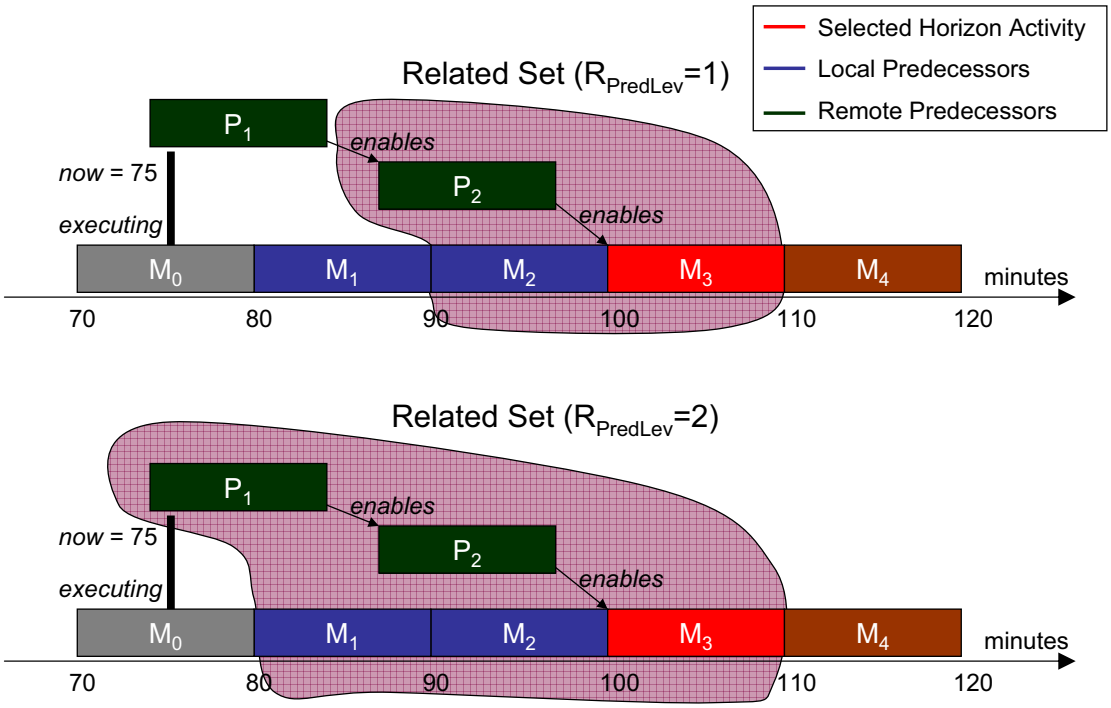

Figure 6.9: Related Set of a Selected Horizon Activity

set are assumed to last their expected durations.

The agent determines which predecessors to include in the *related set* of a *horizon* activity by specifying a desired *predecessor level* parameter, $R_{PredLev}$. Figure 6.9 illustrates the *related set* of an activity M_3. On the top, with $R_{PredLev} = 1$, the *related set* of M_3 includes M_3 and its immediate "first-level" predecessors: Those linked directly to M_3 by a precedence constraint. On the bottom, with $R_{PredLev} = 2$,

the *related set* of M_3 includes M_3, its immediate "first-level" predecessors, and the "second-level" predecessors: Those linked to the "first-level" predecessors of M_3.

The *related set* focuses the analysis of failure of a *horizon* activity on a subset of activities, consisting of the activity itself and those activities that are *closely related* to it. Activities outside the *related set* are assumed to have little or no impact on the activity. While this assumption does not hold in some cases, in practice, it generally does not affect how the agents benefit from the JIT-BF strategy. Activities outside the *related set* that can impact the failure potential of a *horizon* activity are *pending* indirect predecessors that fell outside of the *related set* because of the *predecessor level* considered. However, if a *horizon* activity h has unexecuted indirect predecessors outside of the *related set*, this indicates that its execution is not yet imminent. Analyzing the failure potential of h provides the agent with a rough look-ahead that can be used to safeguard h when it looks likely to fail. This analysis will get increasingly more accurate as h approaches execution and the predecessors of h outside its *related set* execute (and they have actual deterministic durations).

6.2.3 Notation Used

To improve clarity and conciseness, the rest of this chapter will use the following notations:

- $H(a)$: The set of *horizon* activities of in the schedule of an agent "a".
- $R(x)$: The *related set* of a *horizon* activity "x".
- $C(x)$: The set of *duration combinations* of the activities in $R(x)$ that lead to an inconsistency.
- $NC(x, dc)$: The set of negative cycles that were produced when asserting a *duration combination*, dc, of the activities in $R(x)$ into the STN [3].
- $q(x)$: The *quality* of an activity "x". Either the "expected" quality when the activity is *pending*, or the "actual" quality once it completes.
- $d(x)$: The *duration* of an activity "x", as asserted into the STN.

[3]The STN returns only one negative cycle at a time. All negative cycles that result when asserting the duration combination into the STN are resolved until the duration combination is successfully inserted. These negative cycles make up the $NC(x, dc)$ set.

- $d_{PDF}(x)$: The duration distribution of an activity "x". The duration distribution is specified in the format:

$$d_{PDF}(x) = ((d_1\ P(d_1))\ (d_2\ P(d_2))\ ...\ (d_N\ P(d_N)))$$

  Each pair in the distribution consists of (1) d_i, a possible duration of activity "x", and (2) $P(d_i)$, the probability that d_i will occur.

- $SSc(x)$: The set of *scheduler-defined* successors of an activity "x". These successors are the activities that the *scheduler* constrains to execute after "x".

- $PSc(x)$: The set of *problem-defined* successors of an activity "x". These successors are the activities that the *plan* constrains to execute after "x".

- $Sb(x)$: The set of *siblings* of an activity "x". These *sibling* activities are the *alternative* methods to "x".

- $q_{all}(x)$: The sum of the quality $q(x)$ of an activity "x", and the qualities of all its *problem-defined* successors, $PSc(x)$. This metric is defined recursively:

$$q_{all}(x) = q(x) + \sum_{i \in PSc(x)} q_{all}(i) \tag{6.1}$$

6.2.4 Discovering Schedule Weaknesses

Once the agent has identified the set of *horizon* activities (and their associated *related sets*), it can determine which of these activities are in danger of failing (e.g. by missing their deadline), or can cause their successors to fail. The agent leverages its STN in conjunction with the durational uncertainty model. Each *horizon* activity is examined for potential failures by locating the *duration combinations* of the activities in its *related set* that cause an inconsistency. The agent exploits the STN's constraint propagation tools in a *hypothetical* "what-if" search procedure: Each duration combination is inserted into the STN to find *what* would happen *if* the duration combination materializes during schedule execution. A potential inconsistency is discovered when infeasible duration combinations are rejected by the STN, producing a negative cycle.

To introduce the duration combinations of the *related set* activities into the STN, the duration distributions of all activities in the *related set* must be available. While C_TAEMS provides duration distributions for all methods in the plan, the duration

distributions of a higher level tasks need to be derived from their underlying scheduled methods. However, given their limited view of the plan activities (see section 2.2), agents cannot assume that they know all the methods under a task. Due to this limitation, combining all the duration distributions of all the underlying methods of a task into a "merged" duration distribution may not be possible. We overcome this problem by simply assigning to a task the duration distribution of its *earliest quality* method (i.e the method that provides the task with positive quality, *enabling* its successors to start). This duration distribution is part of the information that is transmitted by the agents' Distributed State Mechanism (see section 2.3.2 for further details on the DSM), so an agent can assume that this distribution is known if it is aware of a task (e.g. a predecessor to one of the agent's methods). In practice, this approximation provides reasonable results for problems where it is uncommon for several methods under a task to finish so closely together that their potential finish times overlap due to the uncertainty in their durations. The plans we have considered generally fit under this category. If more accurate task duration distributions were desired (or necessary), the DSM could be extended to propagate further duration distributions of the scheduled methods under a task, so that computation of a "merged" distribution is feasible. However, computing these merged distributions can have drawbacks: Given the larger the number of points in the merged duration distributions, the number of duration combinations increases, and can result in much higher computational overheads. A balance between the desired accuracy of the merged distribution and computational load needs to be struck.

We will use the motivating scenario we laid out in section 6.2.1 to illustrate how Echo introduces *hypothetical* negative cycles into its STN to discover potential weaknesses in the schedule. In our example, we assume that Echo is informed (2 minutes into the mission) that the new expected duration of the softening activity, *S_Alpha*, is 10 minutes, and it may take as long as 12 minutes. To keep the example simple, we assume that $OD_{MaxActs} = 1$, constraining the *on-deck set* to the *pending* activity scheduled to execute next, *G_Alpha_P*. Since Echo is not executing any activities at minute 2, the set of *horizon* activities is equal to the *on-deck set*. We also set $R_{PredLev} = 1$, which restricts the *related set* of *G_Alpha_P*, $R(G\_Alpha\_P)$, to include only "first-level" predecessors, so

$$R(G\_Alpha\_P) = \{S\_Alpha, G\_Alpha\_P\}$$

Setting the duration distribution of task *S_Alpha* to the duration distribution of its

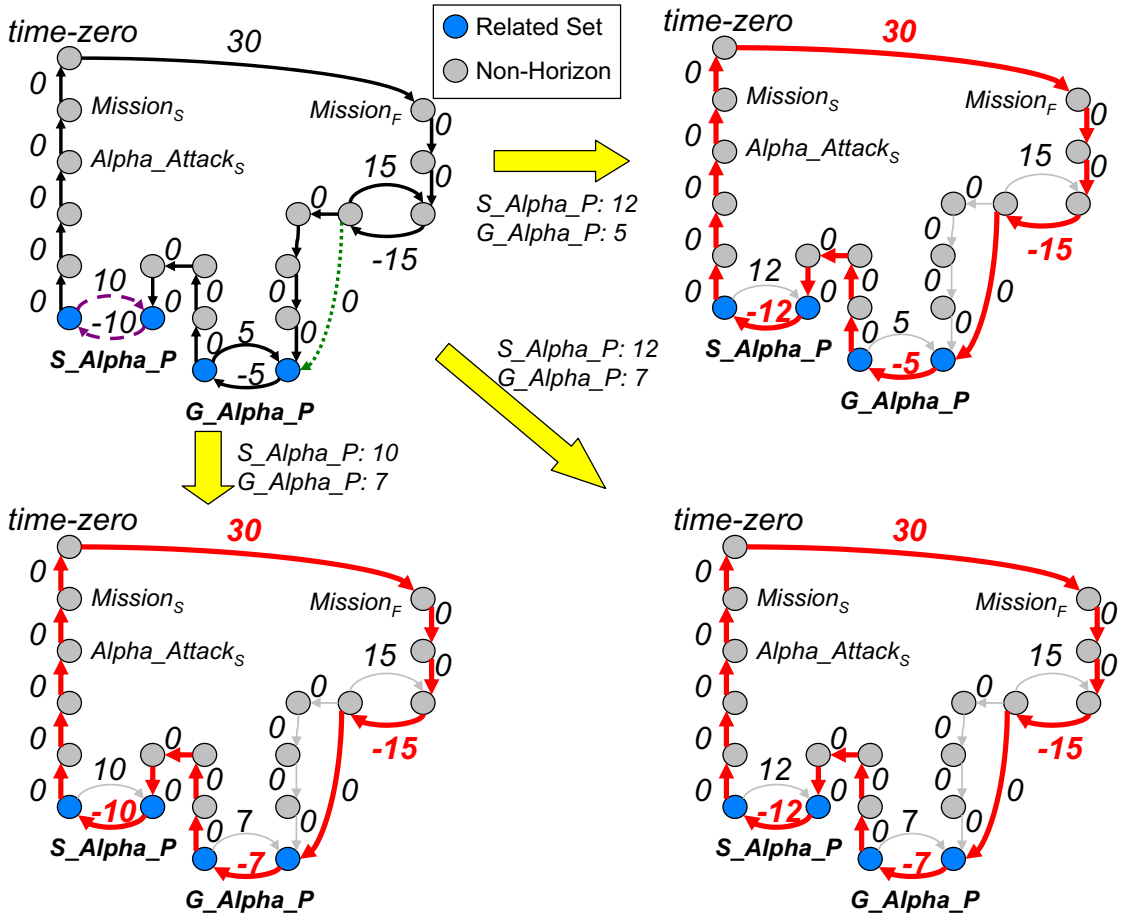

Figure 6.10: The Hypothetical Negative Cycles Produced by the Related Set of G_Alpha_P

earliest quality method *S_Alpha_P*, we have

$$d_{PDF}(S\_Alpha) = d_{PDF}(S\_Alpha\_P)$$

Echo's first step is to *hypothetically* insert the different potential duration combinations of *S_Alpha* and *G_Alpha_P* (as specified by their duration distributions) into the STN. For *S_Alpha_P* (and consequently *S_Alpha*)

$$d_{PDF}(S\_Alpha\_P) = ((8\ 0.25)(10\ 0.50)(12\ 0.25))$$

and for *G_Alpha_P*

$$d_{PDF}(G\_Alpha\_P) = ((3\ 0.25)(5\ 0.50)(7\ 0.25))$$

Given that we have 2 activities with 3 possible durations each, there are $3^2 = 9$ possible duration combinations of $d(S\_Alpha\_P)$ and $d(G\_Alpha\_P)$. Three of these potential combinations lead to an inconsistency:

1. When $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 5$ minutes.
2. When $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 7$ minutes.
3. When $d(S\_Alpha\_P) = 10$ minutes and $d(G\_Alpha\_P) = 7$ minutes.

Figure 6.10 shows the negative cycles that result after inserting these infeasible durations. The upper left corner shows (a partial view of) Echo's STN before inserting the possible duration combinations. The upper right corner shows the negative cycle when $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 5$ minutes. The bottom left corner shows the negative cycle when $d(S\_Alpha\_P) = 10$ minutes and $d(G\_Alpha\_P) = 7$ minutes, and the bottom right corner shows the negative cycle when $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 7$ minutes. The edges in the *hypothetical* negative cycles are shown in red.

An agent inserts a duration combination for the activities in the *related set* of a *horizon* activity into the STN by using a three-step procedure:

1. Initialize the durations of the activities to their minimum duration value (as specified in the duration distributions). This initialization is done to prevent spurious inconsistencies from occurring.

   To provide an example of how such spurious inconsistencies could arise, we'll examine what could happen if Echo does not initialize its activities to their minimum duration. Suppose we want to assert the duration combination $d(S\_Alpha\_P) =$ 12 minutes and $d(G\_Alpha\_P) = 3$ minutes. If we try to change *S_Alpha_P*'s duration to 12, but the duration of *G_Alpha_P* has been left at its expected value of 5, an inconsistency will occur. This inconsistency is spurious, since with the desired duration for *G_Alpha_P* of 3, there is not inconsistency.

2. Change the durations of the activities to their desired duration. This step involves modifying the *duration constraints* (enforcing the durations of the ac-

tivities in the STN) to reflect the desired durations. Modifying these *duration constraints* may introduce inconsistency.

3. If the introduction of the desired durations into the STN succeeded without inconsistencies, the process is finished. If, on the other hand, an inconsistency occurred, the resulting negative cycle returned by the STN is examined and *resolved* (the procedure for resolving the inconsistency is explained next). Once the inconsistency has been resolved, the process returns to the second step, to try again to insert the desired durations. This iteration continues until the duration combination is inserted successfully in the STN.

The agent collects all the *hypothetical* negative cycles returned by the STN during the process of inserting the desired duration combination for future analysis.

Resolving Hypothetical Inconsistencies

When the introduction of a possible duration combination of the *related set* activities results in an inconsistency (flagged by a negative cycle in the STN), this *hypothetical* inconsistency must be resolved before re-attempting to introduce this duration combination. To overcome the *hypothetical* inconsistency, we take similar steps to those we developed in Chapter 5 to resolve *real* inconsistencies (those that are caused by execution events). The conflict is first *explained* by extracting from the *negative cycle* returned by the STN the set of domain-level activities and constraints that are in the conflict. Then, this explanation is used to *resolve* the conflict. Given that the goal in resolving *hypothetical* conflicts is to understand what activities in the schedule would be affected if the inconsistency were to arise during execution, rather than formulating an updated *consistent* schedule, the resolution techniques we use are different. In general, we can classify *hypothetical* inconsistencies into two categories:

1. *External*: The negative cycle involves a *precedence* constraint (i.e. an *enables* NLE, or a scheduler imposed *sequencing* constraint) linking a predecessor activity within the *related set* to a successor activity outside of it. While the predecessor activity will finish before its (problem-defined) deadline, a subset of its successors are no longer feasible. We resolve these conflicts by *retracting* the precedence constraint to the affected successor activity.

   All three *hypothetical* negative cycles shown in Figure 6.10 are *external*, since they involve a precedence constraint between a *horizon* activity (*G_Alpha_P*),

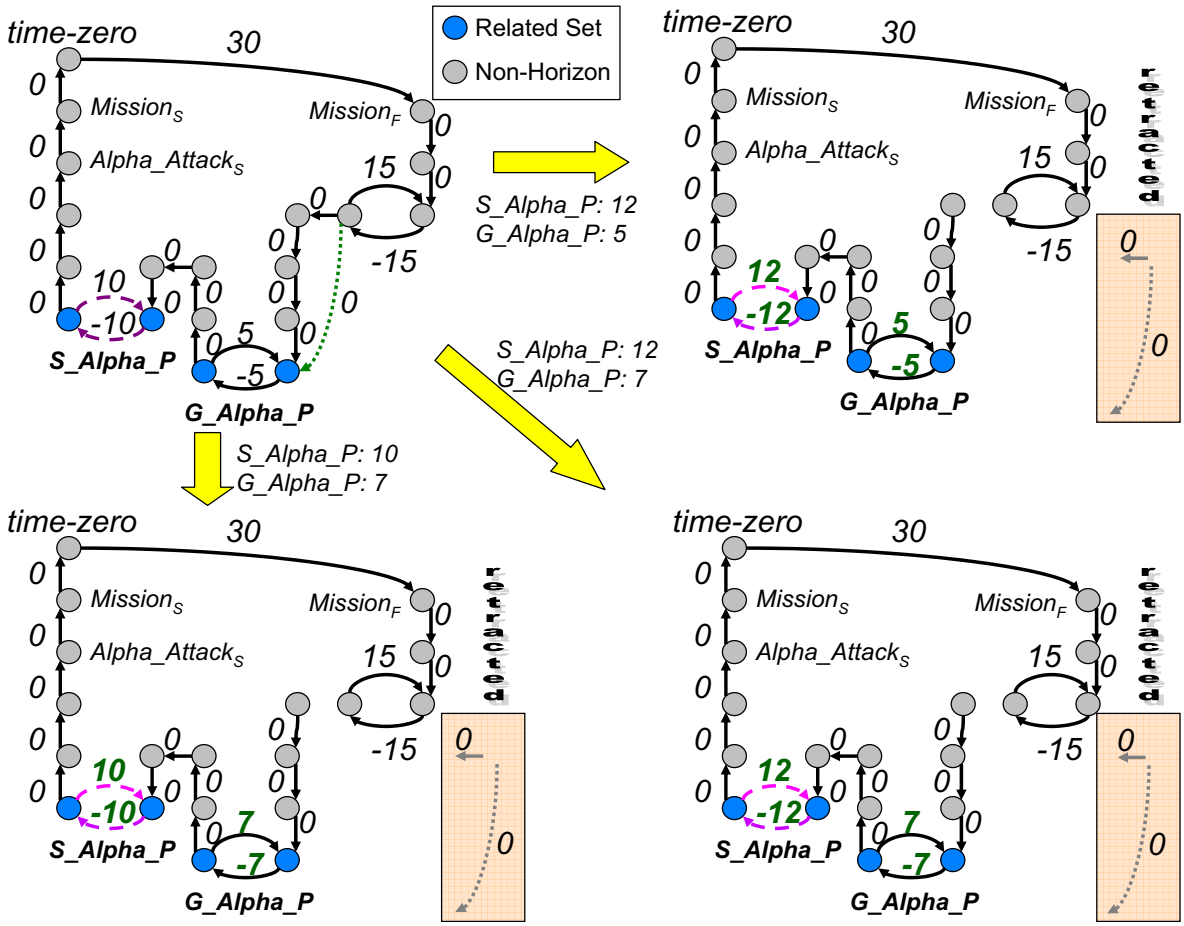

Figure 6.11: Resolving the Hypothetical Inconsistencies

and a successor activity (RH_Alpha). Figure 6.11 shows Echo's STN after each cycle is resolved. As before, the upper right corner shows the situation when $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 5$ minutes. The bottom left corner shows the duration combination $d(S\_Alpha\_P) = 10$ minutes and $d(G\_Alpha\_P) = 7$ minutes, and the bottom right corner shows the duration combination $d(S\_Alpha\_P) = 12$ minutes, and $d(G\_Alpha\_P)7$ minutes. Note that by retracting the precedence constraint between G_Alpha_P and RH_Alpha, there is no longer a constraint in the STN that enforces that RH_Alpha must start after G_Alpha_P, leading to a potentially infeasible schedule. However, our intent in this operation is to determine the affected activities in the schedule, not to enforce schedule feasibility.

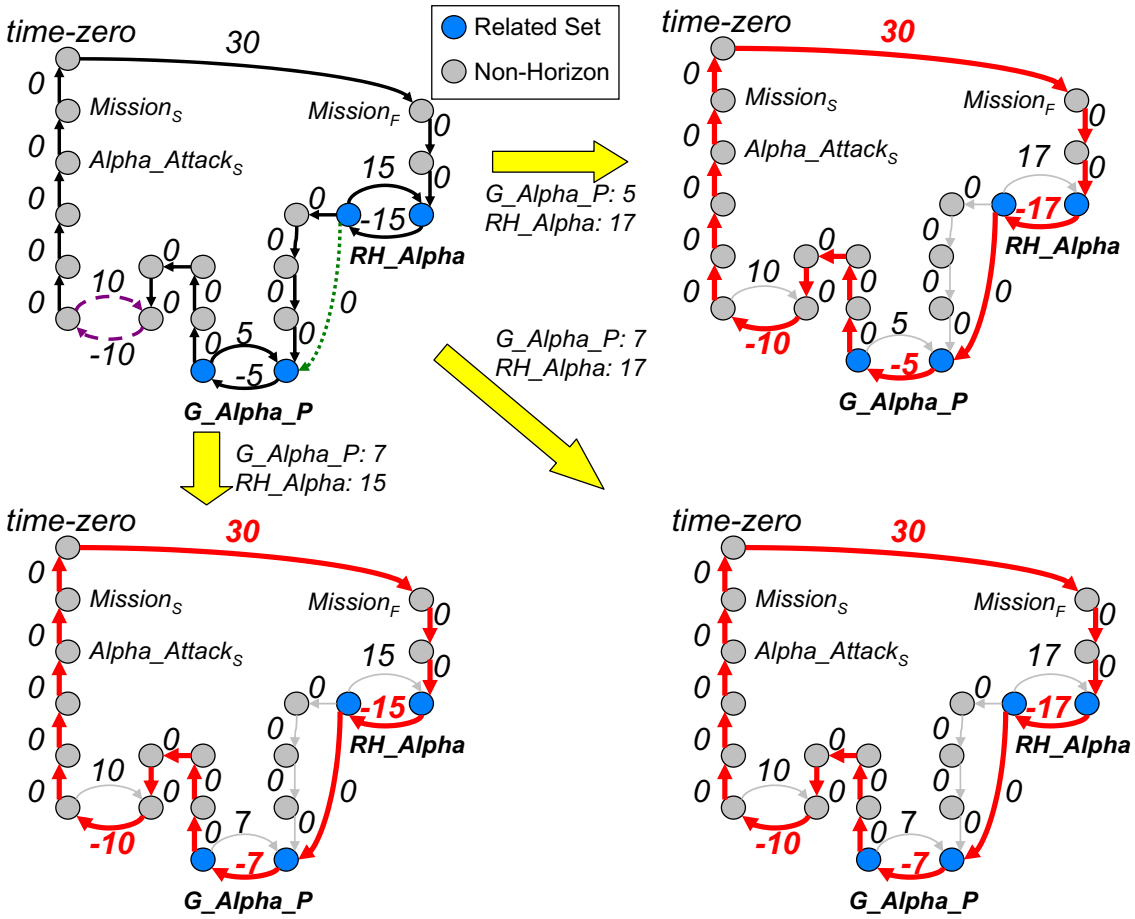

Figure 6.12: The Hypothetical Negative Cycles Produced by the Related Set of R_Alpha

2. *Internal*: The negative cycle involves the deadline constraint of an activity in the *related set*. This implies that the duration we are inserting would cause this activity (a member of the related set) to *fail* its deadline. We resolve these conflicts by *retracting* the deadline constraint from the STN.

   An example of an internal conflict can be construed by making a small modification to the *horizon* activities used in the previous example: Setting the parameter $OD_{MaxActs} = 2$, the *horizon* activities in Echo's schedule change to include *RH_Alpha* (i.e the set of *horizon* activities now is {*G_Alpha_P*, *R_Alpha*}). Using $R_{PredLev} = 1$ as before, the *related set* of the "new" horizon method *R_Alpha*

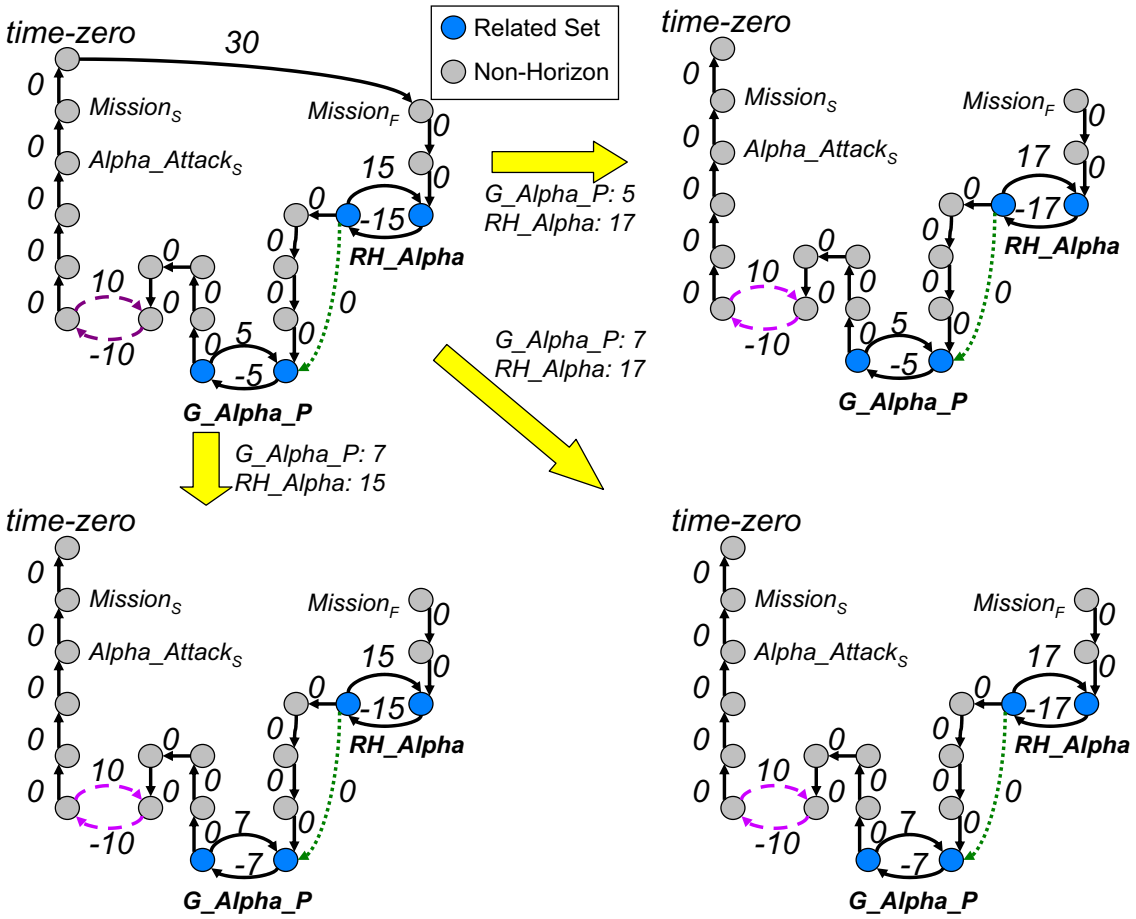

Figure 6.13: Resolving the Hypothetical Inconsistencies

is given by

$$R(G\_Alpha\_P) = \{G\_Alpha\_P,\ R\_Alpha\}$$

With

$$d_{PDF}(R\_Alpha) = ((13\ 0.25)(15\ 0.50)(17\ 0.25))$$

the insertion of the different potential combinations of G_Alpha_P and R_Alpha produce three inconsistencies:

(a) When $d(G\_Alpha\_P) = 7$ minutes and $d(R\_Alpha) = 15$ minutes.

(b) When $d(G\_Alpha\_P) = 7$ minutes and $d(R\_Alpha) = 17$ minutes.

(c) When $d(G\_Alpha\_P) = 5$ minutes and $d(R\_Alpha) = 17$ minutes.

The *hypothetical* negative cycles produced by inserting these *inconsistent* duration combinations are shown in Figure 6.12. All negative cycles involve the deadline constraint on *RH_Alpha*, which this time is within the *related set*. These three cycles can be resolved by retracting *RH_Alpha*'s deadline, as shown in Figure 6.13.

6.2.5 Analyzing Hypothetical Inconsistencies to Discover Effect on Schedule

The agent determines the likelihood of failure of each *horizon* activity, and how the failure would affect the schedule, by analyzing the *hypothetical* negative cycles associated with these activities. A four-step process computes the likelihood of failure for each horizon activity, and determines how their failure would affect the schedule:

1. Identifying the *relevant* negative cycles: Not all conflicts are relevant for the agent's failure analysis. For example, conflicts that involve soft NLEs do not really lead to a failure and can be discarded.

2. Analyzing the relevant *hypothetical* negative cycles: An infeasible duration combination can result in one or more *hypothetical* conflicts [4]. Each negative cycle produced by the STN when asserting a duration combination is analyzed using *conflict explanation* techniques (see Section 2.5.2) to determine how the schedule is affected by this inconsistency.

3. Aggregating the analyses of all inconsistencies related to a duration combination: A combined analysis of the effects of an infeasible duration combination is obtained by aggregating the effects of all the negative cycles that resulted when asserting the duration combination into the STN. In conjunction with the analysis of the effects of the duration combination, the agent determines the likelihood that this duration combination will occur by using the durational uncertainty model. This likelihood is computed as the *conditional* probability that

[4] Recall that several inconsistencies may arise while inserting a single duration combination into the STN. For example, the duration combination may be inconsistent with (1) the deadline of the *horizon* activity, and with (2) a pending activity scheduled to execute later. Two distinct negative cycles will be produced for each of these inconsistencies.

the duration combination of the activities in the *related set* will occur, *assuming* that the rest of the scheduled activities execute to their expected duration.

4. Computing a failure analysis for each *horizon* activity: The agent aggregates the analyses of failures for all infeasible duration combinations of the *related set* of a *horizon* activity into a combined analysis of failure.

The next sections examine each of these four steps in turn, and provide further details.

Determining Relevant Hypothetical Inconsistencies

The agent determines *which* of the *hypothetical* negative cycles encountered while inserting the duration combinations (of the *horizon* activity's *related set*) need to be considered for further analysis by categorizing these *hypothetical* conflicts into two sets, *relevant* and *irrelevant*, depending on the constraint that was retracted to resolve it:

1. *Relevant Conflicts*: These are the conflicts that need to be analyzed further. There are two types of relevant conflicts:

   - *Internal* conflicts involving the deadline constraint of the *horizon* activity, or one of its *enablers* (i.e. a predecessor as defined by an enables NLE). A violation of one of these deadline constraints indicates that either the *horizon* activity or its *enabler* fails to meet its deadline. In our problem domain, deadlines are *inflexible*: An activity that violates its deadline accrues no quality. Consequently, if either the *horizon* activity or one of its *enablers* fails to meet its deadline, the *horizon* activity will not accrue quality.
   - *External* conflicts involving an *enables* NLE linking the *horizon* activity to a later *pending* activity. These conflicts indicate that the *horizon* activity has exceeded its LFT (see section 2.1 for explanation of the temporal bounds of an activity), and is pushing on *pending* activities further down the schedule causing them to fail their deadlines.

2. *Irrelevant Conflicts*: These are conflicts that we do not consider for further analysis. These conflicts include:

- *Internal* conflicts involving the deadline constraint of a *scheduler-defined* predecessor of the *horizon* activity. This deadline violation causes the *scheduler-defined* predecessor activity to fail. However, the *horizon* activity is not affected by this failure, and can still proceed to execute and accrue quality.
- *External* conflicts involving a *disables* NLE. *Disables* relations do not lend themselves well to the robust scheduling techniques we develop in this chapter.

  An inconsistency involving a *disables* NLE that links a source activity A_1 to a target activity A_2 indicates that A_1 completed *earlier* than expected, *disabling* A_2. We have a limited range of actions to forestall such failures: Scheduling a shorter fall-back or redundancy worsens the problem, since A_1 would accrue quality even earlier. We instead rely on the *recovery* strategies outlined in chapter 5 to overcome inconsistencies involving disables NLEs, focusing our robust scheduling strategies on *enables* NLEs, which model the more traditional *precedence* relations.

Irrelevant negative cycles are discarded from further consideration. The negative cycles deemed *relevant* are analyzed to discover their effect on the schedule, were the duration combination that produced them occur. This analysis is presented in the next section.

