# Distributed Method Selection and Dispatching of Contingent, Temporally Flexible Plans

by

Stephen Block

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

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Aeronautics and Astronautics
February 1st, 2007

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

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

# Distributed Method Selection and Dispatching of Contingent, Temporally Flexible Plans

by

Stephen Block

Submitted to the Department of Aeronautics and Astronautics
on February 1st, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

## Abstract

Many applications of autonomous agents require groups to work in tight coordination. To be dependable, these groups must plan, carry out and adapt their activities in a way that is robust to failure and to uncertainty. Previous work developed contingent, temporally flexible plans. These plans provide robustness to uncertain activity durations, through flexible timing constraints, and robustness to plan failure, through alternate approaches to achieving a task. Robust execution of contingent, temporally flexible plans consists of two phases. First, in the plan extraction phase, the executive chooses between the functionally redundant methods in the plan to select an execution sequence that satisfies the temporal bounds in the plan. Second, in the plan execution phase, the executive dispatches the plan, using the temporal flexibility to schedule activities dynamically.

Previous contingent plan execution systems use a centralized architecture in which a single agent conducts planning for the entire group. This can result in a communication bottleneck at the time when plan activities are passed to the other agents for execution, and state information is returned. Likewise, a computation bottleneck may also occur because a single agent conducts all processing.

This thesis introduces a robust, *distributed* executive for temporally flexible plans, called *Distributed-Kirk*, or *D-Kirk*. To execute a plan, D-Kirk first distributes the plan between the participating agents, by creating a hierarchical ad-hoc network and by mapping the plan onto this hierarchy. Second, the plan is reformulated using a distributed, parallel algorithm into a form amenable to fast dispatching. Finally, the plan is dispatched in a distributed fashion.

We then extend the D-Kirk distributed executive to handle contingent plans. Contingent plans are encoded as Temporal Plan Networks (TPNs), which use a non-deterministic choice operator to compose temporally flexible plan fragments into a nested hierarchy of contingencies. A temporally consistent plan is extracted from the TPN using a distributed, parallel algorithm that exploits the structure of the TPN.

At all stages of D-Kirk, the communication load is spread over all agents, thus eliminating the communication bottleneck. In particular, D-Kirk reduces the peak communication complexity of the plan execution phase by a factor of $O\left(\frac{A}{e'}\right)$, where $e'$ is the number of edges per node in the dispatchable plan, determined by the branching factor of the input plan, and $A$ is the number of agents involved in executing the plan.

In addition, the distributed algorithms employed by D-Kirk reduce the computational load on each agent and provide opportunities for parallel processing, thus increasing efficiency. In particular, D-Kirk reduces the average computational complexity of plan dispatching from $O\left(N^3 e\right)$ in the centralized case, to typical values of $O\left(N^2 e\right)$ per node and $O\left(\frac{N^3 e}{A}\right)$ per agent in the distributed case, where $N$ is the number of nodes in the plan and $e$ is the number of edges per node in the input plan.

Both of the above results were confirmed empirically using a C++ implementation of D-Kirk on a set of parameterized input plans. The D-Kirk implementation was also tested in a realistic application where it was used to control a pair of robotic manipulators involved in a cooperative assembly task.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor

# Acknowledgments

I would like to thank the members of the Model-Based Embedded and Robotic Systems group at MIT for their help in brainstorming the details of the D-Kirk algorithm and for their valuable feedback throughout the process of writing this thesis. Particular thanks are due to Seung Chung, Andreas Hoffman and Brian Williams. I would also like to thank my family and friends for their support throughout.

The plan extraction component of the work presented in this thesis was made possible by the sponsorship of the DARPA NEST program under contract F33615-01-C-1896. The plan reformulation and dispatching components were made possible by the sponsorship of Boeing contract MIT-BA-GTA-1.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

The ability to coordinate groups of autonomous agents is key to many real-world tasks. An example scenario is the construction of a lunar habitat, where multiple robotic manipulators are required to work together to assemble the structure. For example, groups of manipulators may be required to act in coordination to lift large or heavy objects. A second scenario is an exploration mission on a foreign planet. This may involve multiple robots, each with a different skill set. For example, one robot with a high top speed may be sent ahead as a scout while a second, slower robot with a larger payload capacity is sent to the areas of interest discovered by the scout to collect samples. A final scenario is in manufacturing, where multiple robots simultaneously carry out different operations on a part. For example, one robot may hold a part in position while a second welds it to the main structure.