Analyzing a Hypothetical Negative Cycle

Each *relevant* negative cycle, c, is *explained* (see description of STN conflict explanation techniques in Section 2.5.2), and its effect on the schedule determined. The agent reduces this analysis into a measure of the impact of the conflict to the schedule:

- $Q_{Lost}(c)$: The quality that would be potentially lost if the *related set*'s duration combination happens, causing the conflict, c, to occur.

The agent computes the potential quality lost if the duration combination occurs, $Q_{Lost}(c)$, by analyzing the temporal constraints and activities involved in the *hypothetical* conflict, mining the negative cycle for the activity that *fails* its deadline [5].

[5]Our scheduling agent implementation, the cMatrix agent (see section 2.3), may not always be aware of the deadlines of *remote* activities. However, the *temporal bounds* are known. If the deadline of a remote activity is unknown, we assume its LFT is its deadline.

This *failed* activity, A_{failed}, is either the *horizon* activity, h, or one of its successors, direct or indirect (e.g. the successor of h's successor). We find A_{failed} by recursively exploring the *precedence chain* that starts with h, and iterating through its successors until we find the activity that fails its deadline. Once the *failed* activity, A_{failed}, has been found, we compute the quality loss by using the equation

$$Q_{Lost}(c) = q_{all}(A_{failed}) \tag{6.2}$$

which sums the quality lost by the activity itself, $q(A_{failed})$, and the quality lost by all the activities that depended on its successful completion, $\sum_{i \in PSc(A_{failed})} q(i)$. This quality loss equation provides an estimate of how conflict c would affect the schedule, were it to occur.

We use this quality loss estimate in lieu of computing the exact quality loss to the schedule, because computing the exact loss is not possible in the general case. An exact measurement of the quality lost by the schedule, in the event c occurs, involves propagating the quality losses of the individual activities (A_{failed} and its successors) up the activity hierarchy to compute the effect on the *taskgroup* (the root activity, see section 2.2.1). However, given the agents' *limited* view of the plan activities in our problem domain, an agent cannot, in the general case, assume that it has enough information to compute this exact lost quality. This situation occurs because an agent's view may not incorporate the ancestor tasks of a remote successor activity. In this case, the agent is unable to calculate exactly, how the loss of this successor activity would affect the quality of the taskgroup.

Figure 6.14 shows an example of how an agent can mine the *hypothetical* negative cycle returned by the STN to locate the *failed* activity, and the calculate the cost to the schedule of the *potential* failure. The left side of the figure figure shows a *horizon* activity M_1, followed by two *pending* methods on the timeline, M_2 and M_3. To simplify the example, M_1 has no predecessors, so its *related set* is $R(M_1) = \{M_1\}$. As shown, the expected duration of M_1 is $d(M_1) = 10$ and its duration distribution is

$$d_{PDF}(M_1) = ((8\ 0.25)(10\ 0.5)(12\ 0.25))$$

The right side of the figure shows the *hypothetical* negative cycle that ensues when we attempt to insert the *potential* duration, $d(M_1) = 12$. An analysis of the inconsistency, c, shows that when $d(M_1) = 12$, M_2 and M_3 are pushed later in the timeline, and M_3

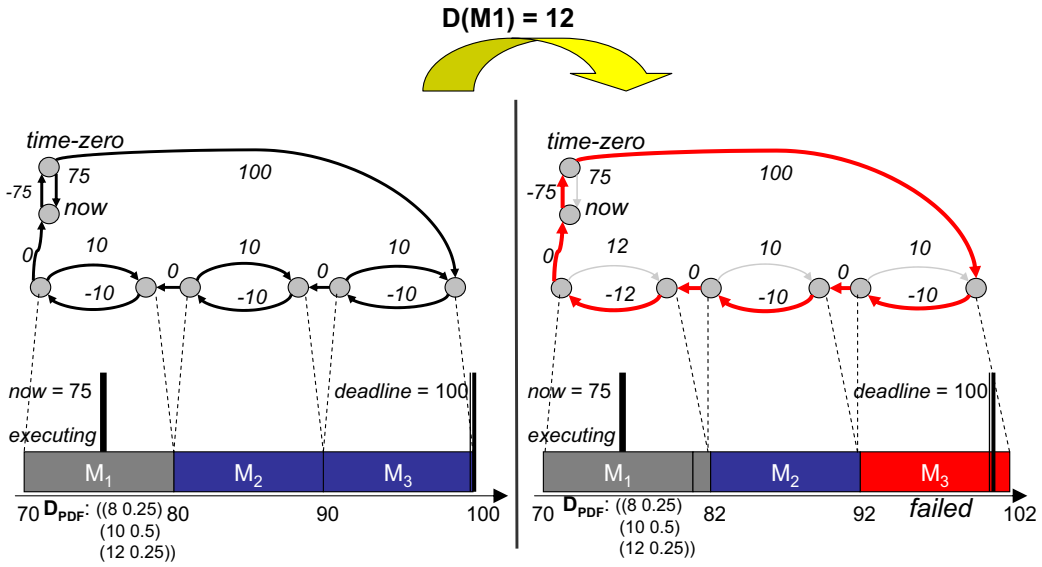

Figure 6.14: A Hypothetical Negative Cycle

fails its deadline. M_3 is the *failed* activity, and $Q_{Lost}(c) = q_{all}(M_3) = q(M_3)$.

Aggregating a Horizon Activity's Hypothetical Inconsistencies' Effects

After the agent has analyzed each of the "relevant" negative cycles, it aggregates these analyses into a combined analysis of failure for each *horizon* activity. The analyses aggregation process for a *horizon* activity, h, consists of two steps:

1. First, the agent forms aggregated analyses for each infeasible *duration combination*, dc, in $R(h)$ (h's *related set*) [6]. This process involves combining the analyses of all the negative cycles that occurred when attempting to assert dc into an aggregated analysis that foretells the effect on the schedule if dc materializes during execution.

2. Next, the agent aggregates the analyses of all the infeasible duration combinations of $R(h)$ into a combined *failure* analysis for the *horizon* activity.

Each of these two steps will be now examined in greater detail.

[6]Remember that the STN may produce multiple negative cycles when attempting to assert an infeasible duration combination.

(1) Aggregating Failure Analysis of a Duration Combination In this step, the agent combines the analyses of all *hypothetical* inconsistencies that occur when attempting to assert an infeasible duration combination, dc, of $R(h)$. The combined analysis is reduced into two components:

1. $P(dc)$: The probability that the duration combination, dc, of the activities in $R(h)$ will materialize during execution.

2. $Q_{Lost}(dc)$: The quality that would be lost if dc occurs. This quality is obtained by adding the quality lost by each of the inconsistencies that arise when asserting dc

$$Q_{Lost}(dc) = \sum_{c \in NC(h,dc)} Q_{Lost}(c) \tag{6.3}$$

   where $NC(h, dc)$ is the set of *hypothetical* negative cycles that the STN produces when inserting the duration combination dc for the activities in $R(h)$.

We can illustrate how these two quantities are obtained using the motivating scenario laid out in Section 6.2.4. With the *horizon* activity *G_Alpha_P*, the *related set* is

$$R(G\_Alpha\_P) = \{S\_Alpha,\ G\_Alpha\_P\}$$

Given the duration distributions of the activities in $R(G\_Alpha\_P)$

$$d_{PDF}(S\_Alpha\_P) = ((8\ 0.25)(10\ 0.50)(12\ 0.25))$$
$$d_{PDF}(G\_Alpha\_P) = ((3\ 0.25)(5\ 0.50)(7\ 0.25))$$

three duration combinations lead to inconsistencies: (1) When $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 5$ minutes, (2) when $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 7$ minutes, and (3) when $d(S\_Alpha\_P) = 10$ minutes and $d(G\_Alpha\_P) = 7$ minutes.

Let's examine the duration combination, dc_1, where $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 5$ minutes. In this case, dc_1 leads to a single *hypothetical* inconsistency shown to the upper right of Figure 6.10. The probability that dc_1 will

occur is

$$\begin{aligned} P(dc_1) =& P\{d(S\_Alpha\_P) = 12,\ d(G\_Alpha\_P) = 5\} = \\ & P\{d(S\_Alpha\_P) = 12\} \times P\{d(G\_Alpha\_P) = 5\} = \\ & 0.25 \times 0.50 = 0.125 \end{aligned}$$

and the quality lost is

$$Q_{Lost}(dc_1) = q(RH\_Alpha) = 30$$

Similarly, for the other two duration combinations (a) dc_2 where $d(S\_Alpha\_P) = 12$ minutes and $d(G\_Alpha\_P) = 7$ minutes, and (b) dc_3 where $d(S\_Alpha\_P) = 10$ minutes and $d(G\_Alpha\_P) = 7$ minutes, their failure analyses are

$$\begin{aligned} P(dc_2) &= 0.0625 \\ Q_{Lost}(dc_2) &= q(RH\_Alpha) = 30 \end{aligned}$$

and

$$\begin{aligned} P(dc_3) &= 0.125 \\ Q_{Lost}(dc_3) &= q(RH\_Alpha) = 30 \end{aligned}$$

While the previous example showed infeasible duration combinations that lead to a single *hypothetical* inconsistency, we now present an example of a duration combination that causes two negative cycles. Figure 6.15 expands the example in Figure 6.14. As before, the *horizon* activity M_1 has no predecessors, so $R(M_1) = \{M_1\}$. However, besides *scheduler-defined* successors (M_2 and M_3), M_1 is also has a *problem-defined* successor, E_1 (i.e. E_1 follows M_1 due to an *enables* NLE). Therefore, $PSc(M_1) = \{E_1\}$. For $R(M_1)$, there is a single infeasible duration combination, dc_{M_1} that occurs when asserting M_1's extended duration, $d(M_1) = 12$ [7]. This duration combination causes two inconsistencies:

1. c_1: A conflict involving the deadline of M_3 (shown above the schedule).

[7] Although it may seem awkward to talk of a duration combination when there is only one activity in the *related set*, we continue using this terminology as we have defined a duration combination as the durations that activities in the *related set* take.

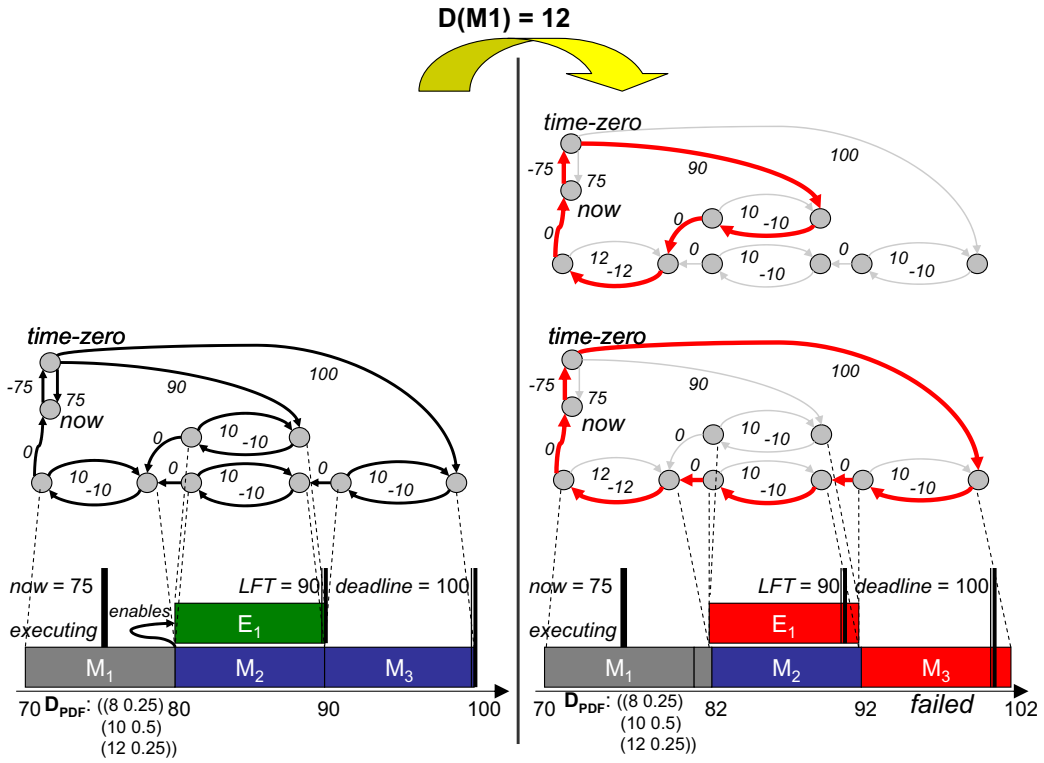

Figure 6.15: Two Hypothetical Negative Cycles

2. c_2: A conflict involving the deadline of E_1 (shown above the first negative cycle c_1).

The STN produces two *hypothetical* negative cycles for each of these inconsistencies [8]. The probability that dc_{M_1} will materialize duration execution is

$$P(dc_{M_1}) = P\{d(M_1) = 12\}$$

and the quality lost due to dc_{M_1} can be obtained by summing the quality lost by c_1 and c_2

$$Q_{Lost}(dc_{M_1}) = Q_{Lost}(c_1) + Q_{Lost}(c_2)$$

[8]The negative cycles are produced sequentially. The order in which the negative cycles are produced depends on the conflict analysis algorithms implemented by the STN. In our example, we assume that the conflict with M_3 is flagged first, and once it is resolved, the conflict with E_1 is flagged. After resolving this second conflict, the assertion of M_1's extended duration, $d(M_1) = 12$ succeeds.

(2) Aggregating Failure Analysis of a Horizon Activity In this last step, the agent computes a combined failure analysis for each *horizon* activity, h, by aggregating the analyses of the infeasible duration combinations of the activities in $R(h)$. This combined failure analysis is reduced to two components:

1. $P_{Fail}(h)$: The probability that h will encounter an inconsistency when it executes. This probability is simply the sum of the probabilities of the duration combinations that cause inconsistencies:

$$P_{Fail}(h) = \sum_{dc \in C(h)} P(dc) \tag{6.4}$$

   where $C(h)$ is the set of duration combinations in $R(h)$ that cause an inconsistency.

2. $WQ_{Lost}(h)$: The expected quality loss of h. This value can be obtained by summing the expected quality loss of each duration combination in $C(h)$:

$$WQ_{Lost}(h) = \sum_{dc \in C(h)} P(dc) \times Q_{Lost}(dc) \tag{6.5}$$

   The expected quality loss simply weighs the lost quality of an infeasible duration combination with the probability of its occurrence.

We will illustrate how an agent computes this aggregated analysis using the motivating scenario we outlined in section 6.2.4. $P_{Fail}(G\_Alpha\_P)$, the probability of failure of Echo's *horizon* activity *G_Alpha_P* is obtained by summing the probabilities of the three infeasible duration combinations of $R(G\_Alpha\_P)$ (shown in the previous section). Consequently,

$$P_{Fail}(G\_Alpha\_P) = 0.125 + 0.0625 + 0.125 = 0.3125$$

Finally, the expected quality loss for *G_Alpha_P* can be obtained by

$$WQ_{Lost}(G\_Alpha\_P) =$$
$$P(dc_1) \times Q_{Lost}(dc_1) \; + \; P(dc_2) \times Q_{Lost}(dc_2) \; + \; P(dc_3) \times Q_{Lost}(dc_3) =$$
$$0.125 \times 30 \; + \; 0.0625 \times 30 \; + \; 0.125 \times 30 = 9.375$$

completing the analysis of failure of the *horizon* activity.

6.2.6 Backfilling To Strengthen the Schedule

We now turn our attention towards the *pro-active* steps that can be taken to reduce or eliminate the possibility of failure, based on the failure analysis of the *horizon* activities. Our initial step, before taking any *pro-active* actions is to determine if any *horizon* methods are *endangered*. A method, h, is classified as *endangered* if its probability of failure, $P_{Fail}(h)$, is greater than a specified threshold probability, $F_{Threshold}$:

$$P_{Fail}(h) > F_{Threshold} \tag{6.6}$$

If *endangered* horizon methods have been detected, the agent takes *pro-active* actions to prevent or diminish the effects of the potential failure. We have developed two strategies that increase the fault-tolerance of a multi-agent by scheduling (or "backfilling) *alternative* methods:

1. *Passive*: This technique enables the agents to backfill the schedule with "redundancies" to the endangered methods. It involves two steps: First, an agent with an *endangered horizon* method, h, informs other agents that "own" redundant methods to h that it is *endangered*. Then, recipient agents use this information to evaluate whether they can provide assistance to the requesting agent. A recipient agent only schedules the "redundant" method if the decrease in flexibility caused by this addition does not cause any of its own *horizon* methods to become *endangered*.

2. *Aggressive*: This technique enables an agent with an *endangered* method, h, to compel changes to another agent's schedule (sometimes its own). This aggressive strategy kicks in only when passive requests for help have failed to schedule any redundant methods to h. The *requesting* agent (h's owner) polls the owners of *alternative* methods for assistance (sometimes including itself). The *recipient* agents generate schedules that include the *alternative* method they own, even if scheduling this *alternative* method leads to a quality loss to their schedules. The *requesting* agent then evaluates the responses and decides whether one of these *alternative* methods should be scheduled, and notifies the owner of this method.

Passive Robust Scheduling

The passive backfilling strategy provides a mechanism that an agent with an *endangered horizon* method h can use to request other agents that "own" redundant methods to h methods to schedule them. The recipient agents decide whether to schedule the "redundant" method they "own", based on how their *horizon* methods are affected. In the rest of the section, we describe the actions of both the *requesting* agent (the agent that owns the *endangered* method), and the *recipient* agents (the agents that receive the request for help).

(1) Requesting Help for an Endangered Method An agent that owns an *endangered horizon* method *requests* help from other agents that own redundant methods by "piggy-backing" on its *Distributed State Mechanism (DSM)* (see Section 2.3 for a description of the cMatrix agent's architecture). The DSM is in charge of propagating updated information about the agent's activities to other team members. To provide support for help requests on behalf of an *endangered* method, h, we have extended the information managed by the DSM (the "state") to include the probability of failure of the activity, $P_{Fail}(h)$, and its potential quality loss, $WQ_{Lost}(h)$.

(2) Responding to Requests for Help When an agent receives the request for help, it attempts to schedule a *redundant* activity to the *endangered* method, h, that could prevent the failure of the parent *cnode*. Our goal when scheduling *redundant* activities is to prevent failure, rather than maximizing the quality accrued by the *cnode*, so the shortest duration *redundant* method is chosen for scheduling.

Before it commits to executing a *redundant* method, the *recipient* agent needs to determine the effects of this addition to its schedule. The key to this decision is the effect of adding the *redundant* method on the *recipient* agent's *horizon* activities. To measure this effect, we compare the number of "safe" *horizon* methods *before* and *after* adding the *redundant* method to the schedule. We define a method as "safe" if it has a probability of failure lower than the failure threshold $F_{Threshold}$. The set of "safe" *horizon* methods is given by:

$$H_{safe}(a) = \{m \in H(a) \mid P_{Fail}(m) < F_{Threshold}\} \tag{6.7}$$

where $H(a)$ is the set of *horizon* activities in agent a's schedule, and $H_{safe}(a)$ are the

"safe" *horizon* activities. A "redundant" method is deemed to have a detrimental effect on the schedule if the set $H_{safe}(a)$ contains fewer methods *after* adding the *redundant* method to the schedule than it did *before*. This indicates that adding the "redundant" method to the schedule increases the failure potential of one or more of the *horizon* methods so that it is no longer "safe". The *recipient* agent only keeps *redundant* methods on its schedule if they do not produce detrimental effects. Otherwise, the *redundant* methods are rejected, *even* if the expected quality loss of the *endangered* method h is greater than the expected quality loss of the newly "unsafe" *horizon* method. The rationale behind this rejection is that in the *passive* strategy, several *recipient* agents potentially can asynchronously schedule their redundant methods at the same time. If the *recipient* agents are allowed to *endanger* their local methods to help the *endangered* method of the *requesting* agent, the additional expected quality loss of the *recipient* agents could potentially exceed the expected quality loss of the *requesting* agent's *endangered* method. Since a *recipient* agent cannot determine whether to schedule its *redundant* method is advantageous when its addition causes an increase in the expected quality loss of the *horizon* methods, a conservative policy is chosen, and the *redundant* method is rejected.

Figure 6.16 illustrates a *recipient* agent (B in the figure) deciding whether to add a *redundant* method (M_{RD} in the figure) to its schedule. Agent A has requested help from agent B by informing it that its method M_P has a high probability of failure. Agent B now has to decide whether to acquiesce to this request and schedule the *redundant* method M_{RD}. The three situations that agent B can encounter when adding M_{RD} to its schedule are shown in the figure, and we explain the decision that B makes in each situation:

1. *Conflict with Existing Scheduled Methods*: This case is shown on the top right part of the figure. Adding M_{RD} is inconsistent with agent B's currently scheduled methods. B cannot schedule M_{RD} without removing one or more activities in its schedule to make space for it.

   Action: B does not schedule M_{RD}.

2. *Detrimental Effect on Horizon Methods*: This case is shown in the bottom left part of the figure. Adding M_{RD} to the schedule increases the failure potential of the *horizon* method, Q, so that it is no longer "safe".

   Action: B does not schedule M_{RD}.

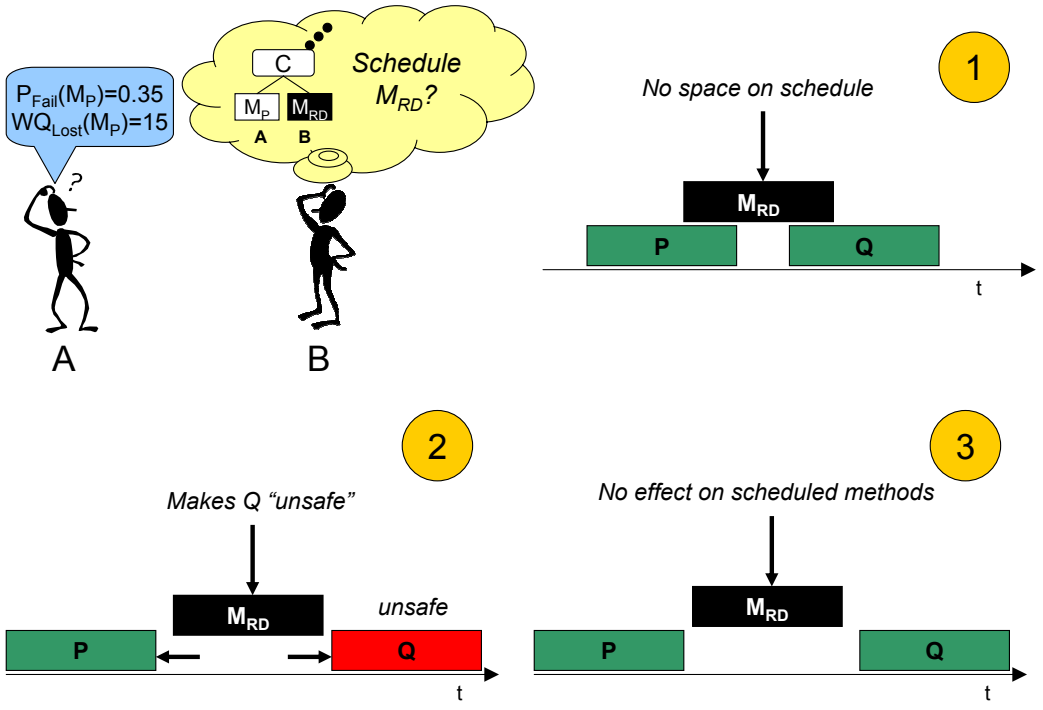

Figure 6.16: Agent "B" Considers the Passive Request from Agent "A"

3. *Harmless Effect on Horizon Methods*: This case is shown in the bottom right part of the figure. Adding M_{RD} does not cause any of the *horizon* methods to cease being "safe".

   Action: B adds M_{RD} to its schedule.

Aggressive Robust Scheduling

Given that *passive* requests for help do not compel the "owners" of *alternative* methods to schedule them, the request may fail if (1) the agents don't have space on their schedules to add the *redundant* method, or (2) the addition increases the expected quality loss of their *horizon* methods. Our *aggressive* strategy enables an agent with an *endangered* method to *coerce* the "owner" of an *alternative* method to schedule it.

To coordinate the scheduling of an *alternative* method, agents use a three-step negotiation process:

1. The *requesting* agent (i.e. the "owner" of the *endangered* method) asks the *recipient* agents (i.e. the "owners" of the *alternatives*) to make a "bid" to

schedule the *alternative* method.

2. When the *recipient* agents receive the *request* for a "bid", they attempt to add the *alternative* method to their schedules. If they succeed, they return to the *requesting* agent their "bid". The bid specifies how the addition of the *alternative* method affects their schedule.

3. After receiving all the "bids" (if any), the *requesting* agent selects one of this "bids", and issues a command to the *recipient* agent that made the *winning* "bid" *compelling* the agent commit to scheduling the *alternative* method.

The rest of the section will describe this mechanism in greater detail.

(1) Requesting Bids to Help an Endangered Method As the schedule executes, each agent monitors the potential failures of their *horizon* activities to determine whether an *aggressive* request for help should be initiated. An agent starts an *aggressive* request for help for a method, m, if the method meets three conditions,

1. m is the first *on-deck* method (i.e. it is the method scheduled to execute next), so taking immediate action is pressing.

2. m is *endangered*, or $P_{Fail}(m) > F_{Threshold}$.

3. *Passive* strategies have failed to offer help to m. This condition is implied by the first two. Since m is both the first *on-deck* method and *endangered*, *passive* requests for help would have been triggered for m (as long as $OD_{MaxActs} > 1$). However, we make this condition explicit for clarity.

If a method m meets the three conditions above, the *requesting* agent (the "owner" of m) initiates an *aggressive* request for help for the method by sending a call for "bids" to all the owners of *alternative* methods to m. These *recipient* agents are (1) the owners of the redundancy methods that can execute parallelly to m, and (2) the *requesting* agent itself if it owns any fall-back methods that can execute instead of m.

(2) Responding to Bid Requests When a *recipient* agent, ag, receives a "bid" request on behalf of an *endangered* method, m_P, it responds by formulating a "bid" in a three-step process:

1. First, it determines the *shortest* duration *alternative* method to m_P that it owns, m_A.

2. Second, it formulates a schedule that contains m_A, if feasible. The agent unschedules any *pending* methods necessary to make space for adding m_A to the schedule.

3. Finally, if the scheduling of m_A succeeded, the agent creates a "bid", b, that includes $Q_{Lost}(b)$, a measure of the quality the *recipient*'s schedule loses when adding m_A to the schedule.

The quality loss metric $Q_{Lost}(b)$, measures the quality that the *recipient* agent's schedule would lose if the *requesting* agent *coerces* the *recipient* agent to commit to bid b. We compute this quality by combining two quantities:

1. $T_{Lost}(b)$: This metric measures the quality loss to the methods on the agent's timeline. This loss is measured by computing the difference between the sum of the qualities of the *pending* methods *before* and *after* the *alternative* method, m_A, is added to the schedule.

   With $P_0(ag)$ the set of *pending* methods in the schedule of agent ag before m_A is added, and $P_1(ag)$ the set of *pending* methods in the schedule of agent ag after m_A is added, the quality loss incurred by adding m_A, $T_{Lost}(b)$, is given by

   $$T_{Lost}(b) = \sum_{i \in P_0(ag)} q_{all}(i) \; - \sum_{i \in P_1(ag)} q_{all}(i) \tag{6.8}$$

   As with the calculations of quality loss for the *hypothetical* inconsistencies in section 6.2.5, this measure provides an estimate of the quality loss based on the quality of the methods that get added/discarded during the bid. As explained, given the *limited* view of the agents, it is not generally possible to compute the exact quality loss, making an estimate necessary.

2. $H_{Lost}(b)$: This metric calculates the expected quality loss of the *horizon* methods in the agent's schedule. With $H_0(ag)$ the set of *horizon* methods in the schedule of agent ag before m_A is added, and $P_1(ag)$ the set of *horizon* methods *after* m_A is added, the expected quality loss of the *horizon* methods is

   $$H_{Lost}(b) = \sum_{i \in H_0(ag)} WQ_{Lost}(i) \; - \sum_{i \in H_1(ag)} WQ_{Lost}(i) \tag{6.9}$$

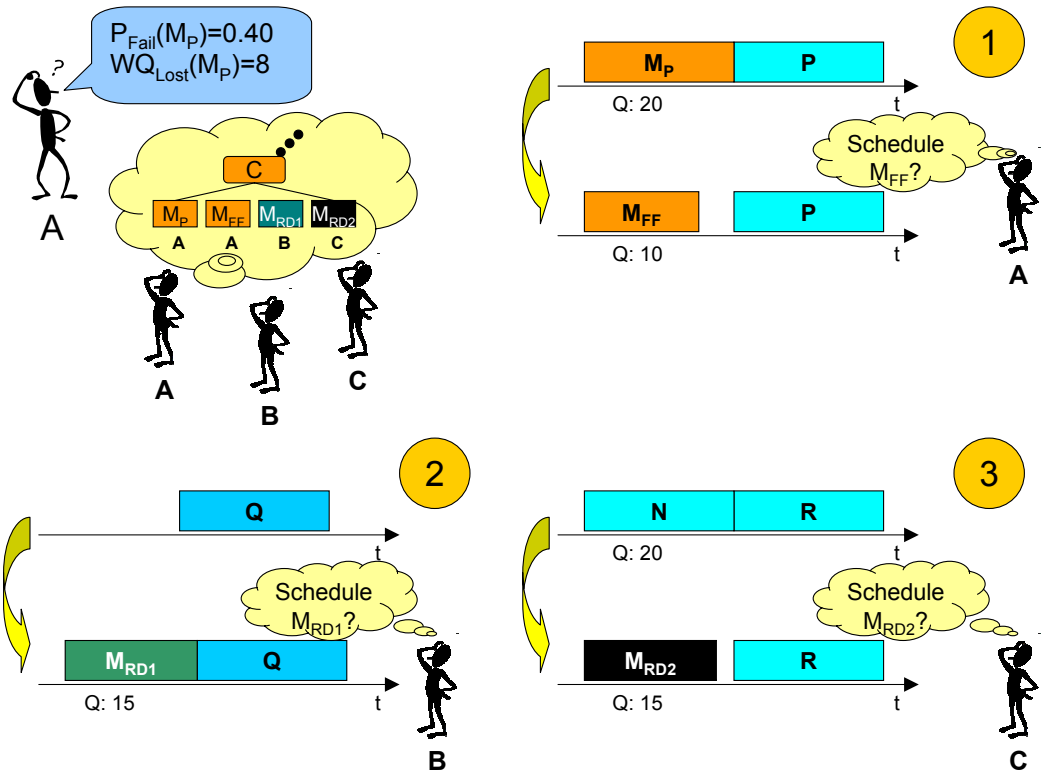

Figure 6.17: Agents Consider the Aggressive Request from Agent "A"

The metric $Q_{Lost}(b)$ is defined as a linear combination of these two quantities

$$Q_{Lost}(b) = Bid_T \times T_{Lost}(b) \; + \; Bid_H \times H_{Lost}(b) \tag{6.10}$$

where Bid_T and Bid_H are constants.

Figure 6.17 shows an example of a situation where the *requesting* agent A calls for bids that schedule *alternative* methods to its *endangered* method, M_P. The three *recipient* agents are:

1. A itself, since it "owns" a fall-back method M_{FF}.
2. B, the "owner" of the *redundant* method M_{RD1}.
3. C, the "owner" of the *redundant* method M_{RD2}.

All three agents generate bids that include their respective *alternative* methods. To simplify the example, we will assume that $H_{Lost}(b) = 0$ for all three bids (so $Q_{Lost}(b) =$

$T_{Lost}(b)$), and $q_{all}(m) = q(m)$ for all methods involved:

- Agent A makes a bid, b_A, that includes M_{FF} by forfeiting the execution of the *endangered* method M_P. The quality loss is the difference in quality between the primary method M_P, and its fall-back method M_{FF}, or

$$Q_{Lost}(b_A) = q(M_P) - q(M_{FF}) = 20 - 10 = 10$$

- Agent B makes a bid, b_B, that includes M_{RD1}. Agent B does not need to forfeit any of its currently scheduled *pending* methods to add M_{RD1} to its schedule [9]. Consequently,

$$\begin{aligned} Q_{Lost}(b_B) = & \sum_{i \in P_0(B)} q(i) \; - \sum_{i \in P_1(B)} q(i) = \\ & \sum_{i \in P_0(B)} q(i) \; - (\sum_{i \in P_0(B)} q(i) \; + \; q(M_{RD1}) \;) = \\ & - q(M_{RD1}) = -15 \end{aligned}$$

- Agent C makes a bid, b_C, that includes M_{RD2}. To add M_{RD2} to its schedule, agent C had to unschedule method N. Therefore,

$$\begin{aligned} Q_{Lost}(b_C) = & \sum_{i \in P_0(C)} q(i) \; - \sum_{i \in P_1(C)} q(i) = \\ & \sum_{i \in P_0(C)} q(i) \; - (\sum_{i \in P_0(C)} q(i) \; + \; q(M_{RD2}) \; - \; q(N) \;) = \\ & q(N) - q(M_{RD2}) = 20 - 15 = 5 \end{aligned}$$

(3) Selecting a Winning Bid Once all the *recipient* agents have submitted their "bids" to the *requesting* agent, the *requesting* agent needs to identify the *winning* "bid", b_{win}, to help its *endangered* method, m_P. The *winner* agent (the *recipient* agent submitting the *winning* "bid") will be *commanded* to commit to the bid schedule if the expected quality loss of the *endangered* method, m_P, is greater than the quality

[9] Given that no methods had to be unscheduled, and we assume that $H_{Lost}(b_B) = 0$, the *passive* strategy would have scheduled M_{RD1} in this case. We use this simplified example for illustrative purposes.

loss of the bid

$$WQ_{Lost}(m_p) > Q_{Lost}(b_{win}) \tag{6.11}$$

A *winning* "bid" is selected based on the information furbished by the *recipient* agents, so that the quality lost is minimized. Therefore the *winning* "bid", b_{win}, from a set of bids, $Bids$, is simply

$$b_{win} = b \; s.t. \; \min_{b \in Bids} Q_{Lost}(b) \tag{6.12}$$

Using the example in Figure 6.17, the *requesting* agent A, evaluates the three bids from the *recipient* agent A, B, and C (bid_A, bid_B and bid_C respectively). Out of this three, b_B has the minimum quality loss, so $b_{win} = b_B$. Since

$$WQ_{Lost}(m_p) > Q_{Lost}(b_B)$$

agent A sends a command to agent B to schedule the "redundant" method M_{RD1}.