Coordinating such a group of agents involves executing an activity plan that describes the actions each agent will perform. In order for the execution to be reliable we must provide robustness to unknown events and to disturbances. We provide robustness through an executive which performs dynamic execution of temporally flexible plans.

## 1.2 Previous Work

Previous work on robust plan execution has provided robustness to three types of unknowns: temporal uncertainty, execution uncertainty and plan failure.

*Temporal Uncertainty.* For many systems, the precise duration of an activity is not fixed. It may be acceptable to conduct the activity for any period of time within a given range, or it may be the case that the duration of the activity is beyond the control of the executive. Temporally flexible plans [4] [19] allow us to accommodate both cases by modeling activities of uncertain duration. Use of these plans allows us to provide robustness to variation in the execution times of activities, as discussed below.

*Execution Uncertainty.* Traditional execution scheme assign execution times at planning time, such that any temporal flexibility in the plan has been eliminated by dispatch time. This means that the executive is unable to adapt to uncertainty in execution times. To overcome this problem, while executing the plan in a timely manner, a dispatcher is used that dynamically schedules events. In particular, we use the methods of Tsamardinos et. al. [27], which are based on a least commitment strategy. This scheme is known as dispatchable execution [20], and provides robustness to execution uncertainty. Here, scheduling is postponed until execution time, allowing temporal resources to be assigned as they are needed. This allows the dispatcher to respond to the variable duration of activities at execution time. To minimize the computation that must be performed on-line at dispatch time, the plan is first reformulated off-line.

*Plan Failure.* Plan failure occurs when the temporal constraints in the plan are such that no satisfying execution schedule can be found. To address plan failure, Kim introduced a system called *Kirk* [12], that performs dynamic execution of temporally flexible plans with contingencies. These contingent plans are encoded as alternative choices between functionally equivalent sub-plans. In Kirk, the contingent plans are represented by a Temporal Plan Network (TPN) [12], which extends temporally flexible plans with a nested choice operator. Kirk first extracts a plan from the TPN that is temporally feasible, before executing the plan, as described above. Use of contingent plans adds robustness to plan failure.

Kirk uses a centralized architecture in which a single master agent is responsible for generating and executing an activity plan. In the case of a multi-agent plan, all other agents are inactive during the plan extraction phase. During plan execution, the other agents are simply instructed by the master agent to execute their activities at the relevant time. This means that the master agent must communicate with each agent at the time of execution of its activity and this creates a communication bottleneck at the master agent. As a result, the executive lacks robustness to communication latency. Similarly, the master agent may also suffer from a computational bottleneck, as it must process the plan for all of the agents involved in the plan.

Recent work by Wehowsky produced a distributed version of the plan extraction component of Kirk [31]. Prior to plan extraction, the TPN is distributed between the available agents. However, this work does not provide a distributed version of the reformulation or dispatching algorithms.

## 1.3 Problem Statement

The control of autonomous agents to achieve useful, real-world tasks requires activity plans that involve groups of agents. Reliable execution of these plans in an uncertain environment requires robustness to temporal uncertainty, execution uncertainty, plan failure and communication latency.

The problem solved by this thesis is to create a plan execution system for autonomous agents that is robust to these four types of uncertainty. The executive takes as input a multi-agent activity plan in the form of a TPN, extracts a temporally consistent plan from the contingencies available and schedules activities dynamically to respond to variable execution times. It does this in a way that avoids bottlenecks when communicating between agents.

## 1.4 Example Scenario

To motivate the discussion of our work, we introduce an example task taken from the lunar construction scenario. Consider the task of assembling a lunar structure using numerous robotic manipulators.