Aborting Alternative Activities

So far, we have discussed how to *add* alternative methods to strengthen a multi-agent schedule. However, we recognize that the execution of these additional activities involves a trade-off: If several agents are concurrently executing "redundant" methods to decrease the chances of failure, they (1) may be giving up executing other activities in the schedule, or (2) minimally, the *flexibility* of their remaining *pending* methods is reduced (since the additional "redundant" activity pushes the execution of other *pending* methods further in the future). To minimize the effects of this trade-off, we have provided the agents with the ability to *abort* methods when it no longer makes sense to continue executing. We have defined four conditions under which methods are *aborted*:

1. When a higher-quality "redundant" method has finished successfully.

2. When a lower-quality "redundant" method has finished successfully, and continued execution of an *endangered* alternative can have detrimental consequences on *pending* activities.

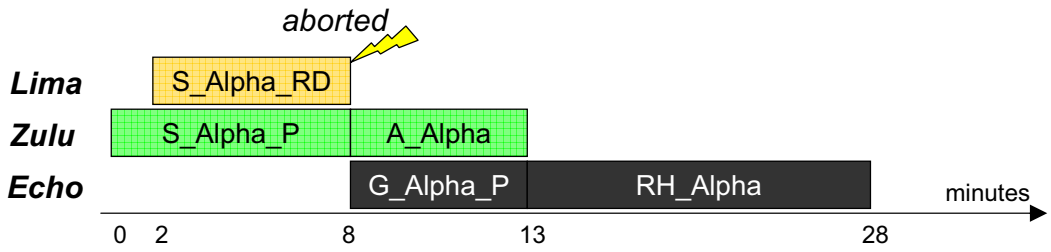

Figure 6.18: Abort Condition 1: Higher-Quality Sibling Finishes

3. When a method exceeds its LFT, and remaining *pending* activities should not be jeopardized.

4. When a method exceeds its deadline.

(1) Aborting a Method when a Higher-Quality Sibling Finishes The first *abort* condition occurs when two or more "redundant" methods are executing, and one of them, m_1, finishes with quality, $q(m_1)$. Continued execution of a "redundant" method, m_2, where $q(m_2) \leq q(m_1)$ makes little sense. When the owner of m_2 is notified that m_1 has finished successfully, it *aborts* m_2, freeing its timeline to execute some other activity.

An example of this situation can be illustrated by extending the motivating scenario in Figure 6.2: Lima has encountered stiff resistance, and one of its helicopters is shot down 2 minutes into the mission. Given the new situation, Lima estimates that its softening activity, *S_Alpha_P*, will last 10 minutes instead of the 5 minutes initially predicted. Lima sends a *passive* request for help to Tango, and Tango responds by adding *S_Alpha_RD* to its schedule, starting mortar bombardment. However, the battle progresses well, and 8 minutes into the mission, the terrorists have been subdued. Lima informs Tango that it has finished *S_Alpha_P* successfully. Upon receipt of this message, Tango ceases its mortar bombardment and aborts *S_Alpha_RD*. The team schedule at minute 8, after *S_Alpha_P* finishes execution, is shown in Figure 6.18.

(2) Aborting an Endangered Method when a Lower-Quality Sibling Finishes The second *abort* condition occurs when two or more "redundant" methods are executing, and one of them, m_1, finishes with quality, $q(m_1)$. However, an unfinished sibling, m_2, where $q(m_2) > q(m_1)$, is still executing, and can potentially accrue

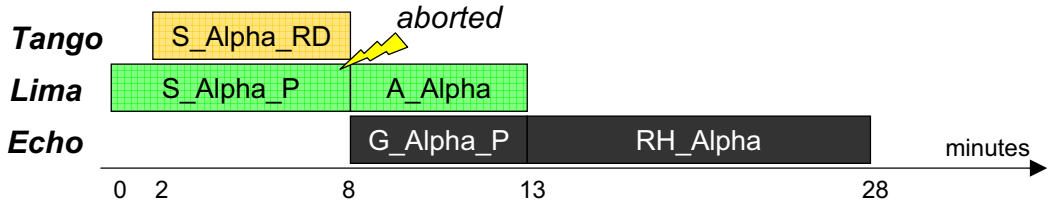

Figure 6.19: Abort Condition 2: Lower-Quality "Safer" Sibling Finishes

greater quality for the schedule. To decide whether to continue executing m_2, we use its *hypothetical* analysis of failure (remember that m_2 is *executing* and is therefore a *horizon* activity). Method m_2 is *aborted* when two conditions are met:

1. The probability of failure is greater than the *failure threshold*, or $P_{Fail}(m_2) > F_{Threshold}$.

2. The potential expected quality loss, if m_2 fails, is greater than the quality difference between m_1 and m_2 (i.e. the quality gain if m_2 finishes):

$$WQ_{Lost}(m_2) > (\, q(m_2) - q(m_1) \,)$$

An example of this situation can be demonstrated by expanding on our motivating scenario: Lima has encountered stiff resistance and one of its helicopters is shot down 2 minutes into the mission. Lima estimates that its softening activity, *S_Alpha_P*, will last 10 minutes instead of the 5 minutes initially predicted, and sends a *passive* request for help to Tango. Tango adds *S_Alpha_RD* to its schedule, and starts mortar bombardment. At minute 8, Tango communicates to Lima that it has accomplished its goals. The accrued quality of *S_Alpha_RD* is $q(S\_Alpha\_RD) = 3$. While Lima could continue bombing Alpha and subdue the enemy threat further, it has to consider that Echo cannot proceed to enter the premises at Alpha until Lima finishes its bombardment. With $F_{Threshold} = 0.10$, the likelihood that Lima won't be able to accomplish its bombardment goals in time, $P_{Fail}(S\_Alpha\_P) = 0.25$ is high. If Lima delays the mission, it can cause Echo's hostage rescue activities to fail. This potential failure is reflected in the expected quality loss

$$WQ_{Lost}(S\_Alpha\_P) = 0.25 \times q(RH\_Alpha) = 0.25 \times 30 = 7.5$$

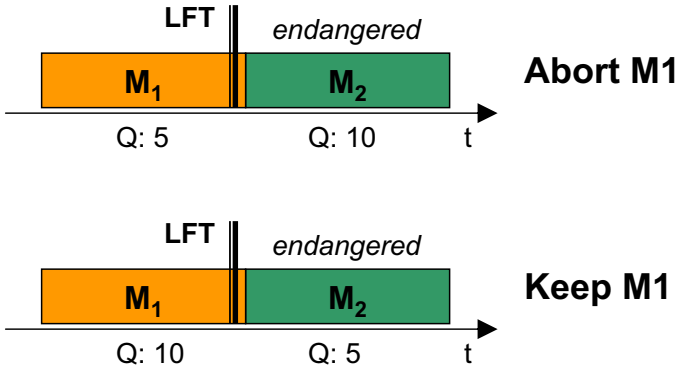

Figure 6.20: Abort Condition 3: Method has Exceeded its LFT

which is higher than the quality gain that would be achieved by *S_Alpha_P* in case of successful execution,

$$q(S\_Alpha\_P) - q(S\_Alpha\_RD) = 5 - 3 = 2$$

Consequently, Lima *aborts S_Alpha_P*. The team schedule at minute 8, after *S_Alpha_RD* finishes execution, is shown in Figure 6.19.

(3) Aborting a Method when it Exceeds its LFT The third *abort* condition enables agents to reason about whether continued execution of a method, m, that has exceeded its prescribed LFT should proceed [10]. We make the decision on whether to proceed the execution of m by comparing the *losses* that the schedule would incur if:

1. The method m is aborted, which leads to the loss of m. The loss to the schedule is given by $q_{all}(m)$.

2. The method m proceeds, and causes successor activities to *fail*. The loss to the schedule is given by $WQ_{Lost}(m)$. Since m is in execution, and in fact, it has already exceeded its *expected* duration, the duration distribution, $d_{PDF}(m)$ is adjusted to discard the already *exceeded* durations.

An example of both situations is shown in Figure 6.20. The top of the figure shows a method M_1 that has exceeded its LFT, and *endangers* the next method on

[10] The third *abort* condition only considers the situation where the LFT is earlier than the deadline. The case when they are the same is considered in the fourth *abort* condition.

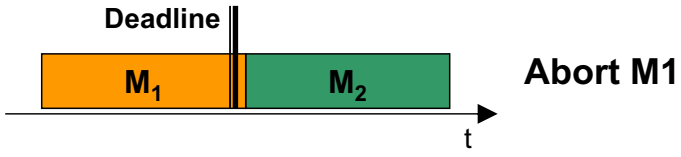

Figure 6.21: Abort Condition 4: Method has Exceeded its Deadline

the schedule, M_2. The original duration distribution of M_1 is shown as

$$d_{PDF}(M_1) = ((8\ 0.25)(10\ 0.50)(12\ 0.25))$$

However, given that M_1 has already exceeded 10 minutes of execution, the *adjusted* duration distribution is

$$d'_{PDF} = ((12\ 1.0))$$

With $q(M_1) = 5$ and $q(M_2) = 10$, we calculate the loss if M_1 continues executing as

$$WQ_{Lost}(M_1) = P\{d(M_1) = 12\} \times q(M_2) = 1.0 \times 10 = 10$$

and the loss if M_1 is aborted as

$$q_{all}(M_1) = q(M_1) = 5$$

Since $WQ_{Lost}(M_1) > q_{all}(M_1)$, the method M_1 should be aborted.

The bottom of the figure shows the same methods, M_1 and M_2, but this time the quality figures are reversed: $q(M_1) = 10$ and $q(M_2) = 5$. Consequently, $WQ_{Lost}(M_1) = 5$ and $q_{all}(M_1) = 10$, and $WQ_{Lost}(M_1) < q_{all}(M_1)$, meaning that continued execution of M_1 is more valuable than the loss of M_2.

(4) Aborting a Method when it Exceeds its Deadline This fourth *abort* condition is a function of the *hard* deadlines in our problem domain. A method that does not meet its deadline has *failed*, and it cannot accrue quality. Therefore, it makes little sense to continue executing it.

An example of this situation is shown in Figure 6.21. Here method M_1 has *failed* its deadline, and its continued execution is not only useless, but it *endangers* the next

method on the schedule, M_2. Consequently, M_1 is *aborted*.

6.3 Experimental Design and Results

We tested the validity of our robust scheduling techniques by performing experiments that compared the performance of two agent teams: (1) A team of "reactive" agents that rely solely on the conflict resolution techniques described in Chapter 5 to recover from failures once they occur, and (2) a team of "pro-active" agents that besides recovering from failure, attempt to prevent them by using the JIT Backfilling strategy we developed in this chapter to increase the fault-tolerance of the schedule.

We used two problem sets in our experimental comparisons. The first problem set consists of nine subsets of randomly generated C_TAEMS scenarios. The scenarios were constructed to highlight the lost opportunities to strengthen the schedule incurred by purely "reactive" agents. The second problem set consisted of scenarios provided by the Coordinators program for the Year 2 evaluation.

6.3.1 JIT Backfilling Parameters

While we experimented with different settings of the parameters used by the JIT-BF algorithm, to maintain consistency in the experimental results, we selected a set of parameters that were fixed for all the experimental runs reported in the following sections. The chosen parameter set was:

- $R_{PredLev} = 1$: This parameter restricts the *related set* of a *horizon* method to itself and its immediate predecessors.
- $OD_{MaxActs} = 3$: This parameter fixes an upper limit of 3 methods in the *on-deck* set.
- $OD_{MaxTime} = 30$: This parameter fixes the time horizon of the *on-deck* set to 30 ticks [11].
- $F_{Threshold} = 0.1$: This parameter indicates that the JIT Backfilling strategies kicked in when the probability of failure of a *horizon* method was computed to

[11] This value is meaningless without a reference to the duration of the methods in the scenarios. The duration of methods in our problems ranged from 4 to 13 ticks.

be higher than 10%.

- $Bid_T = 1$: This parameter indicates that the quality lost by the timeline methods of an agent (as it adds a *redundant* method to its schedule to fulfill a bid request) was counted in full.

- $Bid_H = 0$: This parameter indicates that the expected quality loss of the *horizon* methods of an agent (as it adds a *redundant* method to its schedule to fulfill a bid request) was not counted. The reason is that in the problems we experimented with, we discovered that $T_{Lost}(b)$ dominates $H_{Lost}(b)$, so that computing the expected quality loss does not make a difference to the decisions taken by the agents in all but a few scenarios.

6.3.2 Generated Scenarios

The scenarios we generated to test the validity of our JIT-BF strategy emphasized two features: (1) Scheduling without thought of potential failures misses opportunities from "strengthening" the schedules against these failures, and (2) Agents cannot easily recover from a failure once it has occurred, and the schedule performance suffers.

An partial view of an example 10-agent C_TAEMS scenario generated for these experiments is shown in Figure 6.22. As in Chapter 5, the activity hierarchy is arranged left-to-right, with scheduled activities shown in blue, and unscheduled activities in red. Higher-level tasks have a rounded-edge rectangular shape, methods a straight-edge rectangular shape.

We generated 10-agent, 20-agent, and 30-agent scenarios (scenario generation is described in Section 4.2.1). The 10-agent scenarios have 2 "problem" tasks. The 20-agent scenarios have 4 "problem" tasks. The 30-agent scenarios have 6 "problem" tasks. Problem tasks have a SUMAND QAF, and have eight window children each. These eight "windows" are divided into pairs. The *paired* "windows" are linked by an enables NLE, and are *temporally overlapped* (i.e. they share the same release times and deadlines). Windows have four cnode children with a MAX QAF. The four cnodes under a "window" are divided into pairs. The *paired* cnodes are linked by an enables NLE. Each cnode expand into three children methods that represent *alternative* ways of accomplishing the cnode objective: The "primary" method, a "fall-back", and a "redundancy". The primary methods were given an expected quality of 15, and an expected duration of 10 time units. The three points in a method's quality and

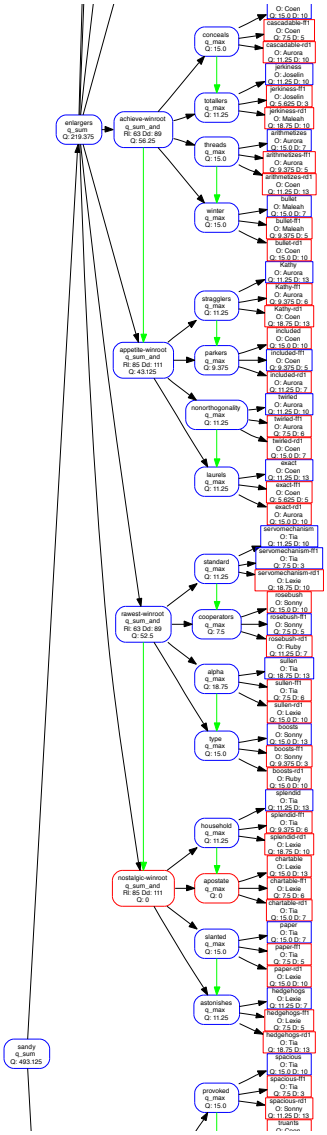

Figure 6.22: Example C_TAEMS Scenario for Testing JIT Schedule Robustness Strategies

duration distributions consisted of an expected value, a second value 30% lower, and a third value 30% higher.

For each scenario, we compared the performance of two strategies for dealing with durational uncertainty:

1. Reactive (NC): This strategy uses a purely "reactive" approach to dealing with durational uncertainty. A centralized solver generates an initial schedule that

assumes methods will execute their expected duration. When inconsistencies arise, the agents take conflict resolution actions to repair the schedule (as described in Chapter 5).

2. Pro-active (WC): In addition to reacting to inconsistencies, this strategy uses a "pro-active" approach that uses the durational uncertainty model to locate unexpected activity durations, and leverages the STN's conflict analysis tools to discover potential failure points (described in Section 6.2). A centralized solver generates an initial schedule that assumes methods will execute their expected duration. As the schedule executes, the agents locate the potential failure points using the JIT-BF strategy, and attempt to strengthen the schedule using *alternative* activities.

We created a total of 9 sets of 150 scenarios (each containing 50 10-agent scenarios, 50 20-agent scenarios, and 50 30-agent scenarios). The nine sets differed in two key parameters: *deadline tightness* and *uncertainty* parameters. Deadline tightness defines the ratio between the size of the time window of a "window" task, and its duration. Uncertainty indicates the probability that the maximum duration of a method will occur. Both parameters are described in greater detail in Section 4.2.1. We hypothesized that there would be an increase in the advantage of agents using "pro-active" strategies to strengthen the schedule against failures as (1) the deadlines became tighter and (2) the uncertainty increased. The intuitive reasoning is that tighter deadlines and increased uncertainty would make failures more likely, increasing the value of "pro-active" steps.

Table 6.1 displays the results of applying the two strategies, NC (reactive) and WC (pro-active), on the generated scenarios. The results validate our initial hypotheses. We run both strategies on each of the 9 sets of scenarios, and divided the results for each set into three parts, based on the number of agents in the problem. For each scenario, we computed the ratio Q_s^r using Equation 4.6, where Q_s^r is the quality ratio achieved by strategy s (either NC or WC). The results in the table show *deadline tightness* varying in the horizontal direction, and *uncertainty* changing in the vertical dimension. For each one of the nine different settings of *deadline tightness* and *uncertainty*, the average ratios Q_s^r were computed on the 10, 20, and 30 agent scenarios. These values are shown side-by-side, with the p-value (obtained using the Student's t-test) comparing the results for both strategies below them. Every set of scenarios showed a marked advantage for the "pro-active" agents using the JIT robust

			tighter $\Longleftrightarrow$ *looser* **Deadline Tightness**					
		Ag	1.0		1.2		1.4	
			NC	WC	NC	WC	NC	WC
Uncertainty	0.125	10	0.7545	0.9148	0.8345	0.9523	0.8428	0.9707
			$p = 0$		$p = 0$		$p = 0$	
		20	0.7484	0.9898	0.8937	0.9659	0.9353	0.9685
			$p = 0$		$p = 0$		$p = 0.0132$	
		30	0.7796	0.9894	0.9356	0.9790	0.9364	0.9792
			$p = 0$		$p = 0$		$p = 0$	
	0.25	10	0.6699	0.9091	0.8143	0.9427	0.8653	0.9472
			$p = 0$		$p = 0$		$p = 0.0017$	
		20	0.6749	0.9942	0.8733	0.9730	0.9275	0.9633
			$p = 0$		$p = 0$		$p = 0.0153$	
		30	0.6766	0.9914	0.8677	0.9850	0.9398	0.9660
			$p = 0$		$p = 0$		$p = 0.0762$	
	0.333	10	0.7041	0.9344	0.7680	0.9119	0.8426	0.9383
			$p = 0$		$p = 0$		$p = 0$	
		20	0.6948	0.9776	0.8638	0.9660	0.9055	0.9708
			$p = 0$		$p = 0$		$p = 0$	
		30	0.6866	0.9790	0.8751	0.9598	0.9339	0.9611
			$p = 0$		$p = 0$		$p = 0.0722$	

Table 6.1: Quality Ratios for "Reactive" vs. "Pro-active" Schedule Robustness Approaches

scheduling techniques developed in this chapter. The *deadline tightness* proved to be the most important factor. Scenarios with the tightest deadlines (1.0) showed the largest performance differences between "pro-active" and "reactive" agents. All sets of scenarios with deadline tightness of 1.0 or 1.2 showed significant gains ($p < 0.05$) for "pro-active" agents using strategy WC. The performance differential, however, decreased as the deadlines became looser (as expected), with both strategies showing comparable performance when the deadlines were the loosest (1.4). Uncertainty also played a factor in the results, although to a lesser degree than deadline tightness. We observe that when uncertainty was highest (0.333), the agents using strategy WC outperformed the agents using strategy NC by the largest margin. The size of the scenario did not prove consequential, as all scenarios in the same set (i.e those with the same deadline tightness and uncertainty values), showed similar results regardless of the number of agents used.

			tighter $\Longleftrightarrow$ *looser* **Deadline Tightness**					
		Ag	1.0		1.2		1.4	
			WC_P	WC_A	WC_P	WC_A	WC_P	WC_A
Uncertainty	0.125	10	0.0643	0.3292	0.0615	0.2127	0.0721	0.1516
			$\sigma = 0.3292$	$\sigma = 0.0577$	$\sigma = 0.0315$	$\sigma = 0.0653$	$\sigma = 0.0340$	$\sigma = 0.0593$
		20	0.0830	0.3505	0.0933	0.2055	0.1005	0.1222
			$\sigma = 0.0271$	$\sigma = 0.0271$	$\sigma = 0.0239$	$\sigma = 0.0363$	$\sigma = 0.0248$	$\sigma = 0.0319$
		30	0.0896	0.3486	0.1093	0.1974	0.0998	0.1074
			$\sigma = 0.0195$	$\sigma = 0.0298$	$\sigma = 0.0207$	$\sigma = 0.0310$	$\sigma = 0.0224$	$\sigma = 0.0268$
	0.25	10	0.0672	0.3336	0.0673	0.1929	0.0776	0.1359
			$\sigma = 0.0405$	$\sigma = 0.0672$	$\sigma = 0.0318$	$\sigma = 0.0544$	$\sigma = 0.0297$	$\sigma = 0.0545$
		20	0.0807	0.3398	0.0970	0.1865	0.1038	0.1120
			$\sigma = 0.0294$	$\sigma = 0.0411$	$\sigma = 0.0269$	$\sigma = 0.0379$	$\sigma = 0.0212$	$\sigma = 0.0369$
		30	0.0901	0.3398	0.1157	0.1885	0.1139	0.1095
			$\sigma = 0.0207$	$\sigma = 0.0322$	$\sigma = 0.0214$	$\sigma = 0.0368$	$\sigma = 0.0229$	$\sigma = 0.0282$
	0.333	10	0.0714	0.3275	0.0806	0.2077	0.0814	0.1699
			$\sigma = 0.0461$	$\sigma = 0.0650$	$\sigma = 0.0462$	$\sigma = 0.0525$	$\sigma = 0.0395$	$\sigma = 0.0502$
		20	0.0766	0.3484	0.1017	0.2190	0.1145	0.1416
			$\sigma = 0.0328$	$\sigma = 0.0424$	$\sigma = 0.0282$	$\sigma = 0.0392$	$\sigma = 0.0261$	$\sigma = 0.0360$
		30	0.0934	0.3565	0.1124	0.2187	0.1273	0.1338
			$\sigma = 0.0194$	$\sigma = 0.0269$	$\sigma = 0.0227$	$\sigma = 0.0299$	$\sigma = 0.0220$	$\sigma = 0.0296$

Table 6.2: Ratio of Methods Executed by JIT Robust Scheduling Strategies

To gain some insight into how the *passive* and *aggressive* strategies to schedule alternative methods affect the methods executed by the agents, we calculated the proportion of methods added by these techniques with respect to all the methods in the schedule. Table 6.2 shows the ratio of executed methods that were added to the schedule by both *passive* (WC_P) and *aggressive* (WC_A) JIT-BF strategies. The computation of these ratios is straightforward. We first divide the methods executed by the agents when running a scenario into three sets:

1. *Scheduler Methods* (S): These methods were added to the schedule either by the initial (centralized) schedule, or by the agents' schedulers.

2. *Passive Alternative Methods* (P): These methods were added to the schedule by the *passive* JIT-BF techniques developed in Section 6.2.6.

3. *Aggressive Alternative Methods* (A): These methods were added to the schedule

by the *aggressive* JIT-BF techniques developed in Section 6.2.6.

The total number of methods in each set can be easily found using the counting function:

$$Num_S = \sum_{i \in S} 1 \tag{6.13}$$

$$Num_P = \sum_{i \in P} 1 \tag{6.14}$$

$$Num_A = \sum_{i \in A} 1 \tag{6.15}$$

where Num_S is the number of scheduler methods, Num_P the number of methods added *passively* to the schedule, and Num_A the number of methods added *aggressively* to the schedule. The total number of executed methods (Num_{all}) is simply the sum of the number of methods in each set:

$$Num_{all} = Num_S + Num_P + Num_A \tag{6.16}$$

Once the total number of methods executed in a problem have been found, we can easily compute the ratio of executed methods added by *passive* (WC_p) and *aggressive* (WC_A) by:

$$WC_P = Num_P / Num_{all} \tag{6.17}$$

$$WC_A = Num_A / Num_{all} \tag{6.18}$$

The average ratios WC_P and WC_A were calculated for each of the 9 settings of *deadline tightness* and *uncertainty*. The ratios for 10, 20, and 30 agent scenarios are reported separately. The average ratios are reported side-by-side, with the standard deviations (σ) reported immediately below.

The results show a predictable trend. When the deadlines are tightest (1.0), and the uncertainty highest (0.333), the number of methods added by the JIT-BF strategies WC_P and WC_A is the largest. A seemingly surprising trend is that WC_P remains fairly constant under all settings. On the other hand, WC_A is both larger than WC_P in all but a few sets, and it varies significantly across the different settings. The explanation for the constancy of WC_P is that due to the design of these scenarios, there is not much room for agents with already scheduled methods to place additional

Ag	NC	WC
25	0.8997	0.9817
	$p = 0.0079$	
50	0.8873	0.9504
	$p = 0.0635$	

Table 6.3: Quality Ratios for "Reactive" vs. "Pro-active" Schedule Robustness Approaches (Coordinators Problems)

methods on their schedules *passively*. A *passive* request in this scenarios typically succeeded when the *recipient* agent was idle during the required time window. A larger burden was then placed on the shoulders of the *aggressive* techniques. It is then the WC_A ratios that show the expected variance as the scenario settings change.

6.3.3 Coordinators Scenarios

To further validate the advantages of our JIT-BF strategies with a more challenging set of scenarios, we tested both teams of "pro-active" and "reactive" agents on a set of C_TAEMS scenarios used in the Year 2 evaluation of the Coordinators program. These scenarios were described in Section 4.2.2. To conform to the scenario characteristics of our domain of interest (see Section 6.1), all soft NLEs (*facilitates* and *hinders*) were removed from the scenarios. A new initial schedule (that took into consideration this removal) was formulated for each scenario.

Table 6.3 shows the quality results obtained by both teams of agents on this set of problems. "Pro-active" agents using strategy WC showed a marked improvement in performance over the "reactive" agents team. The 32 25-agent set of scenarios showed a significant performance differential between the two teams ($p < 0.05$). The 24 50-agent set of scenarios also appears to show a performance gain by the use of strategy WC, but the difference was not significant enough to draw certain conclusions ($p > 0.05$).

Table 6.4 shows the ratios of methods executed by *passive* (WC_P) and *aggressive* (WC_A) strategies. These results show a markedly different trend from the scenarios in Table 6.2. Here, the number of *passively* scheduled methods is much larger than those *aggressively* scheduled, demonstrating the importance of both types of strategies as the types of scenarios change. The reasons for the larger success of *passive*

Ag	WC_P	WC_A
25	0.0810	0.0093
	$\sigma = 0.0304$	$\sigma = 0.0061$
50	0.0632	0.0078
	$\sigma = 0.0236$	$\sigma = 0.0056$

Table 6.4: Ratio of Methods Executed by JIT Robust Scheduling Strategies (Coordinators Problems)

strategies in these scenarios are two: (1) The release and deadlines of "window" tasks spanned a larger time interval that contained a larger number of activities. There was consequently more slack for methods to slide up and down the timeline when *passive* requests for help arrived. (2) There were more fall-back and redundancy methods under each cnode task. While the generated scenarios used in the previous section had 1 fall-back and 1 redundancy method under each cnode, the Coordinators scenarios have 2 fall-backs and 2 redundancy methods. The larger number of *alternative* methods improves the ability of agents to provide aid to each other. Since *passive* requests for help were much more successful, the number of *aggressive* requests was significantly reduced, accounting for the small number of *aggressively* added methods.

6.4 Summary

In this chapter, we presented a robust scheduling framework, called *Just-in-Time* Backfilling (JIT-BF), that uses the STN's conflict analysis tools to locate potential activity failures due to unexpected activity durations. "Unexpected" activity durations are spotted using a durational model of uncertainty. The agents discover potential inconsistencies by introducing these unexpected durations into the STN. The STN rejects inconsistent durations, producing a *negative cycle*, which denotes a potential failure that could occur if the inconsistent duration materializes during schedule execution. The negative cycle can be analyzed using conflict explanation techniques (see Section 2.5.2) to discover what activities are affected by the potential failure. The agents can then try to strengthen the schedule against the potential failure by adding *alternative* activities to the schedule. These *alternative* activities are scheduled in addition to, or instead of, an activity that may potentially fail.

Rather than examining the failure potential of all *pending* activities, the agents

focus on a set of *horizon* activities that are close-to-execution, and determine their potential for failure. We start by locating all the potential inconsistencies that can occur when executing each of the *horizon* activities. Then, we aggregate the analyses of all these inconsistencies into a combined analysis of failure for the activity. This analysis determines whether the *horizon* activity is *endangered* (i.e. has a high probability of failure). A *request* for help to other agents is sent on behalf of *endangered* activities. *Passive* requests for help on behalf of an *endangered* activity piggy-back on the agent's *Distributed State Management (DSM)* (see section 2.3). Agents receiving these requests attempt to schedule "redundancies" to the *endangered* activity. These redundancies are added only if they do not make make the *horizon* activities of their owner agents more likely to fail. *Aggressive* requests for help use a "bidding" mechanism where all the agents that "own" alternatives to the *endangered* activity place a bid to schedule their alternative. The "bidding" agents may need to modify their schedule, including unscheduling some of their previously scheduled activities to make space for the alternative. The *requesting* agent selects the bid with the least detrimental effects for the bidding agent.

To ensure that agents do not continue executing activities when they are no longer useful, we have defined a set of four companion abort conditions that determine when an agent should discontinue the execution of an activity. We abort activities when (1) a "redundant" activity has finished with higher quality, (2) a "redundant" activity has finished with lower quality, but continued execution can *endanger* other activities in the schedule, (3) an activity has exceeded its LFT, and continued execution jeopardizes other (more valuable) activities in the schedule, and (4) an activity has exceeded its deadline (i.e. it has *failed*).

Our results show a significant improvement in the performance of agents using the JIT-BF strategy. By "pro-actively" taking steps to prevent failures before they occur, the agents are able to minimize the effect on the schedule of durational uncertainty. Our results show that scenarios with tight deadlines (implying less flexibility in the schedule), and high uncertainty (implying higher likelihood of failure) benefit the most from taking these "pro-active" steps.

Chapter 7

Conflict-Driven Coordination

In this chapter, we shift our focus away from the management of durational uncertainty (the topic explored in Chapters 5 and 6) to investigate how the STN's conflict analysis tools can be leveraged to improve the quality of the multi-agent schedule. Specifically, we examine a form of *conflict-driven* coordination that enables an agent to update existing temporal inter-agent commitments: In an over-subscribed scheduling domain, where there are more activities than the agents can potentially execute, the scheduling of some "valuable" *desired* activities could be infeasible due to prior commitments between the agents in the team. However, a coordinated revision of these commitments might enable the successful installation the *desired* activity. We present strategies to perform such coordinated revisions. These strategies exploit an analysis of the conflicts that occur during the scheduling of the *desired* activity to discover the set of inter-agent commitments that need to be updated.

Previous research has not explored the use of scheduling conflicts to locate inter-agent coordination opportunities within the context of STN-based distributed scheduling. In this work, we show how the STN's temporal conflict analysis tools can provide a valuable aid that significantly simplifies the identification of conflicting temporal relations. This analysis can be used to make coordinated schedule changes: When an agent attempts to schedule an activity, and fails, the STN produces a *negative cycle* (or a set of them). This *negative cycle* contains the set of conflicting temporal constraints that prevent the scheduling of the *desired* activity. In effect, the *negative cycle* produced by the STN can be used to focus the required search on the relevant set of conflicting temporal relations that need to be updated. Using STN *conflict explanation* techniques similar to those used in previous chapters, the agents can mine

the *negative cycle*, extract the set of conflicting activities in the multi-agent schedule that block the *desired* activity, and locate which activities in other agents' schedules need to be moved earlier or later, and by how much. Using this information the agents can coordinate an update to their scheduled commitments so that the *desired* activity can be installed.

We have designed two strategies that demonstrate how *negative cycles* can be leveraged to coordinate changes to a multi-agent schedule. These two strategies present a trade-off between the size of the plan that an agent needs to be aware of to coordinate changes (i.e. the size of the agent's subjective view, see Section 2.2.2) [1], versus the amount of time needed for coordination. The first approach, the *recursive* strategy, requires smaller subjective views, but is more computationally intensive. This approach only assumes that agents are aware of their "local" activities, and all "remote" activities directly "linked" to them by a temporal relation. When an agent fails to schedule a *desired* activity because of a conflicting temporal commitment made with a "remote" agent (e.g. a precedence relation that links a "local" scheduled activity to a "remote" activity), the agent *requests* the "owner" of the "remote" activity to *shift* it *forward* or *backward* in time, so that the *desired* activity can be installed. If this "remote" *recipient* agent finds that it is unable to *shift* its activity as *requested* because of another conflicting temporal commitment to a third agent, it *recursively requests* this third agent to *shift* its conflicting activities. These *recursive* requests continue until a *recipient* agent is able to make a "local" decision (i.e. it can decide whether it can make the *shift* or not based only on its "local" schedule), and *respond*. The second approach, the *full-chain* strategy, requires larger subjective views, but is computationally faster. This strategy assumes that an agent that "owns" an activity in a *chain* (a set of activities linked by precedence relations) is aware of *all* the activities in the *chain*. With this larger view, an agent that fails to schedule a *desired* activity can locate *all* the activities (both local and remote) that need to be *shifted* to enable the *desired* activity to be installed. The agent can then initiate a *request* to all the involved agents to accomplish this *shift*. We have tested the performance of two teams of agents that use these two coordination strategies against the performance of a "baseline" team that does not modify commitments once they are made. Our results show the two coordinating agent teams significantly outperform the baseline agent team in all test conditions except those where large durational uncertainty makes it

[1] Larger agent views increase the amount of information that needs to be propagated by the agents' DSM (see Section 2.3). Consequently, larger agent views translate into greater communication costs.

difficult to successfully execute chains of activities. Further, our results validate the trade-off between the two coordination strategies: The *recursive* strategy reduced the amount of transmitted information by over 50% compared to the *full-chain* strategy, while the *full-chain* strategy reduced the computation time needed for coordination by over 25%.

7.1 Forming Inter-Agent Temporal Commitments

Cooperative agents executing a joint schedule need to coordinate their decisions to ensure good performance. Multi-agent plans with temporally interconnected activities require the agents to coordinate, making commitments to one another so that the temporal interdependencies are respected. In such scenarios, agents need to be provided with strategies that enable them to communicate with one another and form commitments. Even a simple plan that features only two activities assigned to two different agents and linked by a precedence constraint requires these agents to coordinate, forming a commitment that enforces the precedence relation.

Our cMatrix agent (see Section 2.3) is equipped with a *passive* coordination strategy and an *aggressive* coordination strategy we call "optimistic synchronization" (see Section 2.3). Both strategies are described in the next two sections.

7.1.1 Passive Coordination

The *passive* coordination approach uses an *implicit* coordination strategy that piggybacks on the cMatrix agent's Distributed State Mechanism (DSM). The strategy relies on the transmission of a metric we have labeled *potential* quality. This metric provides the value that an activity could achieve, if scheduled. An agent that wants to schedule a valuable activity, m, but is unable to because its required predecessors have not been scheduled, propagates the *potential* quality of m to the agents that "own" the predecessor activities to m. Upon receipt of this information, an agent that "owns" a predecessor activity will attempt to schedule it, if it does not conflict with its existing commitments.

Figure 7.1 shows an example of a pair of agents coordinating to schedule two activities linked by a precedence constraint using this strategy. Agent B wants to install activity M_2 in its schedule, but M_2 must be preceded by activity M_1 (owned

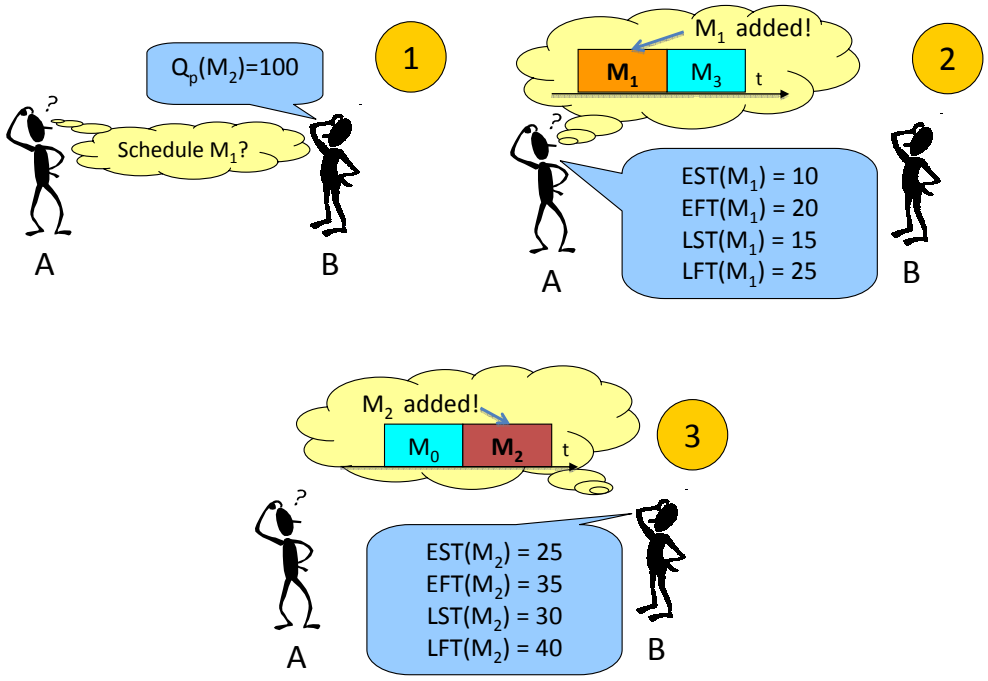

Figure 7.1: Passive Strategy to form Inter-Agent Commitments

by agent A), which is not yet scheduled. Both activities are assumed to have a duration of 10, and the expected quality of M_2, $q(M_2)$, is 100. The coordination process involves three steps:

1. Initially, agent B transmits to agent A the *potential* quality of activity M_2. The figure shows this request in the top left corner. The *potential* quality of activity M_2, $Q_P(M_2)$, is set to 100.

2. After receiving this information, agent A attempts to install activity M_1 in its schedule [2]. M_1 is installed as early as feasible to give the greatest amount of flexibility possible to M_2. If M_1 is successfully installed, agent A transmits the temporal bounds of M_1 to agent B. This step is shown in the top right corner of the figure. The four temporal bounds of M_1 (see Section 2.1 for a description

[2]The installation attempt occurs the next time agent A's scheduler is invoked. After agent A has installed all locally valuable activities (its "contributors", see Section 2.3 for further details) it will attempt to insert M_1 if enough timeline space is left.

of activity temporal bounds) are

$$EST(M_1) = 10$$
$$EFT(M_1) = 20$$
$$LST(M_1) = 15$$
$$LFT(M_1) = 25$$

Agent B updates the information of activity M_1, asserting the provided temporal bounds into its STN.

3. Finally, agent B tries again to install M_2 in its schedule. The bottom part of the figure shows Agent B successfully inserting M_2 into its schedule. The temporal bounds on M_2 are given as

$$EST(M_2) = 25$$
$$EFT(M_2) = 35$$
$$LST(M_2) = 30$$
$$LFT(M_2) = 40$$

Note that the *earliest finish time* of M_1 is before the *latest start time* of M_2, meaning that the precedence relation between the two activities can be respected. Agent B then proceeds to inform agent A about the new temporal bounds of M_2. Agent A then asserts these temporal bounds into its STN, and both agents assert the precedence constraint linking M_1 to M_2 into their STNs. The assertion of this constraint marks a new commitment made by the two agents where they agree that neither will make scheduling decisions that affect the enforcement of this link.

7.1.2 Optimistic Synchronization

The second coordination approach uses an *explicit* coordination strategy. This strategy was briefly described in Section 2.3.2, and can be used to *negotiate* the scheduling of new activities in the schedule. An agent that wants to schedule a valuable activity, m, can initiate a coordination session with other agents that "own" unscheduled predecessors of m. The *requesting* agent asks the "owners" of the unscheduled pre-

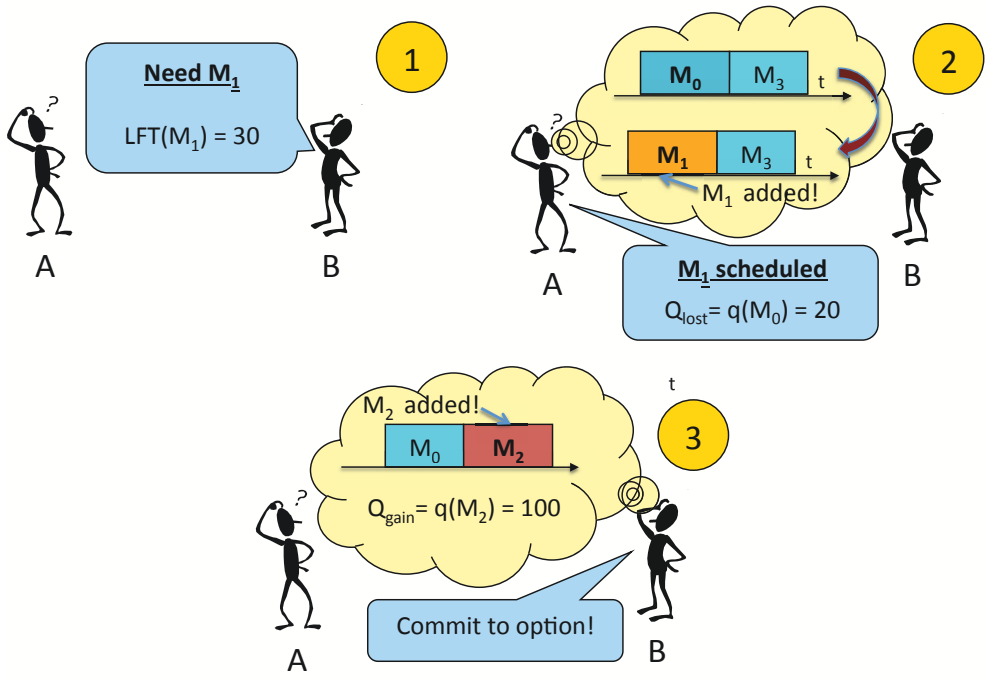

Figure 7.2: Aggressive Strategy to form Inter-Agent Commitments

decessors to schedule them while respecting certain temporal bounds. The *recipient* agents generate schedules that contain the requested predecessor activities, and evaluate how the scheduling of these activities affects the quality of their schedule. The agents return an *option* containing this information to the *requesting* agent. The *requesting* agent evaluates the responses and decides whether to commit to the option.

Figure 7.2 uses the same scenario described in the previous Section 7.1.1 to demonstrate how the pair of agents (A and B) can use this form of coordination to schedule activities M_1 and M_2. As before, agent B wants to install activity M_2 in its schedule, but it needs to coordinate with agent A the scheduling of the predecessor activity M_1. M_1 and M_2 are assumed to have a duration of 10, and the expected qualities for M_0 and M_2 are $q(M_0) = 20$ and $q(M_2) = 100$. The coordination process involves three steps:

1. The *requesting* agent B asks the *recipient* agent A to schedule activity M_1. The request includes the temporal bounds that M_1 must respect for activity M_2 to be installed successfully. The top left section of the figure shows agent B making this request. In this example, agent A specifies that activity M_1 must finish at

the latest by

$$LFT(M_1) = 30$$

to enable activity M_2 to be successfully scheduled.

2. Upon receipt of the request, agent A, creates a schedule that contains activity M_1 and imposes the desired LFT on it. Upon successfully creating such a schedule, agent A generates an *option* that specifies (1) whether it can fulfill the request (it can in this example), and (2) the *quality cost*, which conveys the quality lost by A's schedule in fulfilling this request. The bottom part of the figure shows agent A creating an option that unschedules activity M_0 in order to schedule M_1 with the desired LFT. The *quality cost* corresponds to the quality that agent A's schedule loses by unscheduling M_0, or 20 in this case.

3. After receiving agent A's response, agent B can decide whether to form a new inter-agent commitment to schedule M_1 and M_2 based on the *overall quality gain*, Q_{OG}. In this example, Q_{OG} is found by subtracting the *quality cost* to A's schedule when installing M_1 versus the *quality gain* to B's schedule when installing M_2.

$$Q_{OG} = q(M_2) - q(M_0) = 100 - 20 = 80$$

After determining a positive *overall quality gain*, agent B relays to agent A that they should commit to scheduling M_1 and M_2. Agent A installs the schedule it generated when creating the option (installing M_1 and dropping M_0), and agent B places M_2 in its schedule.

The two coordination approaches presented above are complementary. While the *passive* strategy provides the agents with a low-overhead mechanism that piggy-backs on the activity updates transmitted by the DSM, sometimes more directed actions are needed. An example is the situation shown in Figure 7.2, where agent A needs to retract activities from its schedule to install M_1. The *passive* coordination strategy we outlined would fail in this scenario because it only *adds* activities, but does not *retract* them. The *aggressive* strategy, on the other hand, allows agents to coordinate actions that may involve quality loses in some agents' schedules as long as the overall multi-agent schedule gains.

7.2 Updating Inter-Agent Commitments

Once agents have formulated a schedule involving a set of inter-agent commitments, execution dynamics (i.e. activity changes, or the addition of new activities/constraints) may require these commitments to be updated. This section explores how an STN framework can aid a team of scheduling agents in making coordinated schedule revisions. We develop strategies that use the conflict analysis tools of the STN to pinpoint potential coordination opportunities: When an agents fails to schedule a valuable *desired* activity because it conflicts with prior commitments, an analysis of the inconsistency can lead to the discovery of how these commitments can be modified to enable the activity to be installed on the schedule.

Temporal conflicts in the schedule result in conflicting temporal constraints in the STN, causing *negative cycles* in the STN's graph (see Section 2.1). The information encoded in these negative cycles can be extracted using *conflict explanation* techniques to discover what activities are in conflict. This information can then be used to guide a coordination process, where the agent attempting to install a valuable *desired* activity requests other agents that "own" conflicting activities to *shift* them temporally (either forwards or backwards in time), so that the *desired* activity can be installed. The process of modifying a multi-agent schedule to install a *desired* activity involves two steps:

1. *Determining the Set of Desired Activities*: First, an agent needs to determine the set of activities that would be *valuable* to install in its schedule, but whose installation failed [3]. These activities form the set of *desired* activities. *Desired* activities can be of two types:

   (a) *Intrinsically* valuable: These activities add *value* to the schedule through the quality they would accrue (i.e. the *contributors*, see Section 2.3.2).

   Activity M_1 in Figure 7.3 is *intrinsically* valuable because, if scheduled, it increases the quality of the $Mission$ taskgroup. If M_1 is not installed in agent A's schedule, then the quality of the taskgroup drops by at least 5 (or 15 if the redundant activity M_{1FF} is also unscheduled).

   (b) *Extrinsically* valuable: These activities add *value* to the schedule by sup-

[3] In this chapter, we use a generalized definition of activity installation. A failure to *install* an activity can not only indicate that the activity could not be added to the schedule, but also that the activity is scheduled, but *at the wrong time*.

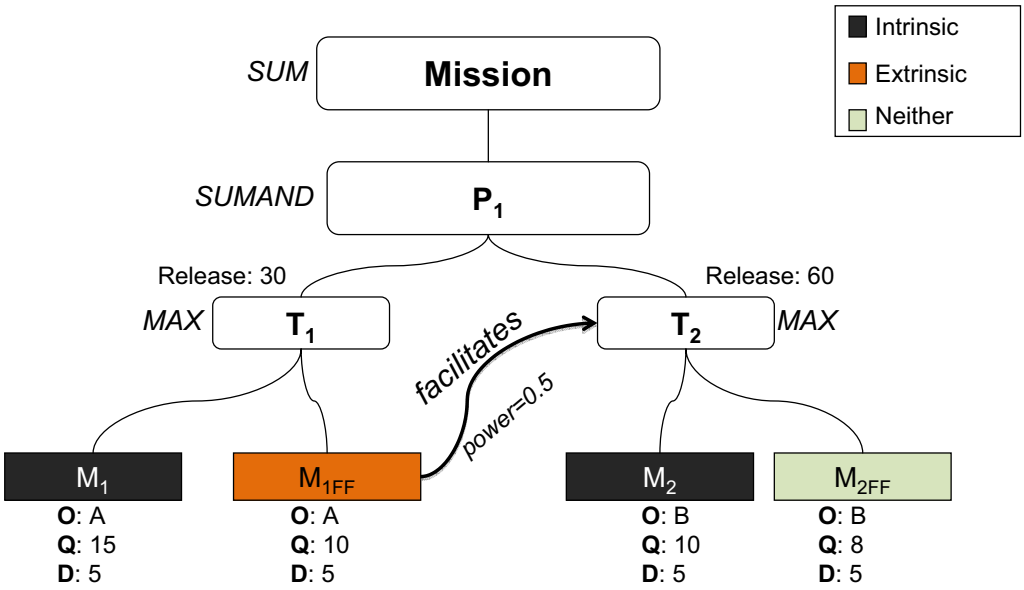

Figure 7.3: C_TAEMS Plan with Intrinsically and Extrinsically Valuable Methods

porting other activities, so that they can achieve higher quality. Typically this involves meeting plan preferences (i.e. such as soft NLEs in C_TAEMS, see section 2.2.1).

With M_1 scheduled, M_{1FF} is not *intrinsically* valuable, given that the parent task of both methods, T_1, is a MAX task, so the quality it accrues is determined by the maximum quality child, M_1. However, M_{1FF} is *extrinsically* valuable because it *facilitates* activity M_2. With M_{1FF} scheduled, and constrained to finish *before* M_2, the quality accrued by M_2 can be boosted from 10 to 15.

Naturally, an activity can be both *intrinsically* and *extrinsically* valuable at the same time.

2. *Coordinating with Other Agents a Joint Schedule Change*: Once the set of *desired* activities has been determined, the agent can attempt to install them by requesting a coordinated action involving other agents in the team that results in a change to the conflicting inter-agent commitments. We present two *conflict-driven* coordination strategies that leverage the information in the *negative cycles* returned by the STN to locate coordination opportunities. The two strategies present a trade-off between the amount of information that needs to

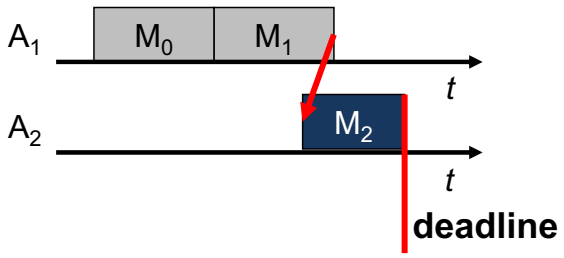

Figure 7.4: Agent A_2 is Unable to Schedule Method M_2

be shared between the agents, and the amount of time required for coordination. The first approach, the *recursive* strategy, requires less information to be made available to the agents, but needs more time to reach coordinated decisions. The second approach, the *full-chain* strategy, can make quicker decisions, but requires more information.

Both coordination approaches involve a three step procedure, where the agents introduce schedule changes in a "what-if" *hypothetical* mode, to see *what* would happen *if* they made the changes. All these *hypothetical* changes are retracted once the sought-after information is obtained. Our coordination strategies consist of a three-step process:

(a) *Request*: The request step involves the discovery of the *shifting set* (defined in the following paragraphs), and how this set is used to *request* changes from other agents. The agent that sends the initial *request*, starting the coordination process, is called the *initiating* agent.

When an agent cannot accomplish a scheduling action (e.g. installing the *desired* activity) because of a temporal conflict with an existing inter-agent commitment, it can leverage the conflict information obtained from the STN's *negative cycle* to locate the "remote" activities that need to *shift* (move forward or backward in time) to allow the scheduling action to proceed, and how much they need to shift [4].

Consider the situation in Figure 7.4. Activity M_1 needs to *shift* backwards, so that its earliest finish time (EFT) is at most equal to the latest start time

[4]The *shift* amount, as obtained from the STN's *negative cycle*, refers to how much a "remote" activity needs to move *past* its feasible temporal bounds, and not to any sliding within these bounds. In fact, sliding within the temporal bounds does not cause a *negative cycle*, since no constraints are violated.

(LST) of M_1 (see Section 2.4.2 for a description of the temporal bounds of an activity). The *shift* amount for activity M_1, sh, is then

$$sh = EFT(M_1) - LST(M_2)$$

This *shift* amount indicates how much the EFT of M_1 needs to *shift* backward, so that the enables NLE between M_1 and M_2 can be respected.

The "remote" activities that need to shift and their shift amounts are aggregated into the *shifting set*, S, which has the form

$$S = \{(A_i\ sh_i) \mid i \in [1, N]\} \tag{7.1}$$

where (1) A_i is a "remote" activity that needs to *shift*, (2) sh_i is the shift amount for activity A_i, and (3) N is the number of remote activities in the *shifting set*. After composing the *shifting set*, an agent sends a *request* to all the agents that "own" the activities in the *shifting set* to move them by the necessary amount.

(b) *Response*: An agent, R, upon the receipt of a *request* to *shift* one or more of its scheduled activities, complies with this request by imposing new temporary release and/or deadline constraints on them. The new constraints force these activities to meet the *requested* temporal bounds. If the *shift* requests the activities to move *forward* in time, temporary *release* constraints are introduced. These release constraints force the earliest start time (EST) of the activities to move forward. When, on the other hand, the *shift* requests the activities to move *backward* in time, temporary *deadline* constraints are introduced. These deadline constraints force the latest finish time (LFT) of the activities to move backward. Consider the example in Figure 7.4. Agent A_1 has received a *request* from agent A_2 to *shift* its activity M_1 backwards. Agent A_1 then imposes a deadline constraint on M_1 to accomplish the *shift*.

Once the *shift* is accomplished, the agent formulates a *response* that specifies $Q_{Lost}(R)$, the quality lost by its local schedule when *shifting* the activities specified in the *request*. This loss of quality occurs when the agent has to remove activities from its schedule to accomplish the *requested* shift. The procedure for finding the activities that need to be removed is simple: First, the agent removes any scheduled activities that are "in the way".

Once the shift is accomplished, it attempts to restore as many of these removed activities as possible. The remaining activities that the agent is unable to put back in the schedule cause the quality loss. For the example of Figure 7.4, agent A_1 needs to unschedule M_0 before the *shift* of M_1 can succeed, because M_0 is "in the way". However, once the shift is accomplish, agent A_1 can attempt to reinsert activity M_0 *after* M_1.

For C_TAEMS plans, a schedule's quality is computed by calculating the quality of the root task (or taskgroup, see section 2.2.1 for C_TAEMS details). Consequently, the exact quality loss can be computed by determining how the quality of the root node changes as the schedule is modified. When the view of the agents is *limited*, however, this calculation may not be exact (but it can be used as an estimate). For the work in this chapter, we provide all agents with a view of the plan where all activities are connected to the root node. This allows us to determine how changes to lower-level activities affect the root. If the original agent's view contains an activity that is disconnected from the root node (e.g. its parent node is unknown), we enlarge the view with all the "ancestor" tasks connecting this activity to the root node. Once the quality loss information has been calculated, it is sent back to the *requesting* agent.

(c) *Command*: When the *initiating* agent, L, receives the responses from all the "remote" agents, R_i, from whom it *requested* a *shift*, it determines whether to commit to the coordinated schedule change based on the differential between the *quality gain*, $Q_{Gain}(L)$, that its schedule would accrue if the change occurs, against the combined *quality costs*, $Q_{Lost}(R_i)$, of all the *responding* agents. The *initiating* agent sends a *command* to commit to the schedule update if:

$$Q_{Gain}(L) \; > \; \sum_i Q_{Lost}(R_i) \tag{7.2}$$

If a *command* is issued by the *initiating* agent, all the agents involved in the coordination process attempt to commit permanently to the schedule change. On some occasions, however, committing to these changes is no longer feasible. Execution dynamics may have changed the state of one or more of the agents in a way that prevents them from performing the desired change to their schedule. These situations resolve themselves as the agents transmit to each other the temporal bounds of their

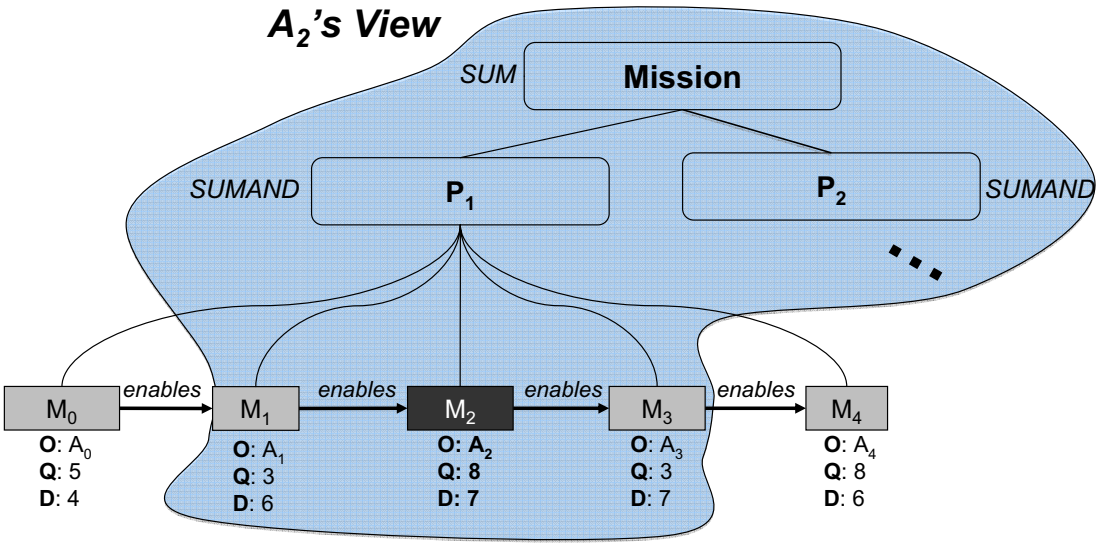

Figure 7.5: The View of Agent A_2 Includes the Immediate Predecessor and Successor

activities (through the normal DSM update process, see Section 2.3.2). If one or more of the agents was unable to commit to the coordinated schedule change, the other agents will realize that the schedule change has failed when they receive the (unchanged) temporal bounds of the activities that were to be *shifted*. At this point, these agents modify again their previously shifted activities to conform to the actual state of the schedule. An example of this situation can be illustrated using the situation in Figure 7.4. Suppose agent A_2 had sent a *command* to agent A_1 to shift M_1 backward to make space for M_2. In the meantime, however, agent A_1 had started executing A_0. Now, A_0 can no longer be switched forward after M_1, so agent A_1 backs out from making the commit, and keeps its original schedule. When agent A_2 receives the temporal bounds for activity M_1, it realizes that these have not changed as expected. At this point, agent A_2 unschedules M_2 since the enables NLE from M_1 to M_2 cannot be respected.

The next two sections describe the different approaches to this three-step coordination procedure that are taken by the *recursive* and *full-chain* coordination strategies.

7.2.1 Coordinating Schedule Changes Using the Recursive Strategy

An agent with a *limited* view of the planned activities may not be aware of all the activities that need to *shift* to accomplish a schedule change. The *recursive* coordination strategy presents a mechanism that these agents can use to coordinate updates to inter-agent commitments in the multi-agent schedule. The strategy equips agents with the ability to *shift* a *chain* of activities (i.e. a set of activities linked by precedence constraints) earlier or later in time through *recursive* requests. This strategy only assumes that agents are aware of *direct* predecessor and successor activities to their "local" activities. Figure 7.5 displays an example of the view of an agent when using the *recursive* strategy for coordination. The figure illustrates a small C_TAEMS plan containing a *chain* of five methods (M_0, M_1, M_2, M_3 and M_4). The view of agent A_2, the "owner" of method M_2 is shown (see Section 2.2.2 for an explanation of agent views). While agent A_2 is aware of the direct neighbors of M_2 (the predecessor M_1 and the successor M_3), it is not aware of M_0 nor M_4.

Using the *recursive* coordination strategy, an agent that "owns" a *desired* activity that it cannot schedule (due to conflicting temporal commitments to other agents) can *initiate* a coordination session to *request* that other agents *shift* their activities, so that the *desired* activity can be scheduled. However, given the agent's limited view of the plan activities, it may be unaware of potential problems the *recipient* agents may encounter in fulfilling the *request*. To accomplish the *shift*, these *recipient* agents may have to violate some current temporal commitments they have made. This obstacle is overcome by the issuing of *recursive* requests: A *recipient* agent that is unable to fulfill a *request* to *shift* one of its "local" activities due to a current temporal commitment to a third agent will in turn issue a *recursive* request to this agent to update the commitment. The *recursive* requests continue until a *recipient* agent is able to formulate a *response* to the *request* based solely on "local" information (i.e. no further inter-agent temporal commitment needs to be modified). This response *recurses* back to the *initiating* agent that started the coordination process. Once the *initiating* agent receives *responses* from all the agents it *requested shifts* from, it uses the information in these *responses* to determine whether the agents should proceed with the coordinated schedule update. If the decision is made to commit to the coordinated schedule change, the *initiating* agent sends a *command* to this effect to the *recipient* agents of its original *request*. These agents *recursively* forward the

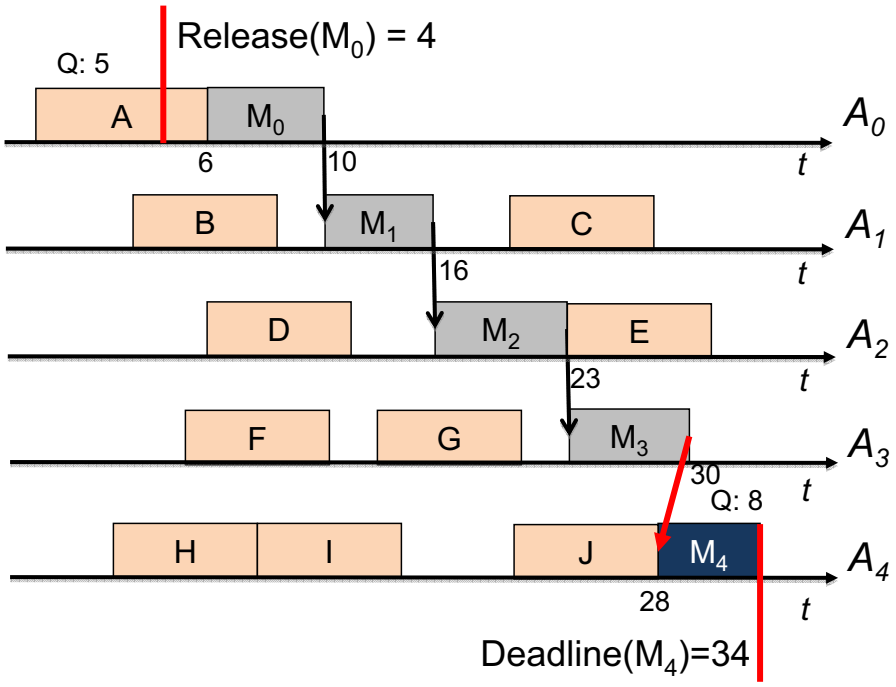

Figure 7.6: Agent A_4 is Unable to Schedule Method M_4

command down the *chain* of activities until all agents involved are informed.