An example task which requires tight coordination of the manipulators is the delivery of a tool to a specific location, perhaps for use by a human astronaut. In this example there are two robotic manipulators. The action to be taken by the manipulators depends upon the initial location of the tool. For some initial locations, a single manipulator can pick up the tool and deliver it to the desired location. For other initial locations, both manipulators must work together: one manipulator picks up the tool and passes it to the second manipulator for delivery. This task is depicted in graphical form in Fig. 1-1. This example is used to demonstrate our work on a pair of Barret Technology's Whole Arm Manipulators (WAMs) in Section 7.3. For this reason, we refer to the two manipulators as WAM0 and WAM1.

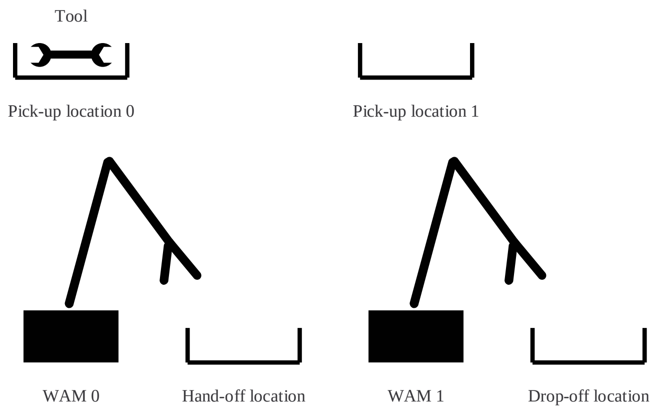


Figure 1-1: This figure shows a cooperative assembly task where two manipulators, WAM0 and WAM1, must deliver a tool to a drop-off location. The drop-off location is reachable by manipulator WAM1 only. Pick-up locations 0 and 1 are reachable by manipulators WAM0 and WAM1 respectively. Both manipulators can reach the hand-off location.

## 1.5 Proposed Approach

We address this problem with a distributed architecture in which all agents participate in the execution process. Agents communicate by sending and receiving messages and this communication is spread evenly between all agents. This evens out communication requirements and eliminates the communication bottleneck, thus providing robustness to communication latency. We distribute computation between all agents to reduce computational complexity and to take advantage of parallel processing, thus improving performance relative to the centralized architecture. The greater the number of agents that is involved in the plan, the greater the extent to which the execution process can be shared out, so the greater the performance improvements.

Finally, our distributed architecture provides a framework for a future distributed system capable of providing robustness to loss of an agent during plan execution. Such a system would detect failure of an agent participating in plan execution, redistribute the plan over the remaining agents, re-plan if necessary, and continue execution.

Our proposed executive is a distributed version of Kirk, called *D-Kirk*, which performs *distributed* execution of contingent temporally flexible plans. D-Kirk consists of the following four phases.

1. Form a structured network of agents and distribute the TPN across the network

2. Select a temporally consistent plan from the TPN

3. Reformulate the selected plan for efficient dispatching

4. Dispatch the selected plan, while scheduling dynamically

The four phases are described below in the context of the manipulator tool delivery task introduced above. A comprehensive summary of the reformulation and dispatching phases of execution in the centralized case is provided by Stedl [25].

A TPN representing the tool passing task for a pair of WAMs is shown in graphical form in Fig. 1-2. Nodes represent time events and directed edges represent simple temporal constraints. A simple temporal constraint $[l, u]$ places a bound $t^{+} - t^{-} \in [l, u]$

on the temporal duration between the start time $t^-$ and end time $t^+$ of the activity or sub-plan to which it is applied.

The objective of the plan is to deliver the tool to the drop-off location, which is reachable by WAM1 only. A tool is available at pick-up locations 0 and 1, but the time at which the tool becomes available at each location is unknown until execution time. Depending upon the time at which the tool becomes available at a particular location, it may not be possible to deliver the tool from that pick-up location to the drop-off location within the time bounds specified by the plan. Therefore, the plan contains two contingencies; one in which the tool from pick-up location 0 is used, and another in which the tool from pick-up location 1 is used. At execution time, the executive selects a contingency, if possible, which guarantees successful execution of the plan. The contingencies are represented in the plan by the single choice node, node 2.