Figure 7.6 uses the C_TAEMS plan shown in Figure 7.5 to illustrate a situation where agent A_4 is unable to schedule M_4 because M_3 is scheduled too late. Agent A_4 then *initiates* a coordination process by *requesting* agent A_3 to *shift* M_3 earlier in time. When agent A_3 receives the *request*, it in turn realizes that it cannot *shift* M_3 earlier unless the predecessor method M_2 also *shifts* earlier. A_3 then *recursively requests* agent A_2 to *shift* M_2. Upon receiving agent A_3's request, agent A_2 *recursively requests* agent A_1 to *shift* M_1 earlier, and finally agent A_1 *requests* agent A_0 to *shift* M_0 earlier. Agent A_0 can make a "local" decision, since *shifting* M_0 earlier does not conflict with any further temporal commitments made to another agent. Agent A_0 then formulates a *response* to agent A_1 which is forwarded *recursively* until it reaches the *initiating* agent A_4. Based on the *response*'s information, agent A_4 can determine whether to proceed with the coordinated schedule change. If the change would improve the multi-agent schedule, agent A_4 sends a *command* to commit to this change to agent A_3, which in turn, forwards it back through the *chain* of activities until it reaches agent A_0. These three steps of the coordination process are shown in

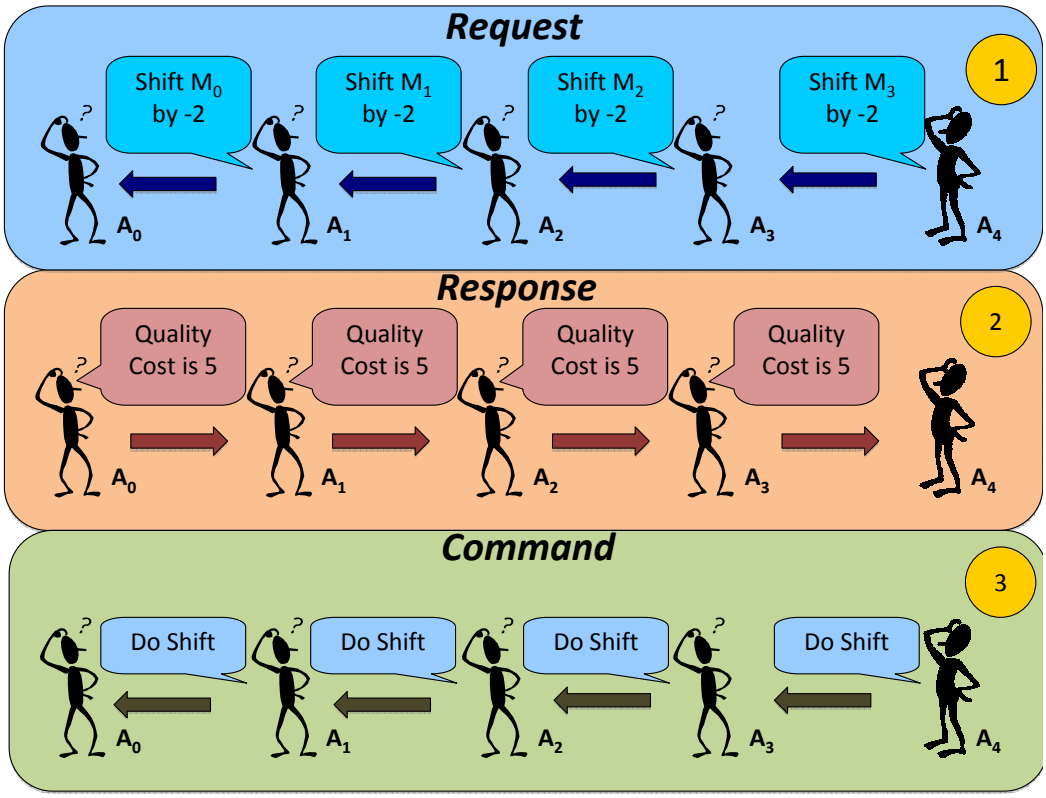

Figure 7.7: The Steps of a Recursive Coordination Process

Figure 7.7. The top of the figure shows *initiating* agent A_4 starting the coordination process with the *request* to A_3 and how this *request recurses* back to agent A_0. The middle part of the figure illustrates the *response* procedure, starting from the "local" *response* generated by agent A_0, which *recurses* down the *chain* back to agent A_4. Finally, the bottom section of the figure illustrates the *command* sent by agent A_4 to commit to the joint schedule change, which again *recurses* up the chain back to agent A_0.

The example above is an instance of a *backward* coordination process: "Remote" activities are asked to shift *earlier* (or *backward*) in time to make space for the desired method M_4. The direction of the coordination process, however, depends on the nature of the inconsistencies encountered. In general, our coordination mechanism starts by attempting a *forward* coordination process that tries to shift "remote" activities *later* (or *forward*) in time, and if this fails, a *backward* process is tried. These two processes are mirror images of one another. A *forward* coordination is attempted first

because the *backward* coordinations can potentially run into already *executed* sections of the schedule that can no longer be *shifted*. *Forward* coordinations, on the other hand, always work with *later pending* activities.

A *recursive* coordination process can then be seen as a series of "separate" coordination *sessions*, where a *requesting* agent asks a set of *recipient* agents to *shift* some of their "local" methods, and the *recipient* agents answer this *request* by formulating a *response*. This *response* contains (1) whether the *shift* is possible, and (2) the *quality cost* incurred in *shifting* the activities (e.g. some activities had to be unscheduled to accomplish the *shift*). Both the *requesting* agent and the *recipient* agents in these individual coordination *sessions* leverage their STNs conflict analysis tools. The *requesting* agent mines the *negative cycles* returned by the STN when attempting to install a *desired* activity (in the case of the *initiating* agent) or when attempting to *shift* a scheduled "local" activity as specified in the received *request*. The *negative cycles* point to what inter-agent commitments need to be modified to accomplish the installation or *shift*. Similarly, a *recipient* agent leverages the *negative cycles* returned by its STN to determine if it can make a "local" decision (i.e. no further inter-agent temporal commitments need to be modified), or it needs to make a *recursive request* to third agent.

The following sections will examine the *request* and *response* step in greater detail, explaining how the STN's *negative cycles* are exploited to locate the temporal commitments to be updated. Finally, we will conclude our discussion of the *recursive* coordination strategy by discussing the *command* process, where the *initiating* agent decides whether the multi-agent schedule should be updated based on the *responses* received.

Requesting a Coordinated Shift of Scheduled Activities

As mentioned previously, a *request* session is started when an *initiating* agent cannot install a *desired* activity at the *desired* time, or a *recipient* agent is unable to *shift* a "local" activity by the *requested* amount due to conflicting inter-agent temporal commitments. To overcome these obstacles, the agent *request* the agents with whom it has the conflicting commitments to coordinate a change in the multi-agent schedule. Each *request* step in our *recursive* coordination mechanism starts with the *requesting* agent discovering the temporal commitments that need to be modified and how. STN-based scheduling agents can leverage the conflict analysis tools of their STNs to obtain

the needed information: By using *conflict explanation* techniques on the *negative cycles* returned by the STN when the *desired* activity fails to install (or *shift*), an agent can discover what "remote" activities need to *shift* to allow the *desired* action, and by how much.

For an STN-based scheduling agent, the scheduling or shifting of a *desired* activity involves the introduction of constraints into the STN. Any addition of a temporal constraint into the STN carries the possibility of a temporal inconsistency, resulting in a *negative cycle* in the STN's graph. In general, scheduling a currently unscheduled *desired* activity (such as an *intrinsically* valuable activity) involves the introduction of the following constraints [5]:

1. *Duration Constraint*: This constraint enforces the duration of an activity in the STN, as specified in the plan.

2. *Plan-defined Precedence Constraints*: This set of constraints enforce the predecessor-successor relationships of the activity, as defined by the plan (e.g. enables NLEs in C_TAEMS).

3. *Scheduler-defined Precedence Constraints*: These constraints enforce resource limitations. In our domain, where an agent can execute only one activity at a time, these constraints enforce the sequencing of activities in an agent's timeline. Using these *sequencing* constraints, the *desired* activity is installed by the agent's scheduler between two currently scheduled activities.

In addition, preference relations (for *extrinsically* valuable activities) or transient "release" / "deadline" constraints (to enforce a *requested shift*) may need to be introduced in the STN.

When one of these temporal constraints *fails* to insert into the STN, the *negative cycle* produced can be analyzed for potential inter-agent commitments that can be updated to allow the constraint insertion. An absence of any inter-agent commitments in the *negative cycle* indicates that there are no coordination opportunities that would enable the *desired* activity to install, or *shift*. If, on the other hand, the *negative cycle* involves an inter-agent commitment (e.g. a precedence constraint between a "remote" activity and a "local" activity), a coordination opportunity may be available. We

[5] We assume that time window constraints (i.e. release or deadline constraints), and hierarchical constraints (forming parent-child relations) are already inserted in the STN, even if the activity is unscheduled.

conduct a three step process to determine which inter-agent commitments need to be updated, and how:

1. First, using *conflict explanation* techniques, we analyze the *negative cycle* to locate the "remote" activity that conflicts with the installation of the *desired* activity, and by how much this activity needs to *shift*.

2. Then, once this information has been extracted, the inconsistency is *resolved* by temporarily *retracting* from the "local" STN the temporal constraints that prevent the "remote" activity from *shifting* (e.g. temporal relations constraining the EST or LFT of the activity, depending on the direction of the *shift*).

3. Finally, once the inconsistency is *resolved*, we attempt to reinsert the original *failed* constraint into the STN. If the insertion is successful, no further "remote" activities need to *shift* to allow the *desired* activity's installation. Otherwise, further coordination may be necessary. The new *negative cycle* is analyzed using the same procedure.

We will illustrate this *request* process by using the example in Figure 7.6 where agent A_4 tries and fails to install method M_4. The *request* process involves three types of agents:

1. *The Initiating Agent*: The agent that starts the coordination process to install a *desired* activity in the schedule. In the example, the *initiating* agent is agent A_4.

2. *A Middle Agent*: One of the agents "in the middle". In the example, agents A_1, A_2 and A_3 are *middle* agents.

3. *The Responding Agent*: The last agent in the *chain*. This agent can formulate a *response* based on "local" information and starts the *response* process. In the example, the *responding* agent is A_0.

We will now illustrate the reasoning of one of each type of agents, showing how each uses the *negative cycle* returned by their STN's to drive the coordination process.

The Initiating Agent The *initiating* agent starts the coordination process. It uses the *negative cycles* returned by its STN when installing the *desired* activity to find

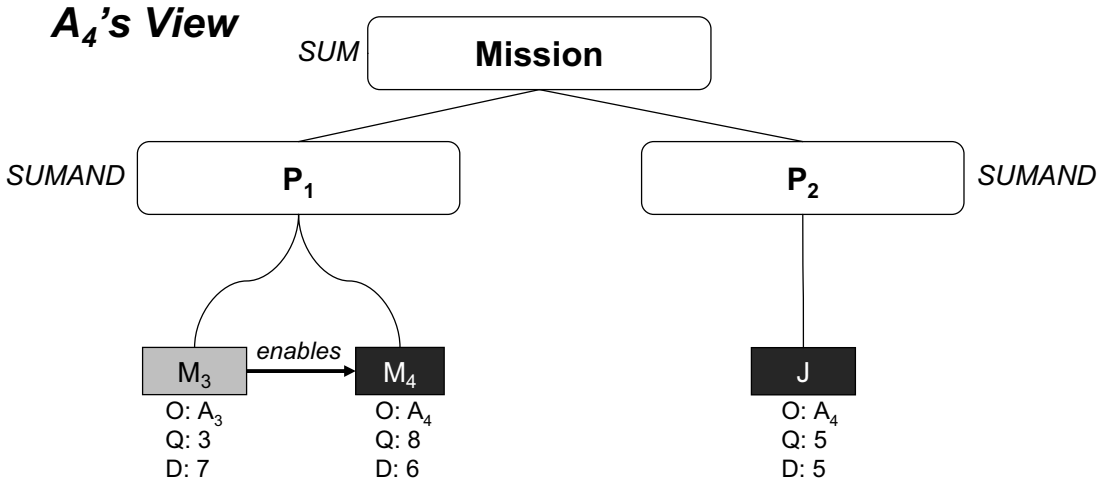

Figure 7.8: The View of Agent A_4

the *shifting set*. We will illustrate how an *initiating* agent finds the *shifting set* using the example in Figure 7.6, In this example, the *initiating* agent is A_4 attempts to schedule the *intrinsically* valuable *desired* activity M_4. The view of agent A_4 for this scenario is shown in Figure 7.8. Agent A_4 is aware of its "own" methods, J and M_4, as well as M_3, the immediate predecessor to the *desired* method M_4. However, A_4 is not aware of the rest of the *chain* of activities preceding M_3. While agent A_4 is able to insert the duration constraint of M_4 and a *sequencing* constraint between the "local" method J and M_4, the insertion of the *enables* NLE between the "remote" method M_3 and the *desired* method M_4 fails. The resulting *negative cycle* produced by the STN is shown in Figure 7.9. The edges in the *negative cycle* are shown in red. Using conflict analysis of this *negative cycle* reveals that the activities in conflict are the "remote" method M_3 and the "local" method M_4. To successfully schedule M_4, agent A_3 would need to *shift* its method M_3 earlier (or *backward*). The addition of the magnitudes of the edges in the cycle, $34 - 6 - 7 - 23 = -2$, reveals the amount that method M_3 would need to *shift* to make enough space for M_4. The *shifting set*, S, then is given by

$$S = \{ (M_3 , -2) \}$$

After this *negative cycle* is analyzed, the agent *resolves* the inconsistency by temporarily *retracting* the *release* constraint of M_3 from its STN. Removing this constraint

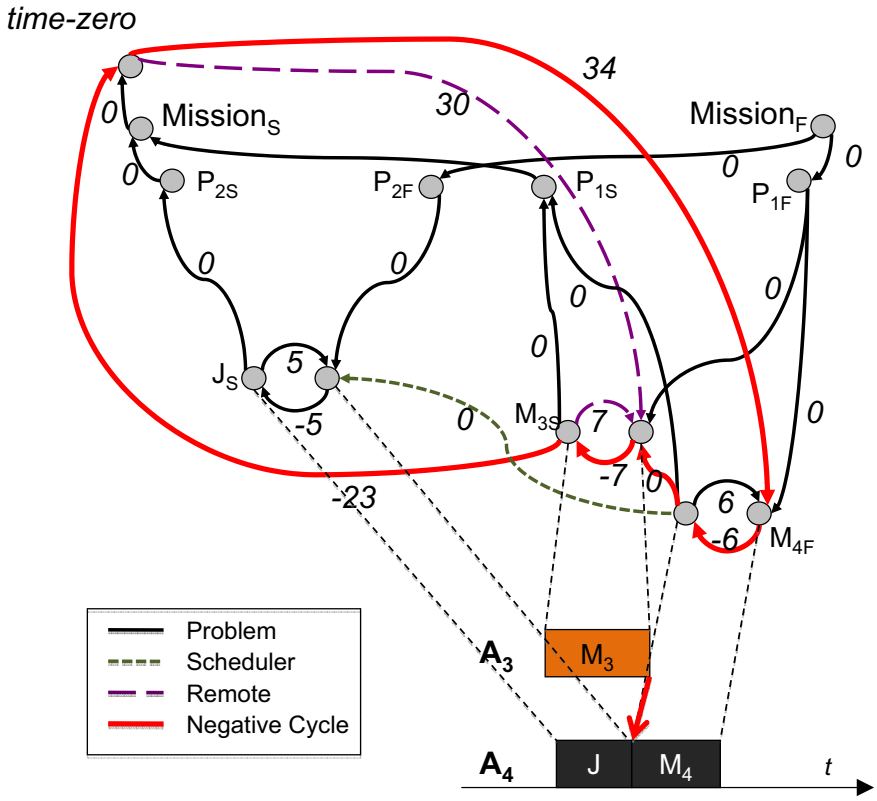

Figure 7.9: The Negative Cycle in the STN of Agent A_4

from the STN has the effect of allowing M_3 to *shift backward*. Once this constraint is removed, the *enables* NLE between M_3 and M_4 can be inserted without further conflicts. The resulting conflict-free STN is shown in Figure 7.10. Once the *enables* NLE between M_3 and M_4 is inserted into the STN, the *desired* method M_4 is successfully installed *provided* M_3 is *shifted backward* by the necessary amount. At this point, agent A_4 sends a *request* to agent A_3 to *shift* M_3 *backward* by 2 time units, starting the coordination process.

A Middle Agent A *middle* agent "owns" one of the activities in the middle of the activity *chain* that needs to be *shifted* to allow the *initiating* agents' *desired* activity to install. Consequently, the *middle* agent is unable to *shift* its activity until the other agents further down the chain do. Thus, to formulate its *response*, the *middle* agent has to pass on the *request* to the "next" agent (or agents) in the activity *chain*, and receive its *response*. An analysis of the *negative cycle* that results when the *middle*

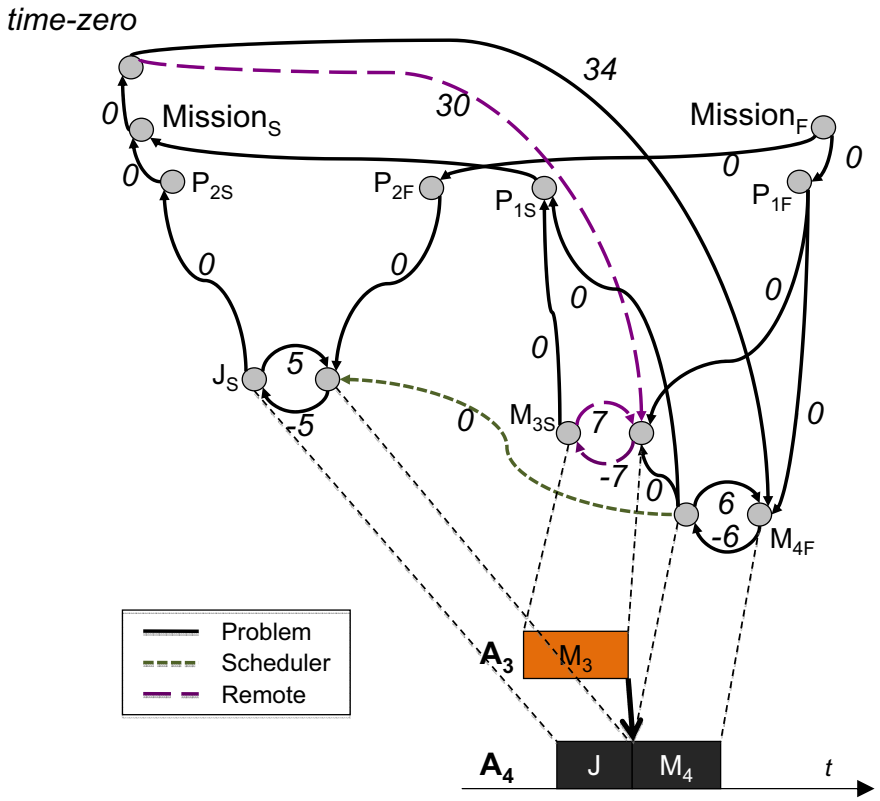

Figure 7.10: The Conflict-Free STN of Agent A_4

agent attempts to *shift* its "local" activity identifies the "remote" activities down the chain that need to move, and by how much. This information forms the *middle agent*'s *shifting set*. This information is used to generate a *recursive request* to the "owners" of these activities.

We will illustrate the *recursive request* process using the *middle* agent A_3 in the example shown in Figure 7.6. The view of agent A_3 is shown in Figure 7.11. Agent A_3 is aware of its "local" activities, as well as M_2 and M_4, the immediate predecessor and successor activities to M_3. After receiving the *request* from agent A_4 to *shift* the "local" method M_3 *backward* by 2, agent A_3 attempts to insert a temporary *deadline* constraint (at time 28) on M_3 that accomplishes the *shift*. The insertion of this constraint causes an inconsistency, and the resulting *negative cycle* returned by the STN is shown in Figure 7.12. Using conflict analysis on this *negative cycle* reveals that the conflicting activity is the "remote" method M_2. To successfully *shift* the "local" method M_3 *backward* by 2, agent A_2 would need to *shift* its method M_2 also.

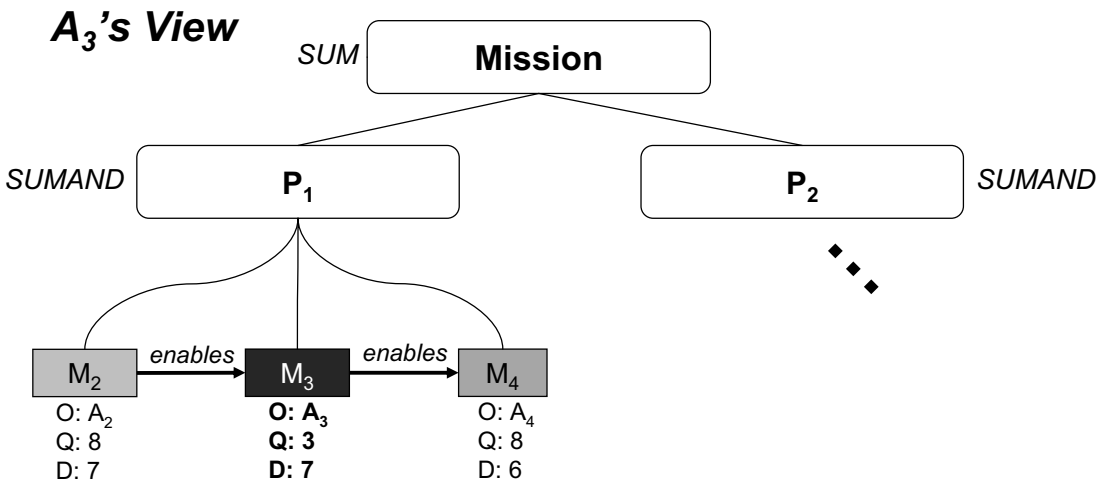

Figure 7.11: The View of Agent A_3

The addition of the magnitudes of the edges in the cycle, $28-7-7-16=-2$, reveals that method M_2 also needs to *shift* by 2 to make enough space for M_3's *shift*. The *shifting set*, S, then is given by

$$S = \{\ (M_2\ ,\ -2)\ \} \tag{7.3}$$

Similarly to the *initiating* agent, agent A_3 *resolves* the inconsistency by temporarily *retracting* the *release* constraint of M_2 from its STN. By removing this constraint from the STN, the "remote" method M_2 can *shift backward*, and the temporary *deadline* constraint for M_3 can be now inserted without further conflicts. The resulting conflict-free STN is shown in Figure 7.13. The insertion of the temporary *deadline* constraint on M_3 *shifts* this method *backward* by the requested amount, *provided* that M_2 is also *shifted backward*. At this point, agent A_3 sends a *recursive request* to agent A_2 to *shift* M_2 *backward* by 2 time units, continuing the coordination process.

The Responding Agent The *responding* agent is the agent that can *respond* to a *request* by making "local" decisions (i.e. scheduling changes to its "local" activities), with no need for further *recursive requests*. Unlike a *middle* agent, the *negative cycle* that results when this agent attempts to *shift* its "local" activity exposes a "local" conflict (i.e. a temporal conflict between "local" activities). This conflict can be removed by temporarily updating the schedule of the agent. This resulting schedule

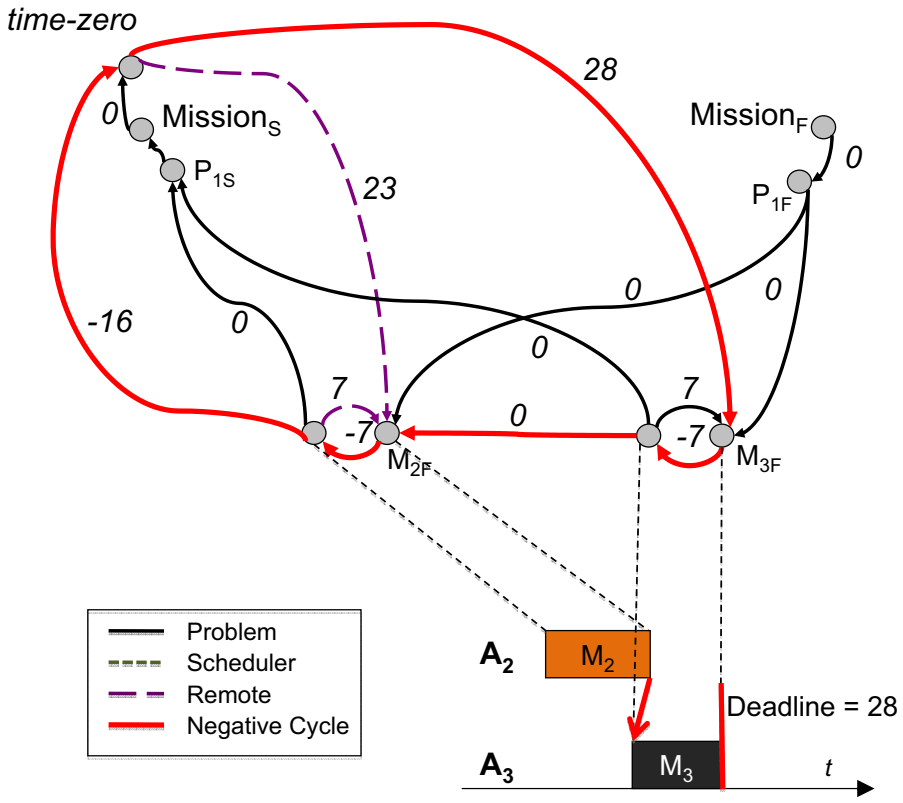

Figure 7.12: The Negative Cycle in the STN of Agent A_3

is used to generate a *response* to the *request*. The *response* contains the *quality cost* incurred for making the schedule change (e.g. a drop in quality caused by the unscheduling of "local" activities to accomplish the *shift*).

We can illustrate the actions of the *responding* agent using agent A_0 in the example shown in Figure 7.6. The view of agent A_0 is displayed in Figure 7.14. Agent A_0 is aware of its "local" activities, A and M_0, as well as M_1, the immediate successor activity to M_0. After receiving the *request* from agent A_1 to *shift* the "local" method M_0 *backward* by 2, agent A_0 attempts to insert a temporary *deadline* constraint (at time 8) on M_0 that accomplishes the *shift*. The insertion of this constraint causes an inconsistency, and the resulting *negative cycle* returned by the STN is shown in Figure 7.15. Using conflict analysis of this *negative cycle* reveals that the conflicting activity is the "local" method A. To successfully *shift* the "local" method M_0 *backward* by 2, this method needs to be unscheduled [6]. The resulting conflict-free STN after

[6]The scheduler may try to place the unscheduled method later in the timeline if possible

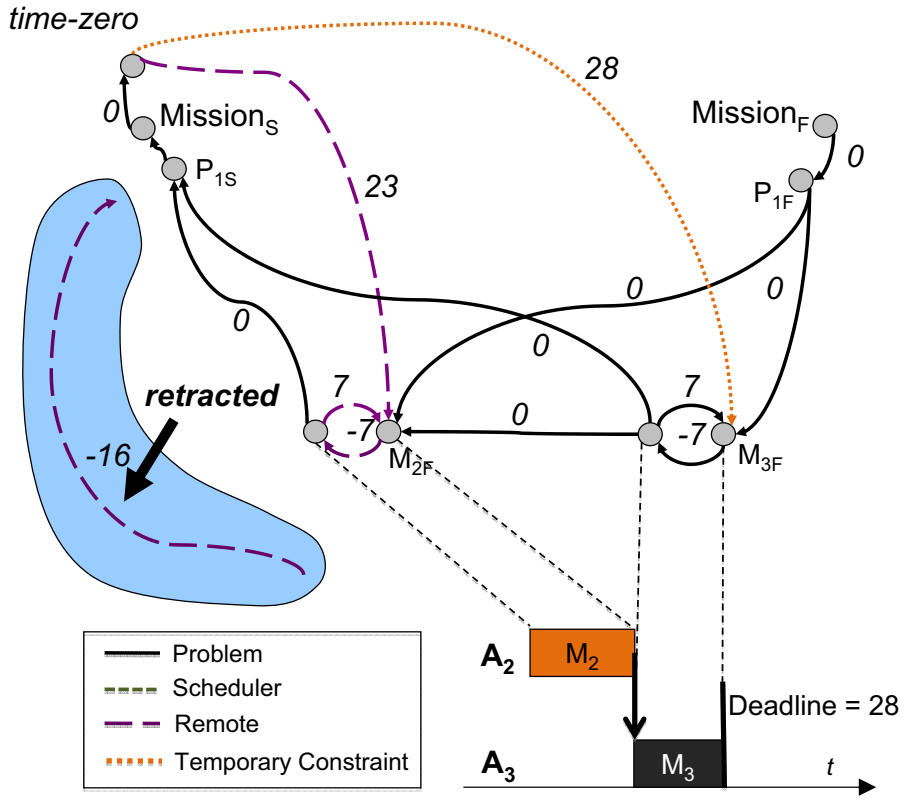

Figure 7.13: The Conflict-Free STN of Agent A_3

unscheduling method A is shown in Figure 7.16.

Once the "local" schedule has been updated, and the temporary *deadline* constraint for method M_0 successfully inserted into the STN, agent A_0 is poised to formulate a *response* to the *request*. The next section examines the mechanisms of the *response* process.

Responding to a Coordinated Shift Request

In the *response* step, an agent evaluates how its schedule would be affected if the coordinated schedule change specified in the *request* were to occur (if it can fulfill the *request*). To compute this effect, the agent, L, calculates the *quality cost*, $Q_{LocalLost}(L)$, to its schedule (the loss in quality incurred by changing the schedule to accommodate the *request*). This *quality cost* is simply the quality *differential* between the current

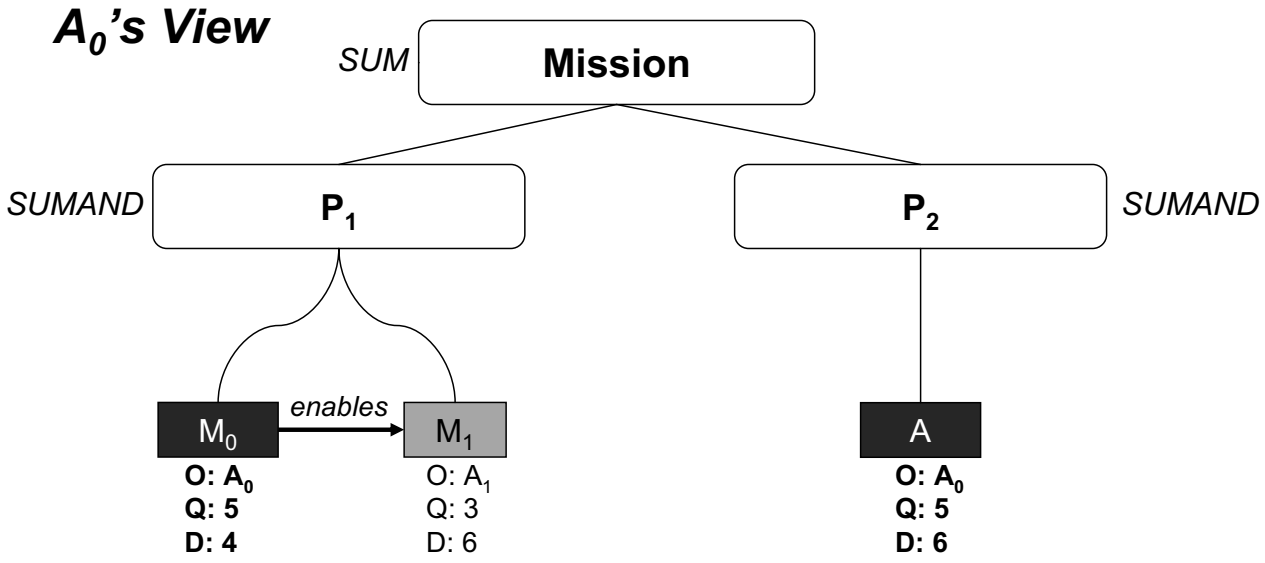

Figure 7.14: The View of Agent A_0

agent's schedule, and the new schedule that results after fulfilling the *request*:

$$Q_{LocalLost}(L) = Q_{After} - Q_{Before} \tag{7.4}$$

where Q_{Before} is the schedule quality before fulfilling the *request*, and Q_{After} the schedule quality after fulfilling the *request*.

While computing the "local" *quality cost*, $Q_{LocalLost}(L)$, is sufficient for the *responding* agent, a *middle* agent also need to aggregate the *quality costs* of the agents from whom it *requested* schedule changes. For each of these "remote" agents, R_i, the *middle* agent adds the *quality cost*, $Q_{RemoteLost}(R_i)$, to the "local" loss to form the combined *quality cost*, $Q_{Lost}(L)$.

$$Q_{Lost}(L) = Q_{LocalLost}(L) + \sum_i Q_{RemoteLost}(R_i) \tag{7.5}$$

After computing this *quality cost*, the agent sends a *response* to the *requesting* agent that contains this information. An agent that could not accommodate the *request* sends a null *response*, causing the coordination process to die out.

An example of this process can be illustrated using agent A_1 in the example of Figure 7.6. After the *responding* agent A_0 informs agent A_1 that the *quality cost* to

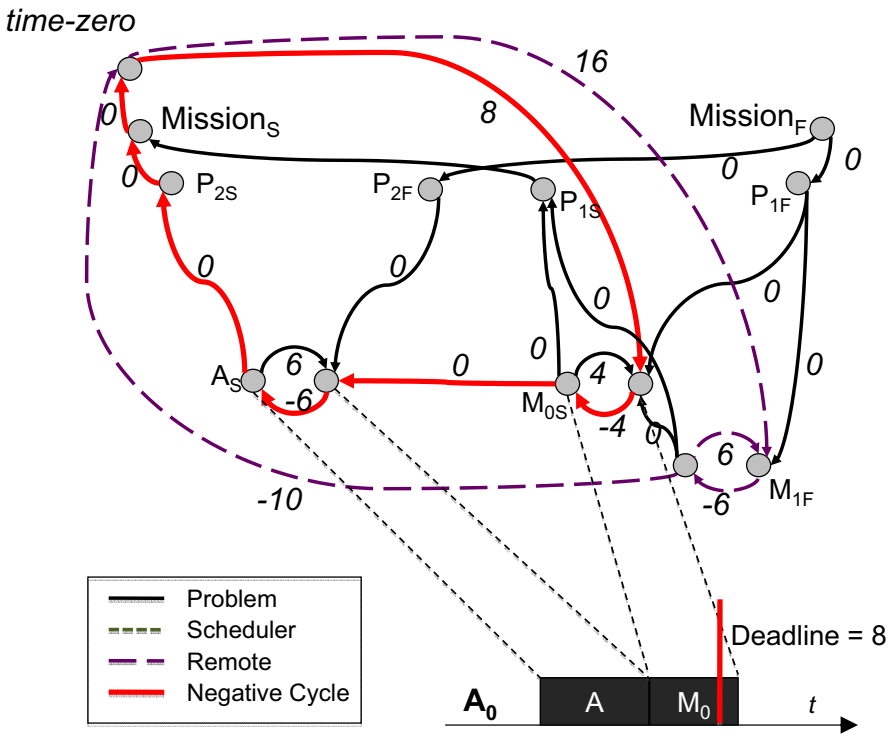

Figure 7.15: The Negative Cycle in the STN of Agent A_0

its schedule is 5, agent A_1 formulates a *response* to agent A_2 with

$$\begin{aligned} Q_{Lost}(A_1) &= Q_{LocalLost}(A_1) + Q_{RemoteLost}(A_0) \\ &= 0 + 5 = 5 \end{aligned}$$

Committing to the Coordinated Schedule Change

The final step of the *recursive* strategy occurs once the *responses recurse* back to the *initiating* agent. With this information at hand, the *initiating* agent determines whether the schedule change would improve the collective multi-agent schedule. If so, it issues a *command* to commit to this change. As in the *request*, this *command recurses* to all *recipient* agents until the *responding* agent is reached.

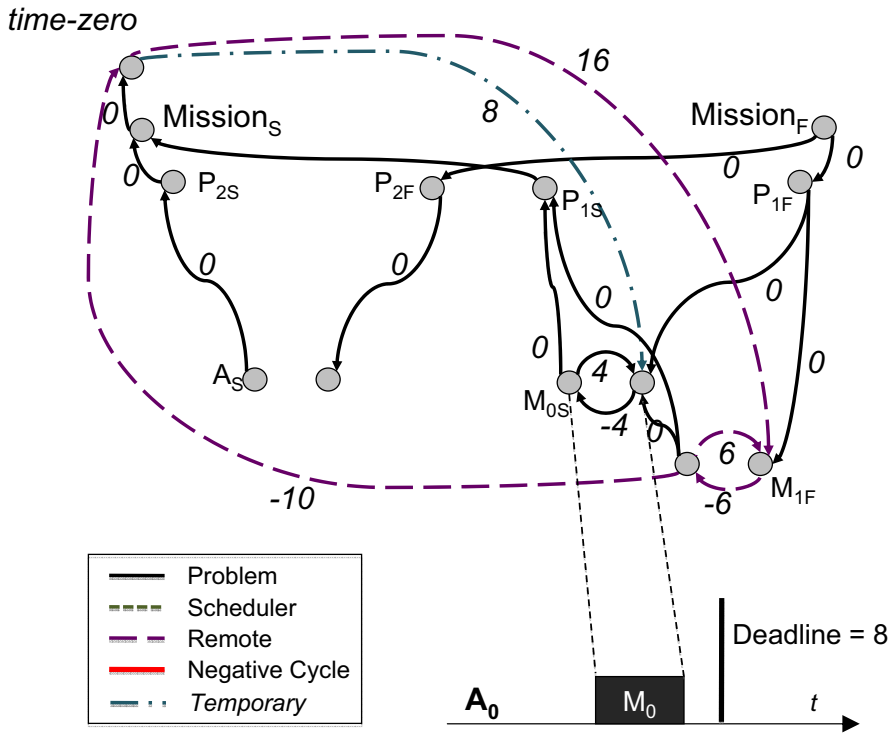

Figure 7.16: The Conflict-Free STN of Agent A_0

7.2.2 Coordinating Schedule Changes Using the Full-Chain Strategy

Like the *recursive* strategy described in the previous section, the *full-chain* strategy provides STN-based scheduling agents with a mechanism to coordinate changes across a *chain* of activities. However, instead of using *recursive requests* to reach all the agents involved, the *full-chain* strategy uses a single *request* to communicate with all involved agents at once and coordinate a joint change to the multi-agent schedule. To make this possible, the *full-chain* strategy requires some additional information when compared to the *recursive* strategy. The *full-chain* strategy assumes that the *view* of an agent that "owns" an activity in a precedence *chain* (i.e. as specified by the plan) contains all the activities in the *chain*. Take for example, the *view* of agent A_2 for the small C_TAEMS plan shown in Figure 7.17. The plan contains an activity *chain* of five methods, each "owned" by a different agent. Agent A_2 "owns" the method M_2 in the middle of the *chain*. The *view* of agent A_2 includes all the *predecessor* and *successor* activities to its "local" method (in other words, the full *chain*). The *views*

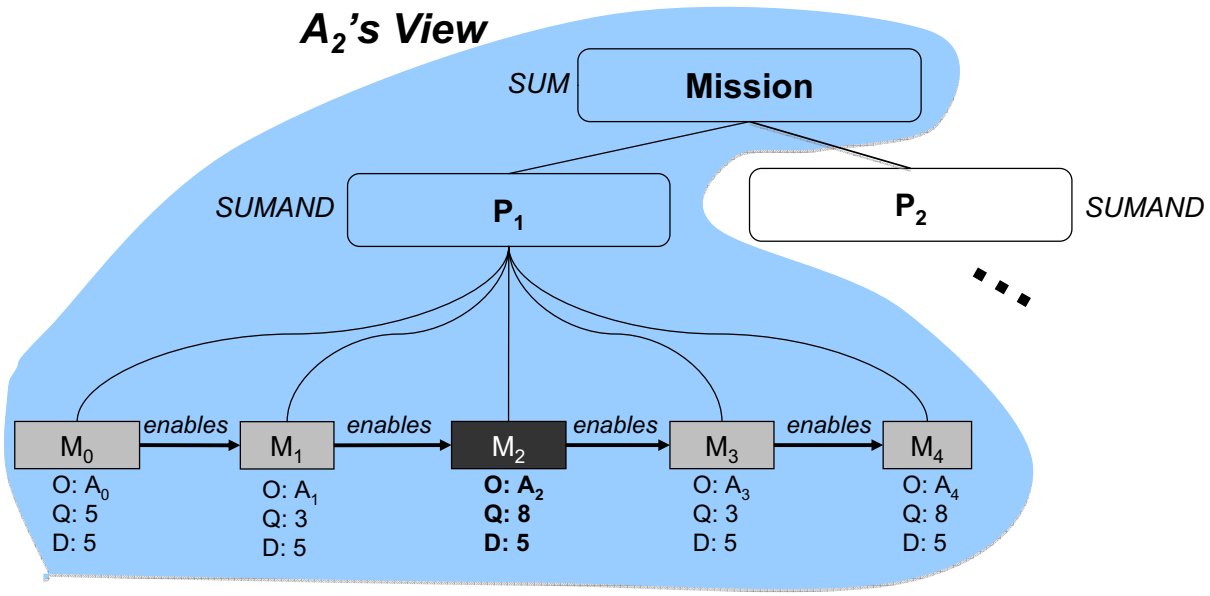

Figure 7.17: The View of Agent A_2 Includes the Whole Chain

of all the other agents involved also includes the full *chain* of activities.

The expanded *view* provided to the agents in the *full-chain* strategy implies greater communication costs, as the agents that "own" activities in the chain keep each other informed about changes to any of these activities (e.g. a change in the temporal bounds). However, with the expanded *view*, an *initiating* agent that has *failed* to install a *desired* activity can coordinate concurrently with all other agents involved in the *chain* to *shift* the activities *forward* or *backward* in time. The *initiating* agent sends *requests* to all agents *involved*. The *requests* sent to each of these agents specifies *all* methods that need to undergo a *shift*. Using this information, when the *recipient* agents *receive* the coordination *request*, they can generate a *response* assuming that all the "remote" methods that have been *requested* to *shift* will do so. These *responses* are sent back to the *initiating* agent, which makes a decision as to whether to proceed with the coordinated schedule change.

We can illustrate the differences between the *recursive* strategy and the *full-chain* strategy by reusing the example shown in Figure 7.6. Agent A_4 has *failed* to schedule the *desired* "local" method M_4 because the *predecessor* method M_3 is scheduled too late. Given the expanded *view* available to agent A_4, it is aware that for agent A_3 to *shift* method M_3 *backward*, M_3's predecessor methods M_0, M_1 and M_2 also need to

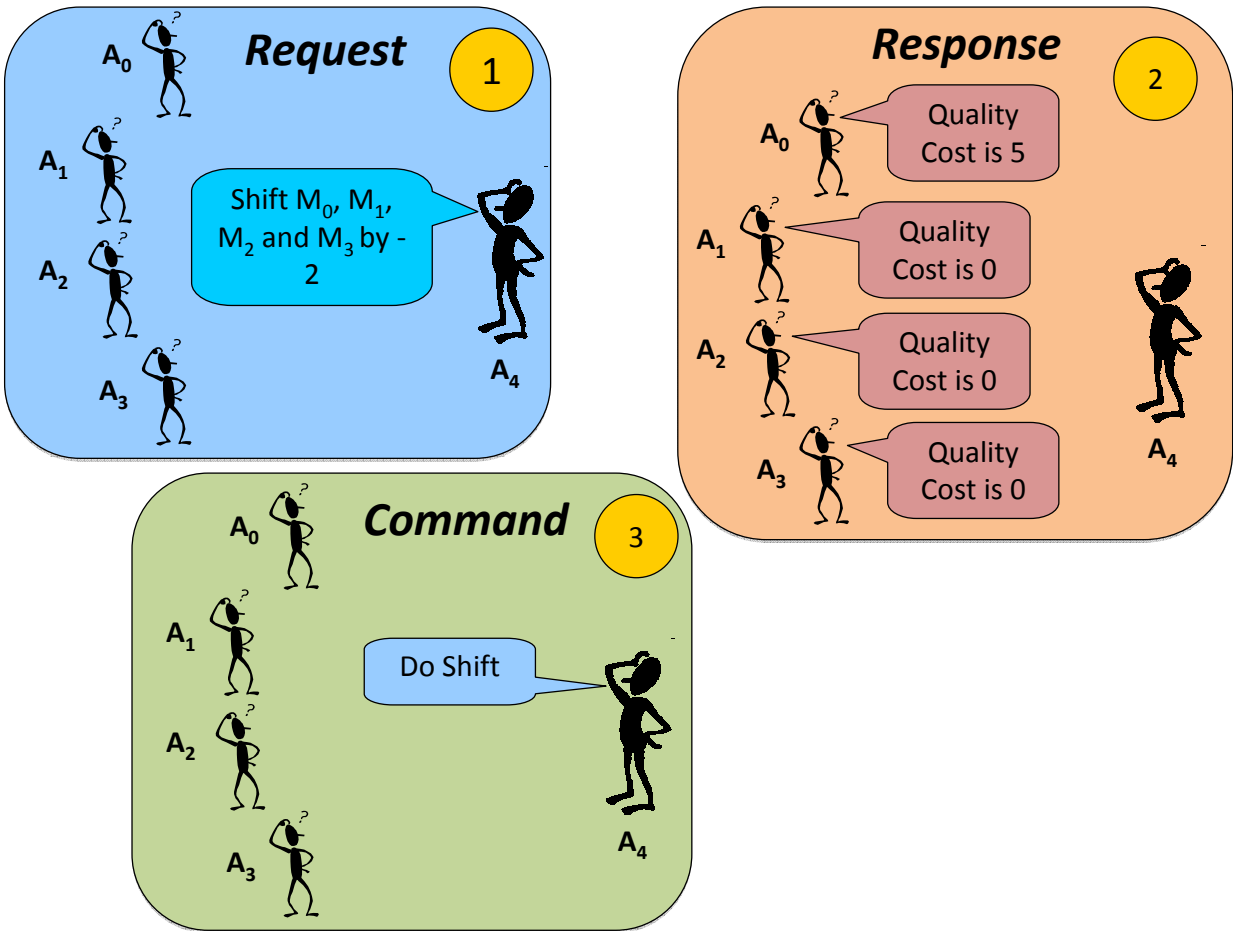

Figure 7.18: The Steps of a Full-Chain Coordination Process

shift backward. Agent A_4 *initiates* concurrent *requests* to all of the *involved* agents: A_0, A_1, A_2 and A_3 to achieve a coordinated *shift* of the *chain* of methods M_0, M_1, M_2 and M_3. The *recipient* agents formulate a *response* that *shifts* their "local" method and *assumes* all the other *chain* methods will be *shifted* (e.g. agent A_3 creates a *response* that *shifts* the "local" method M_3 and assumes the predecessor methods M_0, M_1 and M_2 will also be *shifted*). Finally, when agent A_4 receives all the *responses*, it makes a decision whether the *shift* should occur. If so, it issues a *command* to all *involved* agents to *commit* to the coordinated schedule change. These three steps of the coordination process are shown in Figure 7.18. The top left portion of the figure shows *initiating* agent A_4 starting the coordination process with a *request* to agents A_0, A_1, A_2 and A_3 to *shift* the activity *chain*. The top right part of the figure

illustrates the *response* procedure, where all the *recipient* agents send a *response* to agent A_4. Finally, the bottom section of the figure illustrates the *command* sent by agent A_4 to all the *involved* agents to commit to the joint schedule change. The following sections will examine this three-step process in greater detail.

Requesting a Coordinated Shift of Scheduled Activities

As mentioned previously, several temporal constraints need to be inserted into the STN to complete the installation of the *desired* activity, and any of these insertions can result in a temporal inconsistency (see section 7.2.1 for a more detailed description of the installation process). As explained previously, when the insertion of one these temporal constraints *fails*, an analysis of the *negative cycle* produced by the STN can lead to the discovery of inter-agent commitments that can be updated to allow the constraint insertion (the absence of any inter-agent commitments in the *negative cycle* indicates that there are no coordination opportunities that would enable the installation of the *desired* activity). A three step process determines the inter-agent commitments that need to be updated, and how:

1. First, we analyze the *negative cycle* using *conflict explanation* techniques to locate the "remote" activity that prevents the installation of the *desired* activity, and determine how much it needs to *shift*.

2. Then, the inconsistency is *resolved* by temporarily *retracting* from the "local" STN the temporal constraints that prevent the "remote" activity from *shifting* (e.g. release or deadline constraints)

3. Finally, we attempt to reinsert the original *failed* constraint into the STN. The process is repeated if a new *negative cycle* arises.

An illustration of the *full-chain request* process can be provided by using the example in Figure 7.6 where agent A_4 tries and fails to schedule the *intrinsically* valuable *desired* method M_4. The view of agent A_4 for this scenario is shown in Figure 7.19. Agent A_4 is aware of its "own" methods, J and M_4, as well as all the *chain* methods M_0, M_1, M_2 and M_3. During the scheduling of M_4, agent A_4 is able to insert the duration constraint of M_4 and a *sequencing* constraint between the "local" method J and M_4. However, the insertion of the *enables* NLE between the "remote" method M_3 and the *desired* method M_4 fails. The initial *negative cycle* produced by

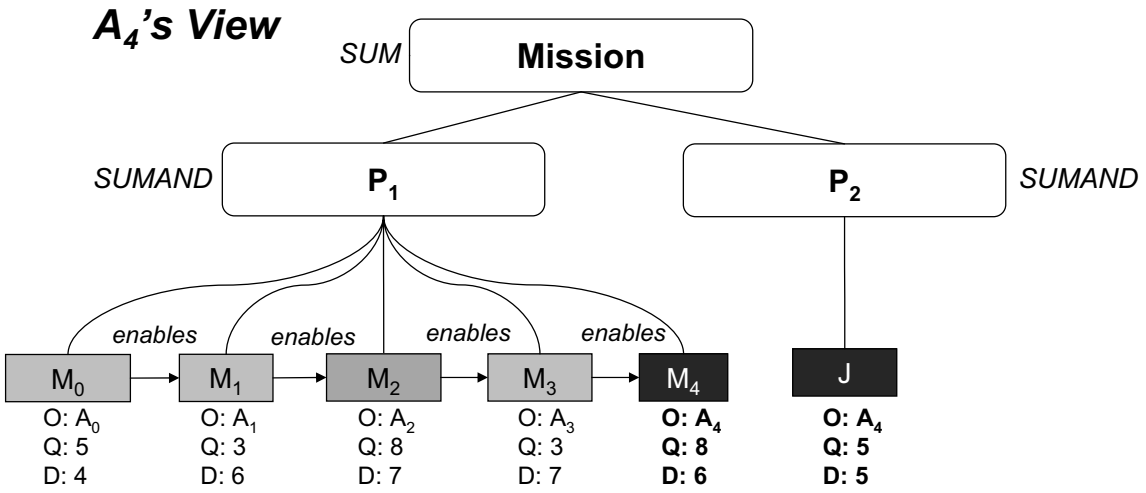

Figure 7.19: The View of Agent A_4

the STN is shown in Figure 7.9. The edges in the *negative cycle* are shown in red. Using conflict analysis of this *negative cycle* reveals that the *enables* NLE between the "remote" method M_3 and the "local" method M_4 cannot be inserted into the STN unless M_3 *shifts* earlier (or *backward*). The addition of the magnitudes of the edges in the cycle, $34 - 6 - 7 - 23 = -2$, reveals that method M_3 needs to *shift* by 2. We add this information to the *shifting set*, S.

$$S = \{ (M_3 , -2) \}$$

The *negative cycle* is *resolved* by temporarily *retracting* the *release* constraint of M_3 from the STN. Removing this constraint from the STN has the effect of allowing M_3 to *shift backward* in time. After *resolving* this inconsistency, agent A_4 attempts to insert the *enables* NLE between M_3 and M_4 once more. A new *negative cycle* is revealed, as shown in Figure 7.21. An analysis of this second *negative cycle* reveals that M_3 is unable to *shift backward* unless method M_2 also *shifts*. The *negative cycle* is *resolved* by *retracting* the *release* constraint of M_2 from the STN. When attempting once more to insert the *enables* NLE between M_3 and M_4, a third *negative cycle* appears, showing that method M_1 needs to *shift backward*, and then a fourth *negative cycle* shows that method M_0 needs to *shift backward*. After the *release* constraint of method M_0 is retracted from the STN, the *enables* NLE can be inserted without further conflicts. The resulting conflict-free STN is shown in Figure 7.22. The final

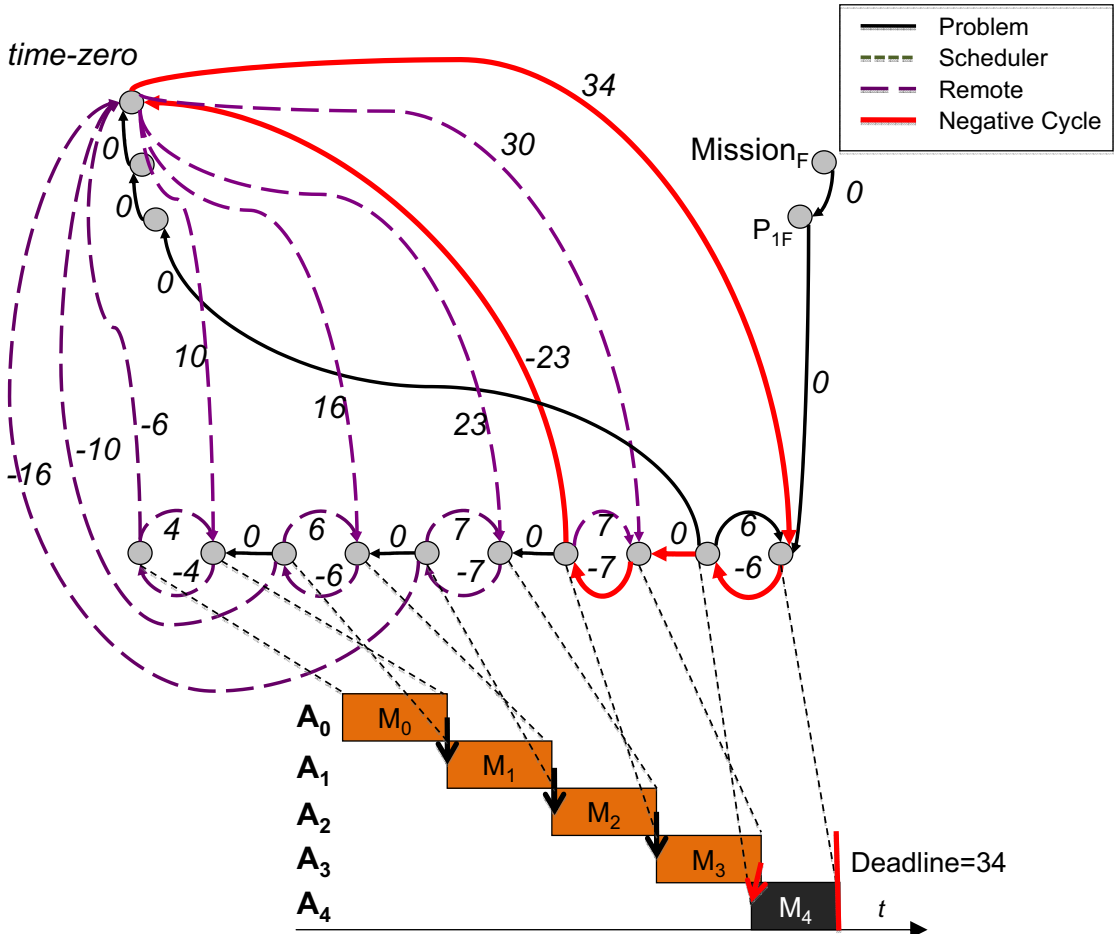

Figure 7.20: The Initial Negative Cycle in the STN of Agent A_4

shifting set, S, is given by

$$S = \{ (M_0\ ,\ -2)\ (M_1\ ,\ -2)\ (M_2\ ,\ -2)\ (M_3\ ,\ -2) \}$$

At this point, agent A_4 sends a *request* to agents A_0, A_1, A_2 and A_3 to *shift* their methods *backward* by 2 time units, starting the coordination process.

Responding to a Coordinated Shift Request

Unlike agents using the *recursive* strategy, all *recipient* agents in the *full-chain* strategy are provided with enough information to formulate *responses* independently of other agents involved in the *request*. The *request* specifies all activities, both "local" and

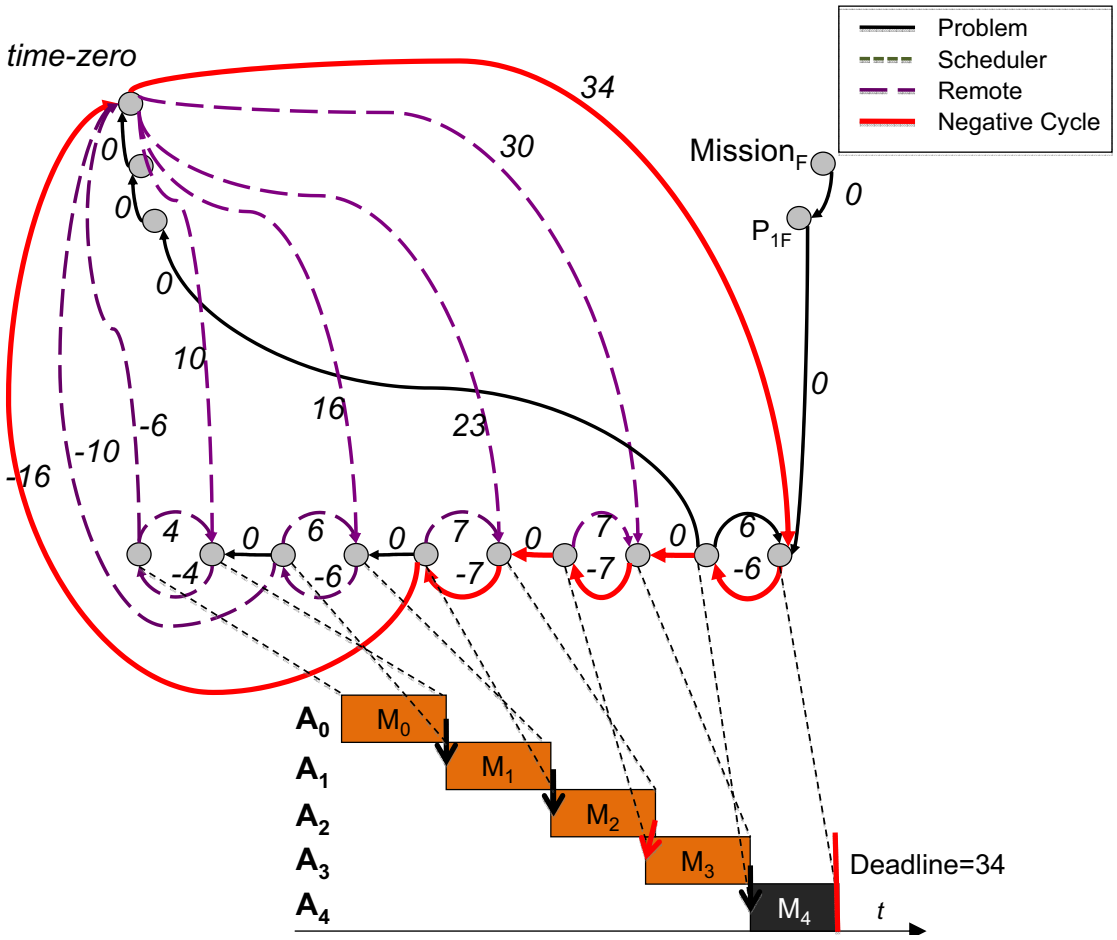

Figure 7.21: The Second Negative Cycle in the STN of Agent A_4

"remote" that have been *requested* to *shift* and the *shift* amount. A *recipient* agent accomplishes the *shift* in two steps:

1. For the "remote" activities that are *shifting*, the agent *retracts* their *release* or *deadline* constraint to allow them to slide freely. Retracting the *release* constraint of a "remote" method allows it to slide *backward*, while retracting the *deadline* constraint enables it to *shift forward*. The removal of these constraints prevents the appearance of "superfluous" inconsistencies involving the "remote" activities in the *shifting set* when attempting to *shift* the "local" activities.

   Continuing the example of Figure 7.6, Figure 7.23 illustrates the STN of agent A_3 after it has withdrawn the *release* constraints of the "remote" *shifting* methods M_0, M_1 and M_2 (the retracted constraints are shown in light gray). With

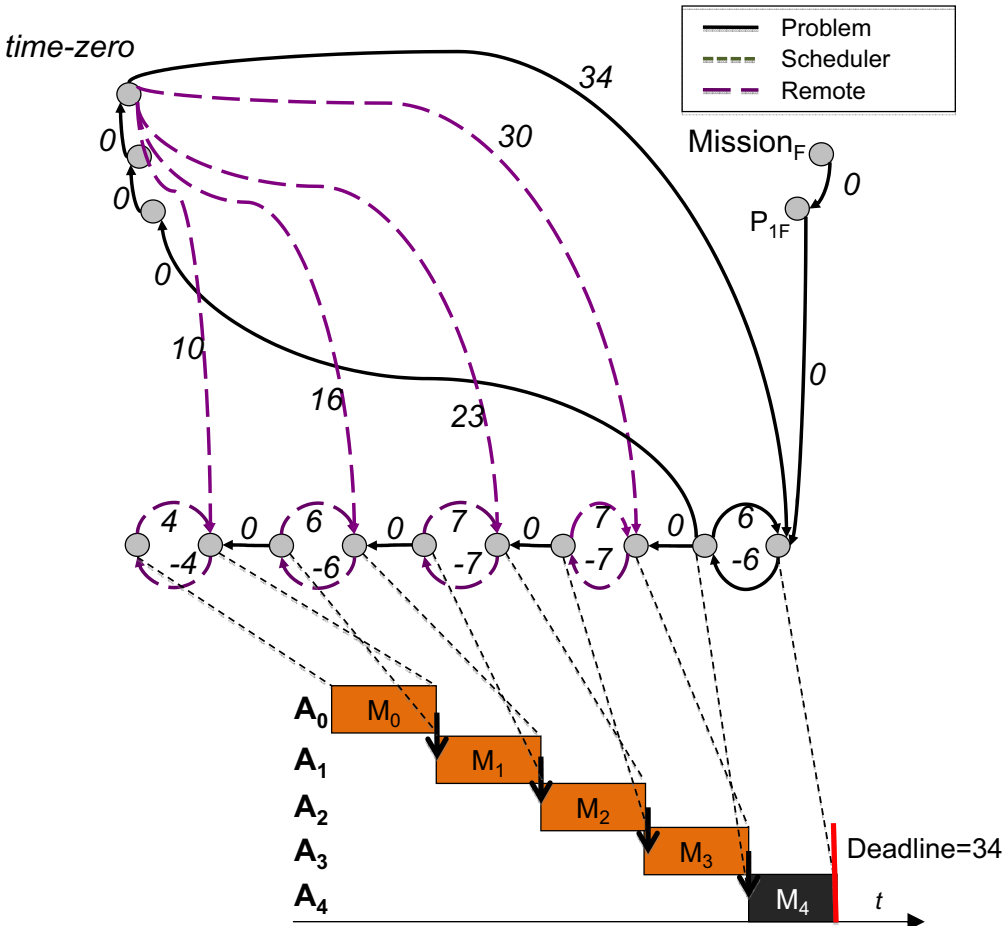

Figure 7.22: The Conflict-Free STN of Agent A_4

these constraints removed, these three methods no longer prevent the local *shifting* method M_3 from sliding *backward*.

2. As in the *recursive* strategy, the agent imposes temporary *release* or *deadline* constraints for the "local" activities that need to *shift*. These constraints uphold the necessary *shift* in start or finish time of the activity.

   Figure 7.24 illustrates how agent A_3 uses a temporary *deadline* constraint to *shift* the LFT of the "local" method M_3. Since all the "remote" methods in the *shifting set* had their *release* constraints retracted, A_3 can insert this constraint into its STN without producing an inconsistency.

   A *negative cycle* indicates that the "local" schedule has a conflicting activity that prevents the *shift*. An example of this situation can be illustrated using

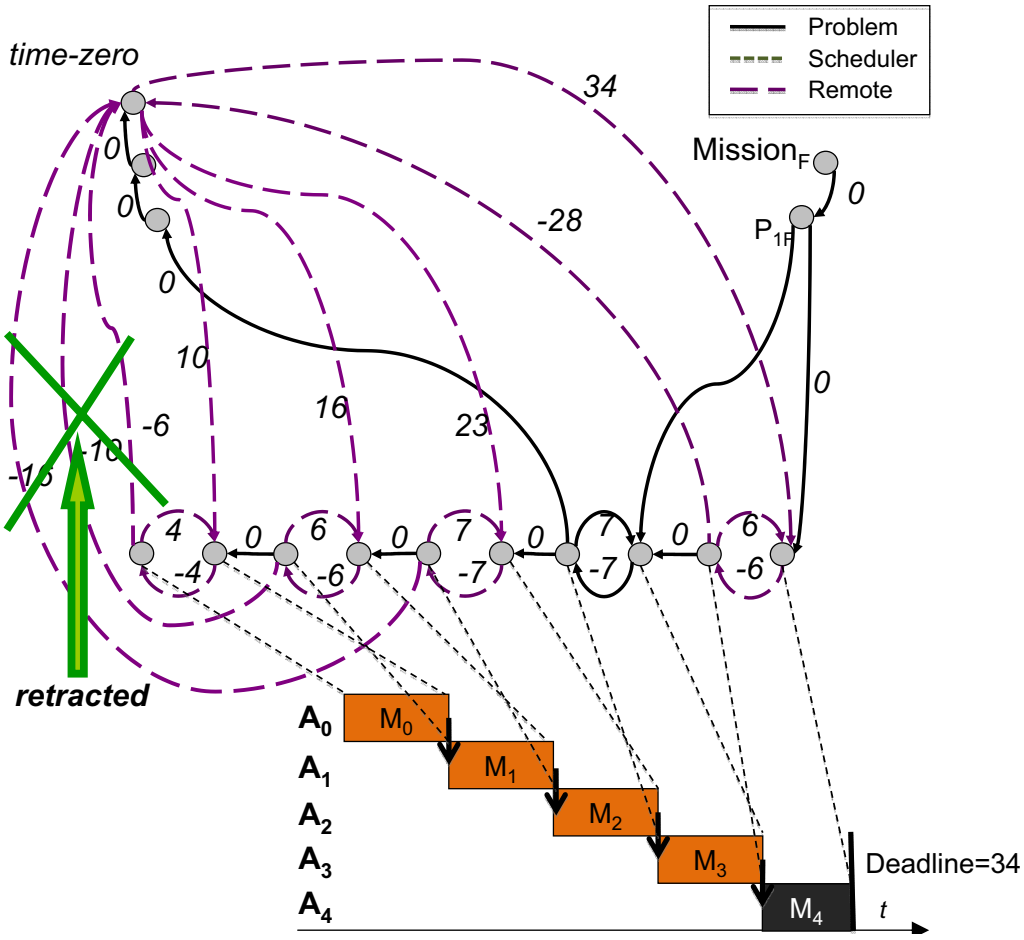

Figure 7.23: Agent A_3 Retracts the Release Constraints of "Remote" Methods

agent A_0. This agent "owns" the first method in the activity chain M_0, and is the "originator" of the inconsistency, given that the late finish of M_0 pushes the whole activity *chain forward* preventing agent A_4 from scheduling M_4. When agent A_0 attempts to slide method M_0 by imposing a *deadline* constraint that fulfills the *requested shift*, its STN flags an inconsistency. The *negative cycle* is shown in Figure 7.25. The conflict can be resolved by unscheduling the conflicting "local" method A. The temporary *deadline* on method M_0 can now be introduced without further conflict. The resulting conflict-free STN is shown in Figure 7.26.