In the case where the tool is at pick-up location 0, which is reachable by WAM0 only, the task can only be completed if the manipulators cooperate. This contingency is shown in the upper half of Fig. 1-2. First, WAM0 moves its hand to pick-up location 0, where it waits for the tool to arrive. When the tool arrives, WAM0 closes its hand around the tool and moves the tool to the hand-off location. At the same time, WAM1 moves to the hand-off location. Having completed these activities, either WAM will wait until the other WAM has completed its activities. Next, WAM1 closes its hand around the tool and one second after completing the close, WAM0 begins to open its hand. Within a second of WAM0 opening its hand, WAM1 moves the tool to the drop-off location, where it immediately opens its hand to release the tool and finally moves to home position 1. Simultaneously, WAM0 moves directly to home position 0. Again, having completed these activities, either WAM will wait until the other WAM has completed its activities, before the plan terminates.

In the case where the tool is at pick-up location 1, which is reachable by WAM1 only, the task can be completed by WAM1 alone. This contingency is shown in the lower half of Fig. 1-2. First, WAM1 moves its hand to pick-up location 1, where it waits for the tool to arrive. Once the tool has arrived, WAM1 closes its hand around

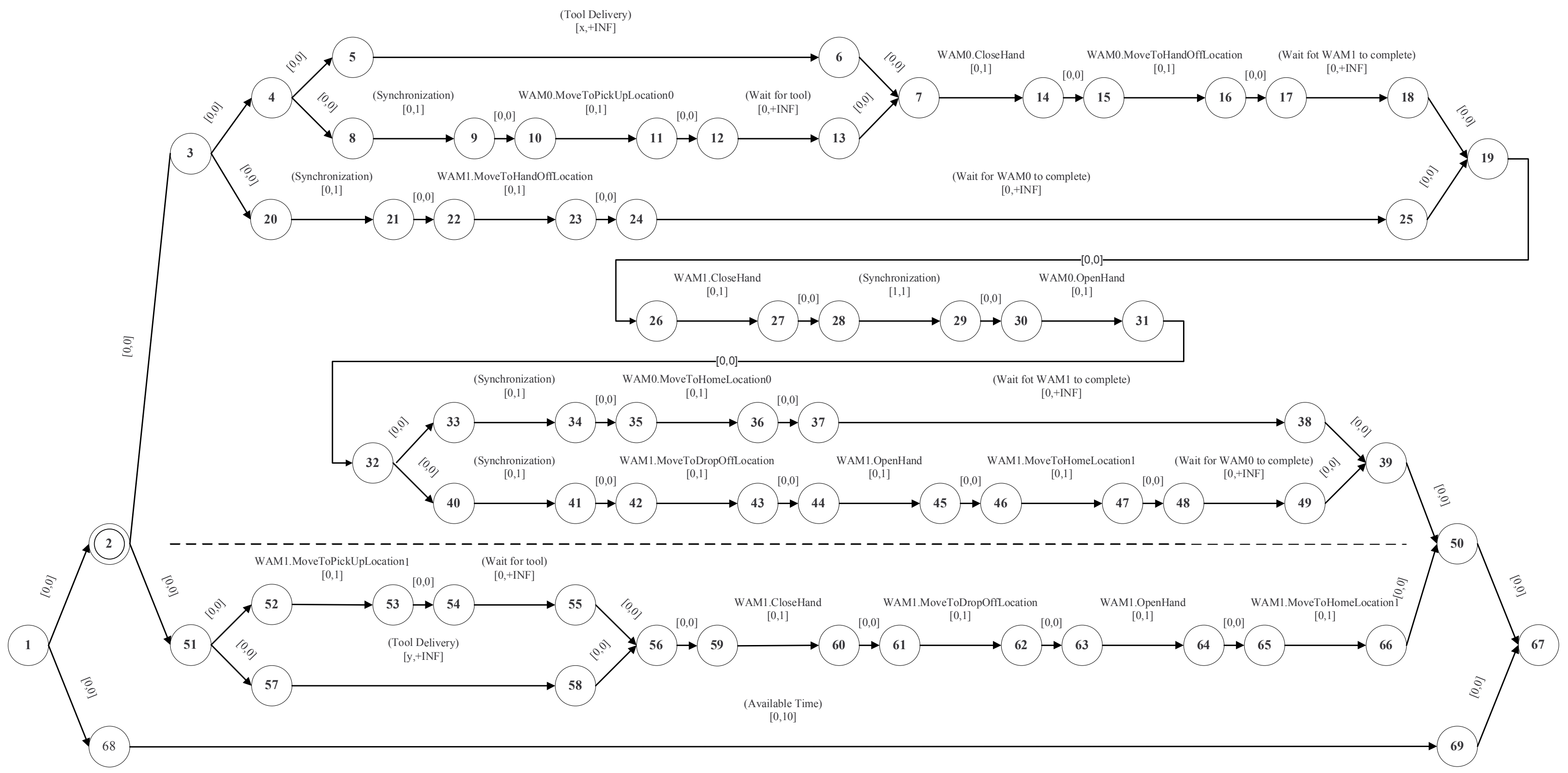


Figure 1-2: This figure shows a graph representation of a plan describing a cooperative assembly task.

the tool and moves it to the drop-off location, where it opens its hand to release the tool and finally moves to home position 1.

The simple temporal constraints $[x, +INF]$ and $[y, +INF]$, between nodes 5 and 6 and nodes 57 and 58 respectively, represent the time we must wait from the start of the plan until the tool becomes available at pick-up locations 0 and 1 respectively. The values of $x$ and $y$, the minimum wait times, are not known until execution time. The simple temporal constraint $[0, 10]$ between nodes 68 and 69 imposes an upper bound of 10s on the overall duration of the plan.

### 1.5.1 Distribution of the TPN across the Processor Network

Our objective in distributing the TPN is to share the plan data between the agents taking part in the execution. We assume that tasks have already been assigned to agents and distribute the TPN such that each agent is responsible for the plan nodes corresponding to the activities it must execute.