Once the "local" activities have been *shifted* as *requested*, the resulting schedule is used to generate a *response* to the *request*. The *response* contains the *quality*

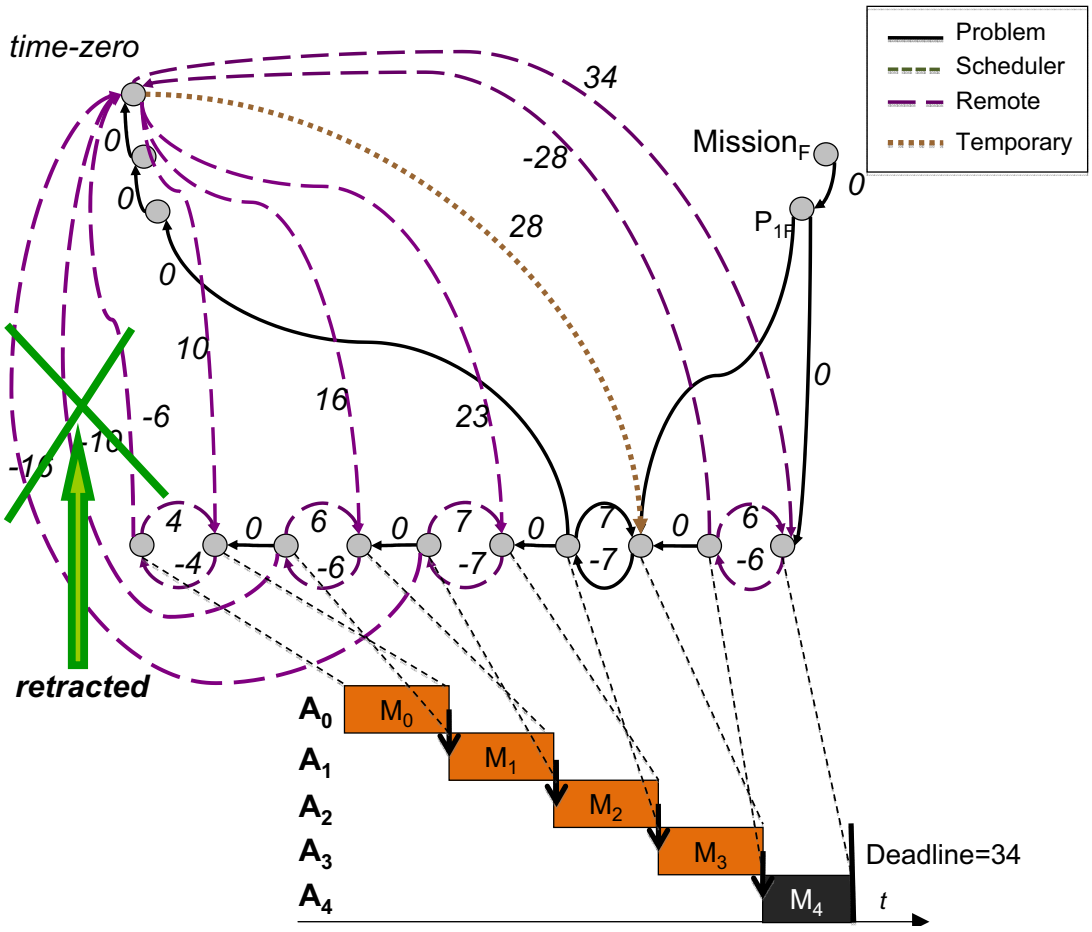

Figure 7.24: Agent A_3 Adds Temporary Release Constraint to M_3

cost incurred for making the schedule change (e.g. a drop in quality caused by the unscheduling of "local" activities to accomplish the *shift*). Similar to the calculation used in the *recursive* strategy, an agent L computes this effect by calculating the *quality cost*, $Q_{Lost}(L)$, to its schedule (the loss in quality incurred by changing the schedule to accommodate the *request*). As before, this *quality cost* is the quality *differential* between the current agent's schedule, and the new schedule that results after fulfilling the *request*:

$$Q_{Lost}(L) = Q_{After} - Q_{Before} \tag{7.6}$$

where Q_{Before} is the schedule quality before fulfilling the *request*, and Q_{After} the schedule quality after fulfilling the *request*. After this *quality cost* has been computed,

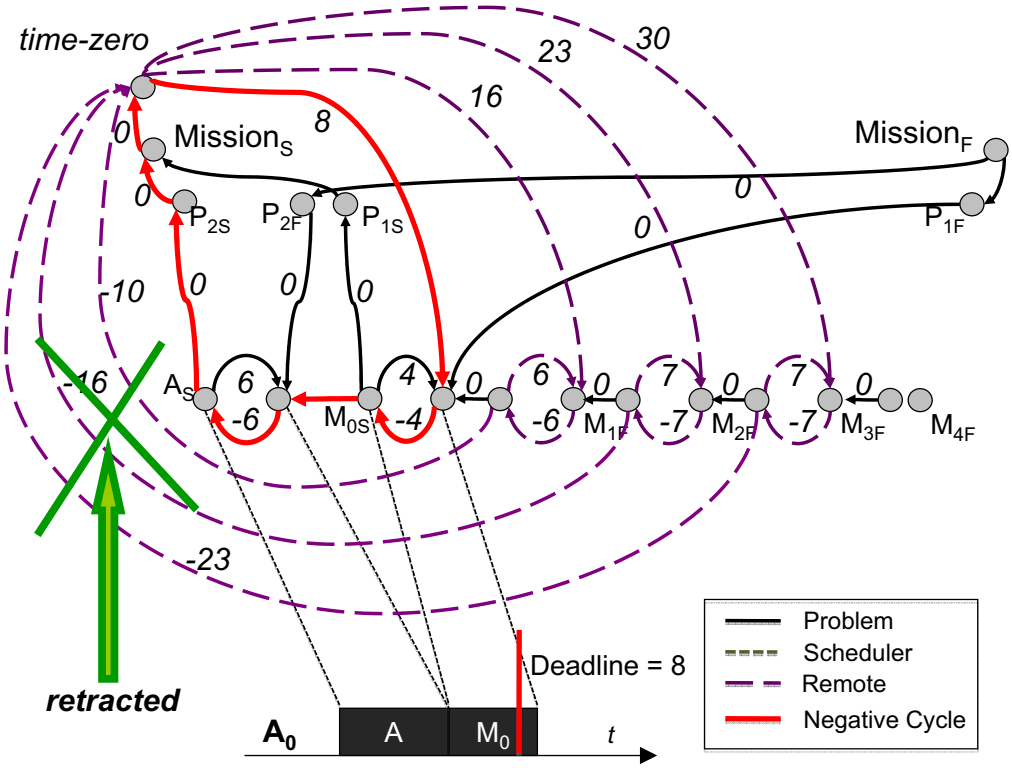

Figure 7.25: The Negative Cycle in A_0's STN

the agent sends a *response* to the *requesting* agent containing this information. An agent that could not accommodate the *request* sends a null *response*, causing the coordination process to die out.

For our example of Figure 7.6, all *recipient* agents except A_0 can *shift* their methods at no cost. Agent A_0 can also *shift* its method M_0, but needs to unschedule method A with a quality loss of 5. The responses sent to agent A_4 are

$$Q_{Lost}(A_0) = 5$$
$$Q_{Lost}(A_1) = 0$$
$$Q_{Lost}(A_2) = 0$$
$$Q_{Lost}(A_3) = 0$$

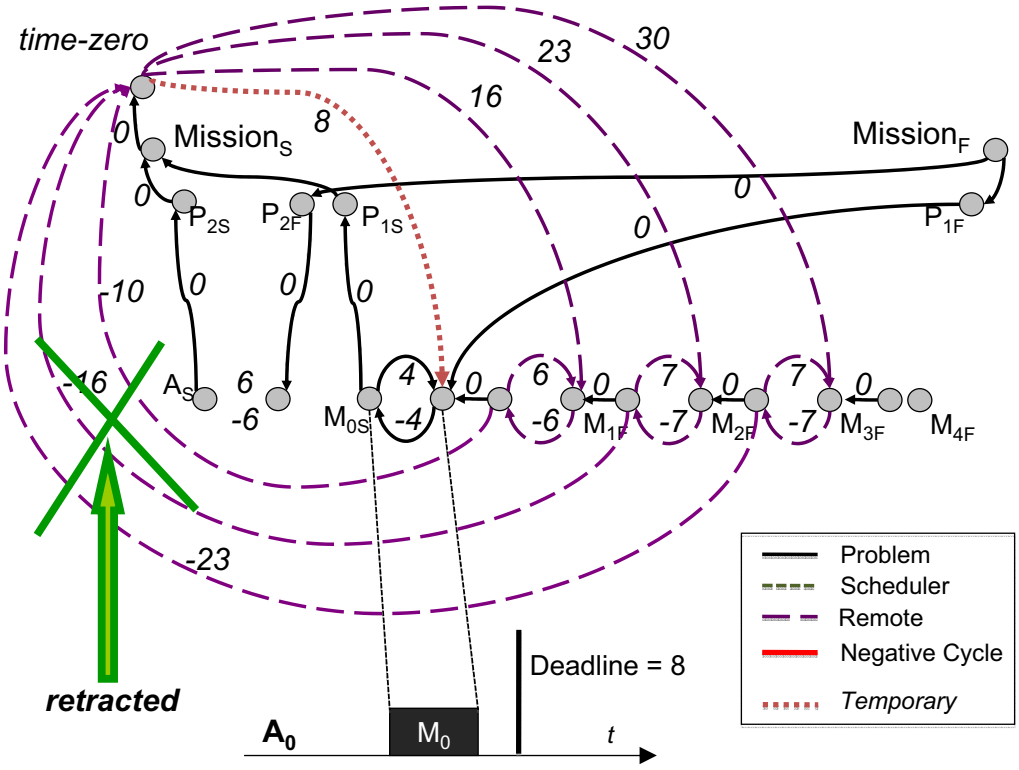

Figure 7.26: The Conflict-Free STN of Agent A_0

Committing to the Coordinated Schedule Change

The final step of the coordination strategy occurs once the *initiating* agent receives all the *responses* from the *recipient* agents. Similarly to the *recursive* strategy, the *initiating* agent uses the information in the *responses* to determines whether the schedule change would improve the collective multi-agent schedule. If so, it sends a *command* to all the *recipient* agents to commit to this change.

7.3 Experimental Design and Results

To test the performance characteristics of our coordination approaches, we conducted experiments that compared the performance of three agent teams: (1) A team of "baseline" agents that do not update commitments once they have been made, (2) a team of agents using the *recursive* coordination strategy to update commitments

and, (3) a team of agents using the *full-chain* coordination strategy to update commitments.

We used two problem sets in our experimental comparisons. The first problem set consists of sixteen subsets of randomly generated C_TAEMS scenarios. These scenarios were designed to highlight the advantages of updating commitments during execution when new scheduling opportunities arise. The second problem set consisted of scenarios provided by the Coordinators program for the Year 1 evaluation.

7.3.1 Testing Coordination Strategies

To compare how the teams using the *recursive* and *full-chain* coordination strategies performed against the "baseline" agents, we designed several sets of randomly generated C_TAEMS scenarios. These scenarios were constructed to highlight two features: (1) Updating inter-agent commitments during execution can increase the quality of the schedule by taking advantage of new scheduling opportunities, and (2) The two coordination strategies present a trade-off between the amount of information transferred and the time needed for coordination.

Figure 7.27 shows a partial view of a 15-agent C_TAEMS scenario generated for these experiments. The activity hierarchy is arranged left-to-right, rather than top-to-bottom for easier viewing. In this figure, higher-level tasks have a rounded-edge rectangular shape, while methods have a straight-edge rectangular shape. The text inside the boxes shows the name of the activity. Additionally, in the case of tasks, the box displays its QAF (such as SUM or SUMAND). The yellow arrows between tasks portray *enables* NLEs, while the maroon arrows represent *facilitates* NLEs. The key activities in this C_TAEMS scenario are shown with dark black borders. These activities *facilitate* an activity *chain*, and are *extrinsically* valuable (see Section 7.2) [7]. The initial schedule does not honor this *facilitates* NLE, but the quality of the schedule can potentially be improved if it is respected. The agents need to coordinate a *shift* of the *facilitated* activity *chain* to take advantage of this opportunity for schedule improvement.

The generated scenarios have a simple five-level activity hierarchy (loosely based on the activity hierarchies used by the Coordinators program).

[7] The activities are also *intrinsically* valuable, but we emphasize their *extrinsic* nature due to the added coordination opportunity.

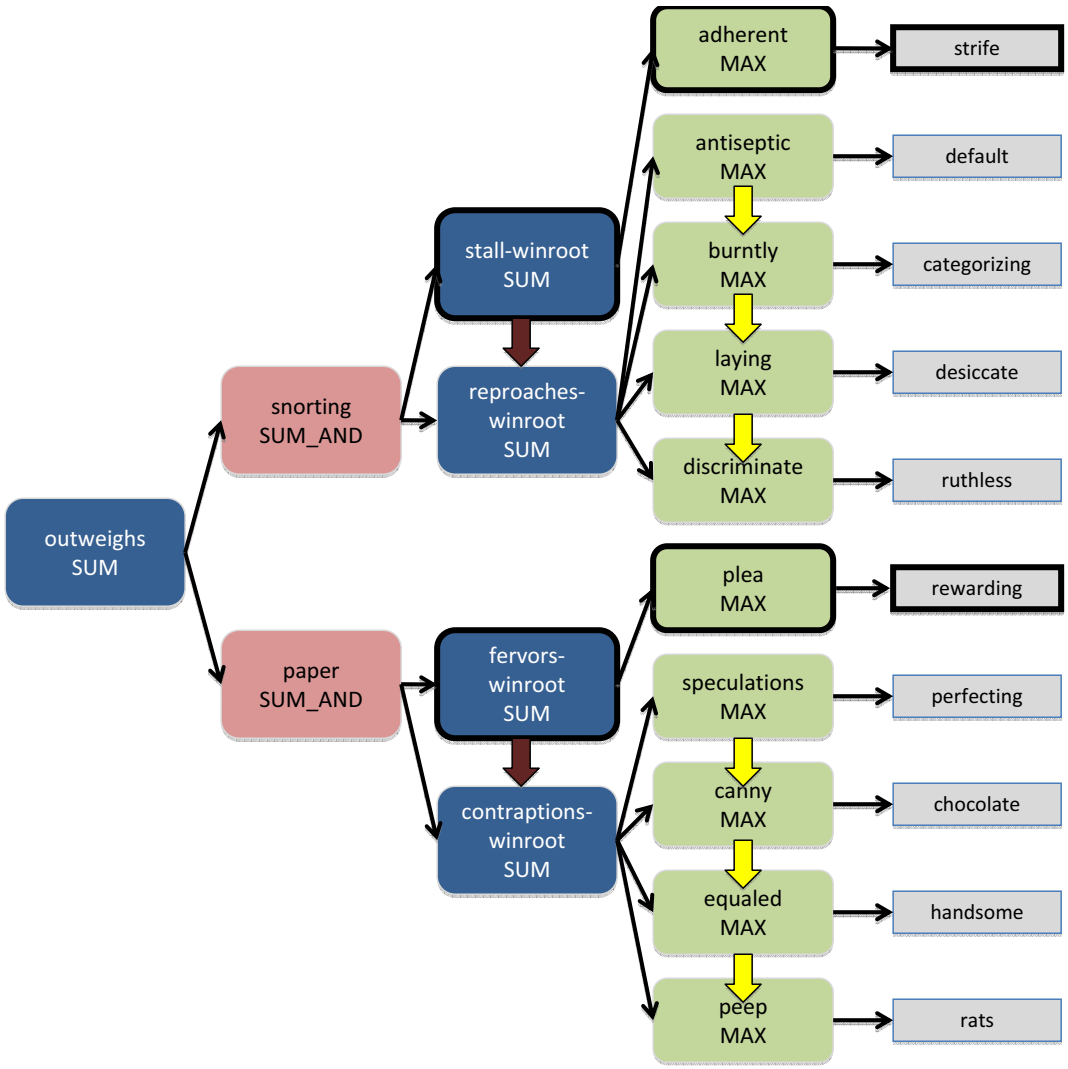

Figure 7.27: Example C_TAEMS Scenario for Testing Conflict-Driven Coordination Strategies (Note: Activity names are meaningless)

1. First Level: The "scenario" (the taskgroup) is a single task with a SUM QAF.
2. Second Level: The "problems" have a SUMAND QAF and each has two children activities.
3. Third Level: The "windows", define a release and deadline for their descendant activities. They have a SUM QAF.
4. Fourth Level: These "cnode" tasks are the direct children of the "window" tasks. They consist of a set of tasks with a MAX QAF. When "windows" have

multiple "cnode" children, these "cnodes" are *chained* through *enables* NLEs. "Cnode" tasks have a single child method.

5. Fifth Level: This is the last level of activities, consisting of executable methods. Each method was given three-point quality and duration distributions. The three points of the quality distribution consist of an expected value for the quality, a second value 10% lower, and a third value 10% higher. The three points of the duration distribution consist of an expected value for the duration, a second value 15% lower, and a third value 15% higher. The methods were given an expected quality of 10, and an expected duration of 10 time units.

The *facilitates* NLEs boost the quality of the *target* activities (by a factor of 2), but do not change their durations.

For each scenario, we compared the performance of three strategies:

1. Baseline (B): This "baseline" team of agents is given an initial schedule (generated by a centralized solver), that provides the agents with the set of inter-agent temporal commitments that they have to abide while executing the schedule. New commitments can be formed using the execution-time *implicit* and *explicit* mechanisms described in section 7.1. Once established, commitments cannot be broken.

2. Recursive (R): In addition to the capabilities provided to the "baseline" agents, this team of agents can update inter-agent temporal commitments using the *recursive* strategy described in section 7.2.1.

3. Full-Chain (F): In addition to the capabilities provided to the "baseline" agents, this team of agents can update inter-agent temporal commitments using the *full-chain* strategy described in section 7.2.2.

We measured (1) the performance of the two agent teams using the *recursive* and *full-chain* strategies against that of the "baseline" team in terms of the quality of the executed schedule, and (2) the trade-off between the amount of information needed to coordinate and the time needed for coordination when using the *recursive* and *full-chain* coordination strategies.

We generated 16 sets of scenarios (each set containing 30 C_TAEMS plans) that differed in two key parameters: The *chain length* and *uncertainty*. The *chain length*

represents the number of chained "cnodes" that need to be *shifted* by the coordinating agents. The uncertainty parameter indicates the probability that the maximum duration of a method will occur. This last parameter is described in greater detail in Section 4.2.1. The number of agents in the scenarios varied in direct proportion to the *length* of the "cnode" chains: Scenarios with *chains* of *length* 2 had 9 agents, scenarios with *chains* of *length* 3 had 12 agents, scenarios with *chains* of length 4 had 15 agents, and scenarios with *chains* of *length* 5 had 18 agents.

Table 7.1 shows the quality results of applying the "baseline" strategy (B), and the two coordination strategies (R and F) on the generated scenarios. For each scenario, we computed the ratio Q_s^r using Equation 4.6, where Q_s^r is the quality ratio achieved by strategy s (B, R or F). For each of the sixteen different settings of *chain length* and *uncertainty* the average ratios Q_s^r were computed. The three values are shown side-by-side, while the p-values (obtained using the Student's t-test) comparing the "baseline" strategy, B, against both coordination strategies, R and F, are shown below. As expected, the results strongly validate the benefits of both coordination strategies: When compared to the "baseline" strategy, both the *recursive* and the *full-chain* strategy show significant increases in performance on the majority of scenarios, regardless of *chain length* or *uncertainty*. Nonetheless, two sets of scenarios displayed a contrarian trend, with the "baseline" team outperforming the coordinating agents. These scenarios had common characteristics: *Chain length* was small (2), and uncertainty was high (≥ 0.25). A close examination of the schedule execution traces for these two sets of scenarios shows that when the activity chains are short, unexpectedly long method durations (caused by the higher uncertainty) can cause the loss of the whole activity chain, resulting in lower quality than the original "uncoordinated" schedule (used by the "baseline" strategy). A second interesting observation is that both coordination techniques have nearly identical performances. This result indicates that the *recursive* strategy, despite the more limited view of the agents, can match the performance of the *full-chain* strategy while maintaining greater privacy of information.

Table 7.2 shows a comparison of the number of bytes transmitted by both coordination strategies (R or F) in messages sent by the agents to keep keep each other informed about the state of their activities (i.e. messages containing updated information about "local" activities sent to "remote" agents that are aware of them). The results are presented as the ratio of the number of bytes transmitted, B_s^r, given by

the equation

$$B_s^r = B_s / B_{max} \tag{7.7}$$

where B_s is the number of bytes transmitted by strategy s (R or F), and B_{max} is the maximum number of bytes transmitted by either of the two strategies for the given scenario. The average ratios B_s^r for each of the sixteen sets of scenarios are shown. As predicted, the *recursive* strategy R shows a significant reduction in the number of bytes transmitted by the agents. On average, the number of bytes transmitted by the agents using the *recursive* strategy was less than 50% of the bytes transmitted by agents using the *full-chain* strategy.

Finally, table 7.3 compares the amount of time the agents spent coordinating updates to temporal commitments, given the coordination strategy that they were using (R or F). The amount of time spent in coordination was computed by adding the number of milliseconds spent in coordination sessions for each scenario. The results are presented as the time ratio T_s^r, given by the equation

$$T_s^r = T_s / T_{max} \tag{7.8}$$

where T_s is the number of milliseconds spent in coordination by strategy s (R or F), and T_{max} is the maximum number of milliseconds spent in coordination by either of the two strategies for the given scenario. The average ratios T_s^r for each of the sixteen sets of scenarios are shown. As hypothesized, the *full-chain* strategy F is faster than the *recursive* strategy R. On average, the *full-chain* strategy spent at least 25% less time in coordination than the *recursive* strategy.

To put the byte and time ratios in perspective, Table 7.4 and and Table 7.5 present the average number of bytes transmitted per scenario using strategies R and F, and the average number of milliseconds spent in coordination per scenario using strategies R and F respectively, across the full set of generated problems. These "raw" numbers show that while the transmission cost savings obtained by the *recursive* strategy are in the 100Kb range, the time advantage achieved by the *full-chain* strategy can be counted in fractions of a millisecond. In other words, while the *recursive* strategy achieved significant savings in communication costs, the computational advantage of the *full-chain* strategy was negligible. Coupled with the previous results showing that both strategies produce equivalent quality values lead us to conclude that the *recursive* strategy is clearly a better choice in the type of scenarios used in this thesis. However,

the *full-chain* strategy could still provide advantages if the problem domain changes. While the low computation times used by our coordination techniques are a function of their low-overhead implementation, a more computationally intensive coordination strategy (see future work in section 8.2) may increase the time savings obtained by the *full-chain* strategy, placing it in a better light. Further, if humans become involved in the decision process, the ability to do a single-shot coordination session rather than a cascading set of *recursive* coordinations could provide real advantages.

7.3.2 Coordinators Scenarios

To further validate the advantages of our coordination strategies, we tested the performance of the three agent teams ("baseline" team against the two teams using the coordination strategies), on two sets of 8-agent scenarios used by the Coordinators program in the Year 1 evaluation, each containing 90 scenarios. The scenarios contained dynamics in the outcome of activities: Methods can accrue different quality than expected, or their durations may be different than expected. These two sets (INT-OC and CHAINS-OC) had one key difference: The INT-OC set had 3 times as many *facilitates* NLEs. Some of these *facilitates* NLEs are not respected by the initial schedule, creating opportunities for coordinated schedule improvement during execution. The original problems were modified slightly to make the *facilitates* NLEs more valuable. Similarly to the generated problems, *facilitates* NLEs were changed to boost the quality of the *target* activities (by a factor of 2), but not change their durations. A new initial schedule (that took into consideration these modified *facilitates* constraints) was formulated for each scenario.

Table 7.6 shows the quality results comparing the baseline against the R and F strategies. On both sets of problems, the teams using the coordination strategies obtain improved performance [8]. As expected, the performance differential is greater in the INT-OC set, given the larger number of coordination opportunities the agents encounter.

[8]The difference between the "baseline" and the "recursive" strategies in the CHAINS-OC set is not significant.

7.4 Summary

This chapter examines the use of conflict explanation strategies (see section 2.5.2) to coordinate updates to the multi-agent schedule. When an agent attempts to install a "valuable" activity and *fails*, current inter-agent temporal commitments (e.g. a precedence relation between a "local" activity and a "remote" activity) are oftentimes the cause of the failure. To schedule the activity, the agent needs to coordinate an update to the conflicting commitment.

We present two mechanisms that provide an agent that tries to install a valuable activity and *fails*, with the ability to modify the necessary conflicting commitments: The *recursive* strategy and the *full-chain* strategy. Both of these strategies give an agent the ability to *request* remote agents to *shift* their activities *forward* or *backward* in time. To locate the remote activities that need to *shift*, the *negative cycles* returned by the STN when the valuable activity *failed* to schedule are analyzed and *explained*.

The *recursive* strategy requires a more limited *view* of the plan than the *full-chain* strategy, but is more computationally intensive. The *recursive* strategy only assumes that, if an agent "owns" an activity in a *chain* (a set of activities linked by precedence relations), it will be aware of any remote activities that are *directly* linked to a local activity. When an agent locates a remote activity that needs to move so that a valuable local activity can be installed, it makes a *request* to the agent that "owns" this remote activity to *shift* it. However, this second agent may encounter a new conflicting inter-agent commitment with a third agent that prevents it from accommodating the *shift*. This new conflict can be overcome by making a new *recursive request* to this third agent to *shift* the necessary activity.

The *full-chain* strategy, on the other hand, requires a larger view of the plan, but uses less time for coordination. The *full-chain* strategy assumes that, if the agent "owns" an activity in a *chain*, then it is aware of all the activities (both local and remote) that form part of the *chain*. With this larger *view*, the local agent can locate every remote activity in the *chain* that needs to *shift* to make space for a valuable activity that failed to install. The agent can then make a single *request* to all the agents involved, and these agents can immediately *respond* without further need of coordination with other agents.

As expected, our results show that agents capable of updating commitments (with either of the two strategies described above) can significantly outperform agents that

maintain all commitments once they are made. Significantly, we found that agents using the *recursive* strategy achieved comparable performance to the agents using the *full-chain* strategy, despite their more limited view. To compare the performance characteristics of the two coordination strategies, we measured the number of bytes they transmitted, and the time they needed for coordination. We found that the *recursive* strategy reduces the bytes transmitted by over 50%, while the *full-chain* strategy cuts the coordination time by over 25%. Nonetheless, given that the computation time for coordination could be measured in only fractions of a millisecond, it is unclear whether this lower computation overhead provides a significant advantage to the *full-chain* strategy.

		Chain Length											
		2			3			4			5		
		B	R	F	B	R	F	B	R	F	B	R	F
Uncertainty	0.05	0.8738	0.9530	0.9530	0.8519	1.0	1.0	0.8668	1.0	1.0	0.8324	0.9985	1.0
			$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$
	0.125	0.8601	0.9506	0.9506	0.8106	1.0	1.0	0.8299	1.0	1.0	0.7841	0.9932	0.9982
			$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$
	0.25	0.9217	0.7789	0.7789	0.8231	0.9998	0.9998	0.7825	0.9945	1.0	0.7700	0.9968	1.0
			$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$
	0.333	0.9242	0.8456	0.8456	0.8424	0.9893	0.9893	0.7675	0.9892	1.0	0.6968	0.9970	1.0
			$p = 0.0451$	$p = 0.0451$		$p = 0$	$p = 0$		$p = 0$	$p = 0$		$p = 0$	$p = 0$

Table 7.1: Quality Ratios Obtained Comparing the "Baseline" vs. the Coordination Strategies

		Chain Length							
		2		3		4		5	
		R	F	R	F	R	F	R	F
Uncertainty	0.05	0.4114	1.0	0.4318	1.0	0.4326	1.0	0.4155	1.0
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	
	0.125	0.4152	1.0	0.4260	1.0	0.4300	1.0	0.4155	1.0
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	
	0.25	0.4142	1.0	0.4252	1.0	0.4167	1.0	0.4048	1.0
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	
	0.333	0.4148	1.0	0.4345	1.0	0.4155	1.0	0.3987	1.0
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	

Table 7.2: Ratios Comparing the Number of Bytes Transmitted Using the Coordination Strategies

		Chain Length							
		2		3		4		5	
		R	F	R	F	R	F	R	F
Uncertainty	0.05	1.0	0.6700	0.9990	0.7059	1.0	0.6452	0.9874	0.7347
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	
	0.125	1.0	0.6543	0.9913	0.6759	1.0	0.6635	0.9795	0.6937
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	
	0.25	1.0	0.6531	0.9972	0.6750	0.9837	0.6729	0.9668	0.6887
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	
	0.333	0.9774	0.6873	0.9925	0.6993	0.9979	0.6588	0.9936	0.7583
		$p = 0$		$p = 0$		$p = 0$		$p = 0$	

Table 7.3: Ratios Comparing the Amount of Time Elapsed During the Coordination Strategies

R	F
1485694	3571105

Table 7.4: Average Number of Bytes Transmitted Using the Coordination Strategies

R	F
0.337	0.233

Table 7.5: Average Number of Milliseconds Elapsed During the Coordination Strategies

Chain Length	B	R	F
INT-OC	0.9584	0.9955	0.9990
		$p = 0$	$p = 0$
CHAINS-OC	0.9884	0.9952	0.9974
		$p = 0.0838$	$p = 0.0043$

Table 7.6: Quality Ratios Obtained Comparing the "Baseline" vs. the Coordination Strategies (Coordinators Problems)

Chapter 8

Conclusions and Future Work

In this thesis, we have focused on the management of a multi-agent schedule in the face of execution uncertainties. STNs provide key advantages to schedulers operating in a highly dynamic domain, an environment commonly faced by scheduling agents. STNs provide strong support for the construction of *flexible-times* schedules, which can absorb many deviations from the planned solution without need for rescheduling. STNs also lend assistance to *incremental* scheduling techniques, which revise the existing schedule rather than formulating a new one from scratch. Both of these features provide key aids in maintaining the coherency of a multi-agent schedule in the face of execution uncertainty. Despite these advantages, STNs have until now received scant attention in distributed scheduling domains. A key limitation preventing wider acceptance is that STNs require full consistency between all temporal constraints in the problem. An STN with an inconsistency provides *no solution*. Given that inconsistencies during schedule execution are the norm in uncertain environments (e.g. an activity finishes late and it can no longer meet its deadline), this property poses a severe shortcoming. This thesis work presents a set of strategies that leverage the *conflict* detection tools presented by an STN framework to provide a set of STN-based scheduling agents with the ability to *explain* and *resolve* inconsistencies as they arise. We then investigate how similar *conflict explanation* techniques can be applied to conflicts encountered during the scheduling search to improve the multi-agent schedule.

8.1 Contributions

Our work has investigated the use of STNs to address the problem of online management of the distributed schedules of a team of agents operating in a dynamic, uncertain environment. This thesis makes the following contributions:

- A framework to recover from inconsistency during the execution of the multi-agent schedule: Our framework for resolving inconsistencies caused by execution-time deviations from the planned schedule (e.g a late finish of an activity) expands on *conflict explanation* techniques developed for centralized schedulers. Our strategy presents a two-step approach that resolves these inconsistencies as they arise: First, the *negative cycle* that results from the temporal conflict is analyzed and *explained*, and then, based of this *explanation*, an action is taken to resolve the conflict. We categorize the actions an agent can take to resolve a conflict into two sets:

  1. "Local" actions: These actions make changes to the "local" schedule without need for coordination with other agents. They are similar to those taken by a centralized solver, which has the full ability to make scheduling decisions on any scheduled activity.
  2. "Non-local" actions: These actions modify constraints that affect other agents. They work by *suspending* a temporal constraint that enforces "remote" constraints (e.g. a precedence constraint between "local" activity and a "remote" activity), restoring STN consistency. The constraint is *restored* once the agent receives updated information that removes the cause of the inconsistency.

  These conflict resolution strategies provide an STN-based scheduling agent with continued use of its STN in the face of temporal inconsistencies that inevitably arise during schedule execution in uncertain domains. Our results show that an agent team that uses these strategies has significantly improved performance over a "conservative" agent team that instead uses a "fail-safe" schedule designed to avoid any temporal inconsistencies from occurring during execution by scheduling activities to their maximum duration.

Similar *conflict explanation* strategies can be leveraged to aid the scheduling search by examining scheduling-time conflicts. These analyses can be used to take actions

that *enhance* the multi-agent schedule. Our work has investigated two areas of schedule improvement: increasing the *robustness* of the multi-agent schedule, and coordinating updates to inter-agent temporal commitments (e.g. precedence relations between "local" and "remote" activities) that increase the schedule's quality. This thesis makes the following contributions:

- A strategy to leverage inconsistencies to detect impending failures caused by durational uncertainty: Our work on *schedule robustness* presented a strategy that increases the *fault tolerance* of a multi-agent schedule by *backfilling* redundant activities in a *Just-in-Time (JIT)* fashion. The *backfilling* process focuses on a set of *close-to-execution* of activities, rather than all *pending* activities. Limiting the focus of the backfilling process helps to maintain tractability, while providing aid to the most vulnerable sections of the schedule: those activities about to be executed. The *close-to-execution* activities are vetted for potential *failures* by examining how they would affect the multi-agent schedule if their durations deviate from the expected. We conduct this assessment by introducing their potential unexpected durations (based on a model of durational uncertainty) into the STN. Problem durations that result in an inconsistency point to a potential impending *failure*. By analyzing these *hypothetical* inconsistencies, we can determine what activities in the schedule are in danger of *failing*, and introduce *redundancies* for these activities into the schedule to protect against this potential failure. Our results show that an agent team using this JIT *backfilling* strategy can significantly outperform a "reactive" agent team that waits for execution inconsistencies to arise, and then resolves them.

- Strategies to exploit inconsistencies to drive inter-agent coordination: Our work on "updating" inter-agent temporal commitments has developed two *conflict-driven* coordination mechanisms to schedule new activities that failed to be added to the schedule because of current commitments. The two mechanisms present a trade-off between the amount of information an agent needs (i.e., the extent to which the agents must share a global view) versus the amount of time needed for coordination. Using either mechanism, the agents analyze the conflicts identified by their STNs when an activity *fails* to install, and locate the set of "remote" activities that need to "move" to make space for the activity. The "move" can then be accomplished through negotiation. Our results show that both these mechanisms can improve the quality of the multi-agent

schedule, when compared with a team of agents that continues to enforce all temporal commitments, once made. The less information-intensive, but slower mechanism can be used when information privacy is important. The faster, but more information-intensive mechanism can be used if the coordination process is computationally intensive, and coordination time needs to be minimized.

8.2 Limitations and Future Extensions

As is the case with most theses, we have taken only initial exploratory steps towards demonstrating the advantages that STN-based scheduling technology can bring to distributed domains. Future research can build on the strategies we have presented to further the integration of STNs in multi-agent settings. In this section, we start by proposing specific additions that could enhance the strategies developed in this thesis, and then close by outlining broader open questions.

- **Increasing the flexibility of the strategies for recovering from inconsistency**: A valid criticism of our conflict recovery strategy is that the algorithm can be excessively rigid. Conflict categories are prioritized and an inconsistency is matched to the first category that applies. Sometimes, this process results in an inappropriate match, which can lead to unnecessary or inefficient scheduling actions while attempting to resolve the conflict. A more flexible approach could potentially lead to better results. This new strategy could provide a deeper analysis of the inconsistencies that arise and develop matching algorithms that discover the best fitting conflict resolution action to apply. Such a strategy could also *learn* from previous "successful" conflict resolution actions, so as to apply these actions when presented with "similar" inconsistencies.

- **Integrating soft constraint reasoning into the JIT Backfilling Framework**: Our JIT Backfilling framework assumes that soft constraints in the problem do not affect the duration distributions of plan activities (see Section 6.1). Such a restriction is reasonable for an algorithm that relies on the accuracy uncertainty model to determine the likelihood of potential failures. However, an expanded strategy that takes soft constraints into account would be advantageous. Such a strategy would likely require much closer coordination between the agents to keep each other informed about all the potential times when an

activity accrues initial and added quality (e.g. in a C_TAEMS model, a SUM task with two children obtains initial quality when the first child finishes and additional quality when the second finishes), as well as the potential uncertainty in these times. Using this additional information, an expanded strategy could reason about when and how soft constraints are met. On the downside, the computational cost of such a strategy would likely increase significantly, and appropriate compromises between this computational cost and accuracy of the *failure* analysis would have to be found.

- **Reasoning about quality uncertainty**: While our JIT Backfilling work has concentrated on durational uncertainty, oversubscribed scheduling problems typically involve the optimization of a reward metric, such as quality in the case of C_TAEMS. C_TAEMS activities have uncertain outcomes both in terms of duration and quality, and an uncertainty model for both quantities is provided. Quality uncertainty can cause activities to become less valuable than expected, or outright fail. Hedging against this uncertainty could strengthen the schedule against this type of failures. An integrated framework that incorporates the JIT Backfilling strategy we have presented, along with new reasoning about quality uncertainty would be a valuable contribution.

- **Development of a integrated coordination framework**: The work presented in this thesis explored how to schedule an activity by updating the multi-agent schedule through the *shifting* or removing of currently scheduled activities. However, some scheduling actions may require a more extensive schedule change where several activities are concurrently added, removed and/or *shifted*. A combined coordination framework that can accomplish such larger schedule changes could lead to higher performance of the agent team.

- **Adding Located Activities**: A feature of interest in most practical domains is located activities. When activities have a location where they need to occur, agents have to travel to this location to perform the activity. This travel introduces a new variable in the problem: Travel time is not "useful" in the sense that agents do not accrue quality while traveling, and thus there is an incentive to minimize it. The strategies we have developed in this work are not location-aware. Enhancing these techniques with location reasoning would increase their value. For example, does it still make sense to introduce a "redundant" activity in the schedule if there is a large travel time for the agent? A

location-aware strategy would need to reason about these compromises to make the right choices.

8.2.1 Open Issues

- **How good are the schedules?** The scheduling technology presented in this thesis relies on heuristic guidance to cope with the complexity of C_TAEMS plan structures. The different strategies we present are compared to one another to evaluate their comparative advantages, yet these results do not provide any indication as to how good these schedules are with respect to the optimal value. While obtaining the optimal schedule is generally not feasible, given the NEXP complexity of the multi-agent C_TAEMS scheduling problem (see Section 2.2.1), it may be possible to develop some metrics that provide some guidance as to how close to optimal a given solution is.

- **How can human input be integrated into the scheduling process?** The overarching goal of the Coordinators project is to provide technology that a human team can use to coordinate their activities. However, the interface between the human and the software has not yet received much attention. Humans are unlikely to fully embrace a technology that tells them what they need to do, without the ability to override the software's instructions. Designing technology that accepts inputs from the human and integrates this input in the distributed scheduling process is essential, but remains a difficult problem.

Bibliography

[1] R. Al-Omari, A. Somani, and G. Manimaran. Efficient overloading techniques for primary-backup scheduling in real-time systems. *Journal of Parallel and Distributed Computing*, 64(5):629648, 2004.

[2] M. Aloulou and M. Portmann. An efficient proactive reactive scheduling approach to hedge against shop floor disturbances. In *Proceedings of the MISTA*, 2003.

[3] R. Arkin. Cooperation without communication: Multiagent schema-based robot navigation. *Journal of Robotics Systems*, 9(3):351–364, 1992.

[4] R. Bartak and O. Cepek. Incremental propagation rules for precedence graph wit optional activities and time windows. In C. Beck, A. Davenport, and T. Walsh, editors, *Proceedings of the ICAPS Workshop on Constraint Programming for Planning and Scheduling, ICAPS-05*, pages 5–11, 2005.

[5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] D. Bernstein, S. Zimmerman, N. Immerman, and S. Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of Operations Research*, 27(4), 2000.

[7] C. Bessiere. Arc consistency in dynamic constraint satisfaction problems. In *National Conference on Artificial Intelligence – AAAI'91*, 1991.

[8] M. Boddy, B. Horling, J. Phelps, R. Goldman, R. Vincent, A. Long, and B. Kohout. C_taems language specification v. 1.06, October 2005.

[9] C. Boutilier, T. Dean, and S. Hanks. Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of 3rd European Workshop on Planning, EWSP-95*, 1995.

[10] C. Boutillier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR*, 11:1–94, 1999.

[11] A. Brambilla and M. Lavagna. Multi agent conflict partition scheduling for coordinated space systems. In *Proceedings of the 5th Workshop on Planning and Scheduling for Space*, October 2006.

[12] A. Brambilla and M. Lavagna. A coordination mechanism to solve common resource contention in multi agent space systems. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space - iSAIRAS 2008*, pages 26–29, February 2008.

[13] M. Bratman. Shared cooperative activity. *The Philosophical Review*, 101(2):327–341, 1992.

[14] M. Brenner. Multiagent planning with partially ordered temporal plans, 2003.

[15] J. L. Bresina and P. H. Morris. Explanations and recommendations for temporal inconsistencies. In *Proceedings of the 5th International Workshop on Planning and Scheduling for Space, Space Telescope Science Institute*, October 2006.

[16] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[17] R. Brown and J. Jennings. A pusher/steerer model for strongly cooperative mobile robot manipulation. In *Proceedings of International Conference on Intelligent Robots and Systems, IROS-95*, 1995.

[18] P. Brucker. *Scheduling Algorithms, 2nd Edition*. Springer-Verlag, 1998.

[19] P. Buzing, A. ter Mors, and C. Witteveen. Multi-agent plan repair with dtps. In K. N. Brown, editor, *Proceedings of the 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group, (PlanSIG 2004)*, pages 3–14, December 2004.

[20] P. Buzing and C. Witteveen. Distributed (re)planning with preference information. In R. Verbrugge, N. Taatgen, and L. Schomaker, editors, *Proceedings of the 16th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2004)*, pages 155–162, 2004.

[21] A. Cesta, G. Cortellessa, F. Pecora, and R. Rasconi. Supporting Interaction in the RoboCare Intelligent Assistive Environment. In *Proccedings of AAAI Spring Symposium on Interaction Challenges for Intelligent Assistants*, 2007.

[22] A. Cesta and A. Oddi. Gaining efficiency and flexibility in the simple temporal problem. In L. Chittaro, S. Goodwin, H. Hamilton, and A.Montanari, editors, *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME-96)*, May 1996.

[23] A. Chades, B. Scherrer, and F. Charpillet. A heuristic approach for solving decentralized-pomdp:. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, 2002.

[24] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of International Conference on AI Planning Systems (AIPS)*, 2000.

[25] V. Cicirello and S. Smith. Wasp-based agents for distributed factory coordination. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):237–266, 2004.

[26] B. Clement and A. Barrett. Continual coordination through shared activities. In *Proceedings of the International Conference on Autonomous Agent and Muti-Agent Systems*, pages 57–67, 2003.

[27] P. Cohen and H. Levesque. Persistence, intention and commitment. *Intentions in Communication*, pages 33–69, 1990.

[28] A. Davenport, C. Gefflot, and J. Beck. Slack-based techniques for robust schedules. In *Proceedings of the 6th European Conference on Planning, ECP-01*, 2001.

[29] J. de Kleer. A comparison of atms and csp techniques. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 290–296, August 1989.

[30] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1–2):35–74, July 1995.

[31] R. Dearden, N. Meuleau, S. Ramakrishnan, D. Smith, and R. Washington. Incremental contingency planning. In *Proceedings of ICAPS Workshop on Planning under Uncertainty.*, 2003.

[32] R. Debruyne, G. Ferrand, N. Jussien, W. Lesaint, S. Ouis, and A. Tessier. Correctness of constraint retraction algorithms. In *Sixteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 03)*, 2003.

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

[34] K. Decker and V. Lesser. Generalizing the Partial Global Planning Algorithm. *International Journal on Intelligent Cooperative Information Systems*, 1(2):319–346, June 1992.

[35] K. Decker and V. R. Lesser. Quantitative Modeling of Complex Environments. *International Journal of Intelligent Systems in Accounting, Finance and Management. Special Issue on Mathematical and Computational Models and Characteristics of Agent Behaviour.*, 2:215–234, January 1993.

[36] K. Decker and J. Li. Coordinated hospital patient scheduling. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98)*, pages 104–111, 1998.

[37] J. Dey, J. Kurose, and D. Towsley. On-line scheduling policies for a class of iris (increasing reward with increasing service) real-time tasks. *IEEE Transaction on Computers*, 45(7):802–813, July 1996.

[38] M. Dias and A. Stentz. Opportunistic optimization for market-based multirobot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS-02*, 2002.

[39] M. B. Dias and A. Stentz. Traderbots: A market-based approach for resource, role, and task allocation in multirobot coordination. Technical Report CMU-RI -TR-03-19, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2003.

[40] A. Drogoull and J. Ferber. From tom thumb to the dockers: Some experiments with foraging robots. In *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior*, pages 451–459, 1992.

[41] M. Drummond, J. Bresina, and K. Swanson. Just-in-case scheduling. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1098–1104, 1994.

[42] D. Dubois, J. Lang, and H. Prade. A possibilistic assumption-based truth maintenace system with uncertain justifications, and its application to belief revision. *Truth maintenance systems*, pages 87–106, 1990.

[43] E. Durfee and V. Lesser. Partial Global Planning: A Coordination Framework for Distributed Hypothesis Formation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1167–1183, September 1991.

[44] R. Emery-Montemerlo, G. Gordon, J. Schneider, and S. Thrun. Approximate solutions for partially observable stochastic games. In *Proceedings of the 3rd AAMAS*, 2004.

[45] K. Erol, J. Hendler, and D. S. Nau. Semantics for hierarchical task-network planning. Technical Report UMIACS-TR-94-31, Computer Science Department, University of Maryland, 1994.

[46] A. Finzi, F. Ingrand, and N. Muscettola. Model-based executive control through reactive planning for autonomous rovers. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS-04*, October 2004.

[47] R. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6), 1962.

[48] J. Foss and N. Onder. A hill-climbing approach to planning with temporal uncertainty. In *Proc. 19th Intl. Florida AI Research (FLAIRS) Conference*, 2006.

[49] M. S. Fox and K. Sycara. Overview of cortes: A constraint based approach to production planning, scheduling and control. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, 1990.

[50] E. C. Freuder, M. Minca, and R. J. Wallace. Privacy/efciency tradeoffs in distributed meeting scheduling by constraint-based agents. In *IJCAI-2001 Workshop on Distributed Constraint Reasoning*, 2001.

[51] L. Friha, P. Berry, and B. Choueiry. DISA: A Distributed scheduler using abstractions. *Revue d'Intelligence Artificielle*, 11(1):27–42, 1997.

[52] A. Gallagher and S. Smith. Recovering from inconsistency in distributed simple temporal networks. In *The 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS-21)*, May 2008.

[53] A. Gallagher, T. Zimmerman, and S. Smith. Incremental scheduling to maximize quality in a dynamic environment. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006, to appear)*, 2006.

[54] H. Gao. Building robust schedules using temporal protectionan empirical study of constraint based scheduling under machine failure uncertainty. Technical Report Masters thesis, Department of Industrial Engineering, University of Toronto, 1995.

[55] L. Garrido and K. Sycara. Multi-agent meeting scheduling: preliminary results. In *1996 International Conference on Multi-Agent Systems (ICMAS '96)*, pages 95 – 102, 1996.

[56] A. Garvey and V. Lesser. Design-to-time Scheduling and Anytime Algorithms. *SIGART Bulletin*, 7(3), January 1996.

[57] B. Gerkey and M. Mataric. Sold!: Auction method for multi-robot control. *IEEE Transaction on Robotics and Automation (Special Issue on Multi-Robot Systems)*, 18(5), October 2002.

[58] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272284, 1997.

[59] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *In International Conference on Dependable Systems and Networks (DSN03)*, 2003.

[60] D. Golderg, V. Cicirello, M. Dias, R. Simmons, S. Smith, and A. Stentz. Market-based multi-robot planning in a distributed layered architecture. In *Proceedings of the 2nd International Workshop on Multi-Robot Systems*, pages 27–38, 2003.

[61] J. Goldsmith and M. Mundhenk. Complexity issues in markov decision processes. In *IEEE Conference on Computational Complexity*, pages 272–280, 1998.

[62] B. Grosz. Collaborative systems. In *Proceedings of the 11th National Conference of Artificial Intelligence, AAAI-94*, 1994.

[63] D. Gurnell. Adaptive coordination for multi agent planning. In *Proceedings of the Twenty Second Workshop of the UK Planning and Scheduling Special Interest Group*, 2003.

[64] K. Hashimito, T. Tsuchiya, and T. Kikuno. A new approach to realizing fault-tolerant multiprocessor scheduling by exploiting implicit redundancy. In *In Proc. of the 27th International Symposium on Fault-Tolerant Computing (FTCS 97)*, 1997.

[65] K. Hashimito, T. Tsuchiya, and T. Kikuno. Eective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. *IEICE Transactions on Information and Systems*, E85-D(3):525534, 2002.

[66] L. Hiatt and R. Simmons. Towards probabilistic plan management. In *Proceedings of the 3rd Workshop on Planning and Plan Execution for Real-World Systems (ICAPS-07)*, 2007.

[67] L. Hiatt, T. Zimmerman, S. Smith, and R. Simmons. Reasoning about executional uncertainty to strengthen schedules. In *Proceedings of the ICAPS-08, Workshop on A Reality Check for Planning and Scheduling Under Uncertainty*, 2008.

[68] B. Horling, V. Lesser, and R. Vincent. Multi-Agent System Simulation Framework. *16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation*, August 2000.

[69] B. Horling, V. Lesser, R. Vincent, A. Bazzan, and P. Xuan. Diagnosis as an integral part of multi-agent adaptability. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 211–219. IEEE Computer Society, January 2000.

[70] B. Horling, V. Lesser, R. Vincent, and T. Wagner. The Soft Real-Time Agent Control Architecture. In *Proceedings of the AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems*, July 2002.

[71] B. Horling, V. Lesser, R. Vincent, T. Wagner, A. Raja, S. Zhang, K. Decker, and A. Garvey. The taems white paper, January 1999.

[72] I. hsiang Shu, R. Effinger, and B. Williams. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling, ICAPS-05*, pages 252–261, 2005.

[73] M. Huhns and D. Bridgeland. Multiagent truth maintenance. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6), 1991.

[74] L. Hunsberger. Algorithms for a temporal decoupling problem in multi-agent planning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2002.

[75] L. Hunsberger. Group decision making and temporal reasoning. Technical Report TR-05-02, Ph.D. Thesis. Harvard Technical Report, 2002.

[76] L. Hunsberger. Distributing the control of a temporal network among multiple agents. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 899–906, New York, NY, USA, 2003. ACM.

[77] L. Hunsberger. A practical temporal constraint management system for real-time applications. In *Proceedings of European Conference on Artificial Intelligence (ECAI-2008)*, 2008.

[78] L. Hunsberger and B. Grosz. A combinatorial auction for collaborative planning. In *Proceedings of the 4th International Conference on Multi-Agent Systems, ICMAS-00*, pages 151–158, 2000.

[79] L. Hunsberger and B. Grosz. A combinatorial auction for collaborative planning. In *Proceedings of the 4th International Conference on Multi-Agent Systems, ICMAS-00*, 2000.

[80] N. Jennings, E. Mamdani, I. Laresgoiti, J. Perez, and J. Corera. Grate: A general framework for cooperative problem solving. *IEE-BCS Journal of Intelligent Systems Engineering*, 1(2):102–114, 1992.

[81] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[82] L. F. Jr. and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[83] H. Jung, M. Tambe, A. Barrett, and B. Clement. Enabling efficient conflict resolution in multiple spacecraft missions via dcsp. In *In Proceedings of the NASA workshop on planning and scheduling*, 2002.

[84] N. Jussien. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, 2001.

[85] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, September 2000.

[86] L. Khatib, P. Morris, R. A. Morris, and F. Rossi. Temporal constraint reasoning with preferences. In *Proceedings of IJCAI*, pages 322–327, 2001.

[87] P. Kim, B. Williams, and M. Abrahmson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of the IJCAI'01*, 2001.

[88] M. Koes, I. Nourbakhsh, and K. Sycara. Constraint optimization coordination architecture for search and rescue robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) 2006*, pages 3977–3982, May 2006.

[89] M. Koes, K. Sycara, and I. Nourbakhsh. A constraint optimization framework for fractured robot teams. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2006.

[90] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13:32–44, 1992.

[91] M. G. Lagoudakis, E. Markakis, D. Kempe, P. Keskinocak, and A. Kleywegt. Auction-based multi-robot routing. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.

[92] P. Lawrence. *Workflow handbook*. John Wiley, 1997.

[93] S. Lemai and F. Ingrand. Interleaving temporal planning and execution in robotics domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2004.

[94] V. Lesser and D. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3), 1983.

[95] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X. Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.

[96] V. Lesser, B. Horling, F. Klassner, A. Raja, T. Wagner, and S. Zhang. Big: An agent for resource-bounded information gathering and decision making. *Artificial Intelligence, Special Issue on Internet Information Agents*, 118(1–2):197–244, May 2000.

[97] I. Little and S. Thiebaux. Concurrent probabilistic planning in the graphplan framework. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*, 2006.

[98] J. Liu, K.-J. Lin, W.-K. Shih, A. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computation. *Computer*, 24(5):58–68, May 1991.

[99] J. S. Liu and K. Sycara. Multiagent coordination in tightly coupled task scheduling. In *1996 International Conference on Multi-Agent Systems*, 1996.

[100] P. Maes. The agent network architecture (ana). *SIGART Bulletin*, 2(4):115–120, 1991.

[101] R. T. Maheswaran and P. Szekely. Criticality metrics for distributed plan and schedule management. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

[102] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 310–317, Washington, DC, USA, 2004. IEEE Computer Society.

[103] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 438–445, Washington, DC, USA, 2004. IEEE Computer Society.

[104] G. Manimaran and C. S. R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):11371152, 1998.

[105] Mausam and D. Weld. Probabilistic temporal planning with uncertains durations. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, 2006.

[106] Mausam and D. Weld. Planning with durative actions in stochastic domains. *Journal of Artificial Intelligence Research (JAIR)*, 31:33–82, 2008.

[107] D. McAllester. Truth maintenance. In R. Smith and T. Mitchell, editors, *Proceedings of the 8th National Conference on Artificial Intelligence*, volume 2, pages 1109–1116, Menlo Park, California, 1990. AAAI Press.

[108] S. Mehta and R. Uzsoy. Predictive scheduling of a single machine subject to breakdowns. *International Journal of Computer Integrated Manufacturing*, 12(1):15–38, 1999.

[109] K. Miyashita and K. P. Sycara. Cabins: A framework of knowledge acquisition and iterative revision for schedule improvement and reactive repair. *Artificial Intelligence*, 76(1-2):377–426, 1995.

[110] J. Modi and M. Veloso. Multiagent meeting scheduling with rescheduling. In *Proceedings of Proceedings of the Fifth Workshop on Distributed Constraint Reasoning, DCR 2004*, 2004.

[111] P. J. Modi, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.

[112] D. Morley and C. Schelberg. An analysis of a plant specific dynamic scheduler. In *NSF Workshop on Intelligent Dynamic Scheduling for Manufacturing Systems*, 1992.

[113] P. Morris and N. Muscettola. Managing temporal uncertainty through waypoint controllability. In *Proceedings of IJCAI'99*, pages 1253–1258, 1999.

[114] P. Morris and N. Muscettola. Dynamic control of plans with temporal uncertainty. In *In IJCAI*, pages 494–502, 2001.

[115] R. Morris, P. Morris, L. Khatib, and N. Yorke-Smith. Temporal planning with preferences and probabilities. In *Proceedings of ICAPS'05 Workshop on Constraint Programming for Planning and Scheduling*, June 2005.

[116] N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, October 2002.

[117] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Principles of Knowledge Representation and Reasoning*, pages 444–452, 1998.

[118] 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):5–47, 1998.

[119] D. Musliner, E. Durfee, J. Wu, D. Dogov, R. Goldman, and M. Boddy. Coordinated plan management using multiagent mdps. In *AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management*, pages 73–80, Stanford University, 2006.

[120] M. Naedele. Fault-tolerant real-time scheduling under execution time constraints. In *In Proc. of the Sixth International Conference on Real-Time Computing Systems and Applications*, 1999.

[121] R. Nair, M. Tambe, M. Yokoo, D. Pynadath, and S. Marsella. Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. In *Proceedings of IJCAI*, 2003.

[122] N. Onder and M. E. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *AAAI/IAAI*, pages 577–584, 1999.

[123] N. Onder, G. Whelan, and L. Li. Engineering a conformant probabilistic planner. *Journal of Artificial Intelligence Research*, 25:1–15, 2006.

[124] C. Ortiz. Varieties of emergent behavior in large-scale systems. In *Proceedings of the Autonmous Intelligent Networks and SystemsSymposium*, 2003.

[125] P. Ow, S. Smith, and R. Howie. A cooperative scheduling system. *Expert Systems and Intelligent Manufacturing*, pages 43–56, 1988.

[126] L. Parker. Alliance: An architecture for fault-tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, 1998.

[127] H. Parunak. Go to the ant: Engineering principles from natural multi-agent systems, 1999.

[128] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[129] B. Peintner and M. Pollack. Low-cost addition of preferences to dtps and tcsps. In *Proceedings of the 19th National Conference on Artificial Intelligence*, July 2004.

[130] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 266–271, Edinburgh, Scotland, Aug. 2005.

[131] A. Petcu and B. Faltings. Pc-dpop: A new partial centralization algorithm for distributed optimization. In *In Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI07*, pages 167–172, 2007.

[132] M. Pinedo. *Scheduling Theory, Algorithms and Systems*. Prentice Hall, 1995.

[133] N. Policella, A. Oddi, S. Smith, and A. Cesta. Generating robust partial order schedules. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, 2004.

[134] N. Policella, X. Wang, S. Smith, and A. Oddi. Exploiting temporal flexibility to obtain high quality schedules. In *AAAI*, 2005.

[135] M. Puterman. *Markov Decision Processes: Discreate Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[136] X. Qin and H. Jiang. A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing*, 32(5):331346, 2006.

[137] H. R. R. Bartak, T. Muller. A new approach to modelling and solving minimal perturbation problems. *Recent Advances in Constraints*, 2004.

[138] A. Raja, T. Wagner, and V. Lesser. Reasoning about Uncertainty in Design-to-Criteria Scheduling. In *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, pages 76–83, Stanford, CA, March 2000.

[139] J.-C. Regin. A filtering algorithm for constraints of difference in csps. In *The Twelth National Conference on Artificial Intelligence (AAAI)*, pages 362–367, 1994.

[140] W. Ruml and M. Fromherz. On-line planning and scheduling in a high-speed manufacturing domain. In *Proceedings of the ICAPS-04 Workshop on Integrating Planning into Scheduling*, 2004.

[141] N. Sadeh. Micro-opportunistic scheduling: The micro-boss factory scheduler. *Intelligent Scheduling*, 1994.

[142] H. E. Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, 2000.

[143] E. Salas and S. Fiore. *Team Cognition: Understanding the Factors that Drive Process and Performance*. American Psychological Association (APA), 2004.

[144] P. Scerri, Y. Xu, E. Liao, J. Lai, and K. Sycara. Scaling teamwork to very large teams. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 888–895, Washington, DC, USA, 2004. IEEE Computer Society.

[145] T. Schiex and G. Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[146] B. Sellner. Pro-active replanning for multi-robot teams. Technical report, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, June 2006.

[147] B. P. Sellner and R. Simmons. Towards proactive replanning for multi-robot teams. In *Proceedings of the 5th International Workshop on Planning and Scheduling in Space 2006*, October 2006.

[148] T. Shintani, I. Takayuki, and K. Sycara. Multiple negotiations among agents for a distributed meeting scheduler. In *Proceedings of the Fourth International Conference on Multiagent Systems*, pages 435–436, 2000.

[149] M. Sims, H. Mostafa, B. Horling, H. Zhang, V. Lesser, and D. Corkill. Lateral and Hierarchical Partial Centralization for Distributed Coordination and Scheduling of Complex Hierarchical Task Networks. In *AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management*, pages 113–120, Stanford University, 2006.

[150] D. Smith, J. Frank, and A. J'onsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1):61–94, 2000.

[151] D. Smith and D. Weld. Conformant graphplan. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 889–896, 1998.

[152] R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12), December 1980.

[153] S. Smith. Is scheduling a solved problem? In *Proceedings First MultiDisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, August 2003.

[154] S. Smith, M. Becker, and L. Kramer. Continuous management of airlift and tanker resources: A constraint-based approach. *Mathematical and Computer Modeling - Special Issue on Defense Transportation: Algorithms, Models and Applications for the 21st Century*, 39(6–8):581–598, 2004.

[155] S. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the 11th National Conference on Artificial Intelligence*, 1993.

[156] S. Smith, G. Cortellessa, D. Hildum, and C. Ohler. Using a scheduling domain theory to compute user-oriented explanations. In S. Castillo, Borrajo and Oddi, editors, *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice. Frontiers in Artificial Intelligence and Applications*, volume 117. IOS Press, 2005.

[157] S. Smith, A. Gallagher, T. Zimmerman, L. Barbulescu, and Z. Rubinstein. Multi-Agent Management of Joint Schedules. In *AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management*, pages 128–135, Stanford University, 2006.

[158] S. Smith, A. T. Gallagher, T. L. Zimmerman, L. Barbulescu, and Z. Rubinstein. Distributed management of flexible times schedules. In *Intl Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2007.

[159] S. Smith, D. Hildum, and D. Crimm. Interactive resource management in the comirem planner. In *Proceedings of IJCAI-03 Workshop on Mixed-Initiative Intelligent Systems*, Acapulco ME, August 2003.

[160] S. Smith, D. Hildum, and D. Crimm. Comirem: An intelligent form for scheduling. *IEEE Intelligent Systems*, 20(2), March-April 2005.

[161] S. Smith, N. Keng, and K. Kempf. Exploiting local flexibility during execution of pre-computed schedules. Technical Report CMU-RI-TR-90-13, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, June 1990.

[162] S. F. Smith. Reactive scheduling systems. In S. W. Brown D, editor, *Intelligent Scheduling Systems*. Kluwer Press, 1995.

[163] M. H. Sqalli and E. C. Freuder. Inference-based constraint satisfaction supports explanation. In *AAAI/IAAI, Vol. 1*, pages 318–325, 1996.

[164] M. Stanojevic, S. Vranes, and D. Velasevic. Using truth maintenance systems: A tutorial. *IEEE Expert: Intelligent Systems and Their Applications*, 9(6):46–56, 1994.

[165] J. Stedl and B. Williams. Managing communication limitations in partially controllable multi-agent plans. In *ICAPS-05 Workshop on Multiagent Planning and Scheduling*, pages 8–14, 2005.

[166] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Proceedings of the 15th National Conference of Artificial Intelligence, AAAI-98*, pages 81–117, 1998.

[167] D. Stillwell and J. Bay. Towards the development of a material transport system using swarms of ant-like robots. In *Proceedings of the International Conference on Robotics and Automation, ICRA-93*, pages 766–771, 1993.

[168] K. Sycara, M. Paolucci, M. V. Velsen, and J. Giampapa. The retsina mas infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):29–48, 2003.

[169] K. Sycara, S. F. Roth, N. Sadeh-Koniecpol, and M. S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1446–1461, December 1991.

[170] P. Szekely, R. Maheswaran, R. Neches, C. Rogers, R. Sanchez, M. Becker, S. Fitzpatrick, G. Gati, D. Hanak, G. Karsai, and C. van Burskirk. An examination of critically-sensitive approaches to coordination. In *AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management*, pages 136–142, Stanford University, 2006.

[171] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research, JAIR*, 7:83–124, 1997.

[172] G. Theraullaz, S. Goss, J. Gervet, and J. Deneubourg. Task differentiation in pollistes wasp colonies: a model for self-organizing groups of robots. In *Proceedings of the 1st International Conference on Simulation of Adaptive Behavior*, pages 346–355, 1990.

[173] I. Tsamardinos. Fast transformation of temporal plans for efficient execution. In *In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-1998*, pages 254–261, 1998.

[174] I. Tsamardinos, T. Vidal, and M. Pollack. Ctp: A new constraint-based formalism for conditional, temporal planning. *Constraints*, 8(4):365–388, 2003.

[175] D. Vail and M. Veloso. Dynamic multi-robot coordination. *Multi-Robot Systems*, 2003.

[176] W.-J. van Hoeve, C. Gomes, M. Lombardi, and B. Selman. Optimal multi-agent scheduling with constraint programming. In *Proceedings of the Nineteenth Conference on Innovative Applications of Artificial Intelligence (IAAI)*, 2007.

[177] P. Varakanthan and S. Smith. Linear relaxation techniques for task management in uncertain settings. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

[178] K. Venable and N. Yorke-Smith. Simple temporal problems with preferences and uncertainty, 2003.

[179] K. Venable and N. Yorke-Smith. Disjunctive temporal planning with uncertainty. In *Proceedings of IJCAI'05*, August 2005.

[180] T. Vidal and H. Fargier. Contingent durations in temporal csps: From consistency to controllabilities. In *TIME*, pages 78–85, 1997.

[181] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *JETAI*, 11(1):23–45, 1999.

[182] T. Wagner and V. Lesser. Design-to-criteria scheduling: Real-time agent control. In *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, pages 89–96, Stanford, CA, March 2000.

[183] T. A. Wagner, A. J. Garvey, and V. R. Lesser. Criteria Directed Task Scheduling. *Journal for Approximate Reasoning (Special Issue on Scheduling)*, 19:91–118, January 1998.

[184] R. J. Wallace and E. C. Freuder. Dispatchable execution of schedules involving consumable resources. In *Artificial Intelligence Planning Systems*, pages 283–290, 2000.

[185] X. Wang and S. Smith. Retaining flexibility to maximize quality: When the scheduler has the right to decide activity durations. In *Proceedings of the 15th*

International Conference on Automated Planning and Scheduling, ICAPS-05, June 2005.

[186] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1), 1962.

[187] A. Wehowsky, S. Block, and B. Williams. Robust distributed coordination of heterogeneous robots through temporal plan networks. In *ICAPS-05 Workshop on Multiagent Planning and Scheduling*, pages 67–72, 2005.

[188] G. Weiss. *Multiagent systems: A modern approach to distributed artificial intelligence*. The MIT Press, 1999.

[189] B. Williams and R. Ragno. Conflict-directed a* and its role in model-based embedded systems. *Journal of Discrete Applied Math*, 2003.

[190] Y.Oh and S.H.Son. Scheduling real-time tasks for dependability. *Journal of Operational Research Society*, 48(6):629639, 1997.

[191] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[192] H. Younes. Planning and execution with phase transitions. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, page 10301035, 2005.

[193] H. Younes and R. Simmons. Solving generalized semi-markov decision processes using continuous phase-type distributions. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, page 742747, 2004.

[194] W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 185–192, New York, NY, USA, 2003. ACM.

[195] Q. Zhou and S. Smith. A priority-based preemption algorithm for incremental scheduling with cumulative resources. Technical Report CMU-RI-TR-02-19, Carnegie Mellon University, July 2002.

[196] R. Zlot, A. Stentz, M. Dias, and S. Thayer. Multi-robot exploration controlled by a market economy. In *Proceedings of the International Conference on Robotics and Automation, ICRA-02*, 2002.

[197] M. Zweben, B. Daun, E. Davis, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann Publishers, 1994.