Before we can perform the distribution, we must arrange the agents into an ordered structure. This structuring defines which agents are able to communicate directly and establishes a system of communication routing, such that all agents can communicate. In D-Kirk, we form a hierarchy of agents using the methods of Coore et. al. [2]. In the case of the tool passing example only two agents are involved in the plan, so the hierarchy is trivial and the choice of WAM0 as the leader of the hierarchy is arbitrary. The agent hierarchy is shown in Fig. 1-3.

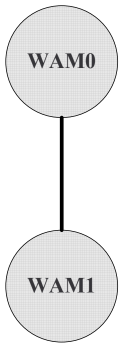


Figure 1-3: This figure shows the trivial agent hierarchy for the two agents involved in the tool delivery task. The choice of WAM0 as the leader is arbitrary.

Having formed the agent hierarchy, we assign plan nodes to agents. White nodes are assigned to WAM0 while gray nodes are assigned to WAM1, as shown in Fig. 1-4.

### 1.5.2 Selecting a Temporally Consistent Plan from the TPN

The simple temporal constraints $[x, +INF]$ and $[y, +INF]$ represent the time we must wait from the start of the plan until the tool becomes available at pick-up locations 0 and 1 respectively. The values of $x$ and $y$, the minimum wait times, determine whether or not each contingency in the plan is temporally feasible. The plan selection phase of D-Kirk takes the TPN, including known values for $x$ and $y$ and extracts a temporally consistent plan by selecting the appropriate contingency.

For the purposes of this example, we assume that $x = 1$ and $y = 20$. This means that the lower bound on the execution time of the upper contingency is 2, which is less than the plan's overall upper time bound of 10. The lower bound on the execution time of the lower contingency, however, is 20, which exceeds the overall upper time bound. Therefore, D-Kirk selects the upper contingency, which involves the two manipulators working together. The resulting temporally consistent plan is shown in Fig. 1-5.

### 1.5.3 Reformulating the Selected Plan for Dispatching

Dispatchable execution retains the temporal flexibility present in the input plan throughout the planning process, allowing it to be used to respond to unknown execution times at dispatch time. However, this approach increases the complexity of dispatching, relative to a scheme in which execution times are fixed before dispatching. To minimize the amount of computation that must be performed in real time during dispatching, we first reformulate the plan. The output of the reformulation phase is a Minimal Dispatchable Graph (MDG), which requires only local propagation of execution time data at dispatch time.

The plan extracted from the TPN can be represented as a Simple Temporal Network (STN) and every STN has an equivalent distance graph. In a distance graph,

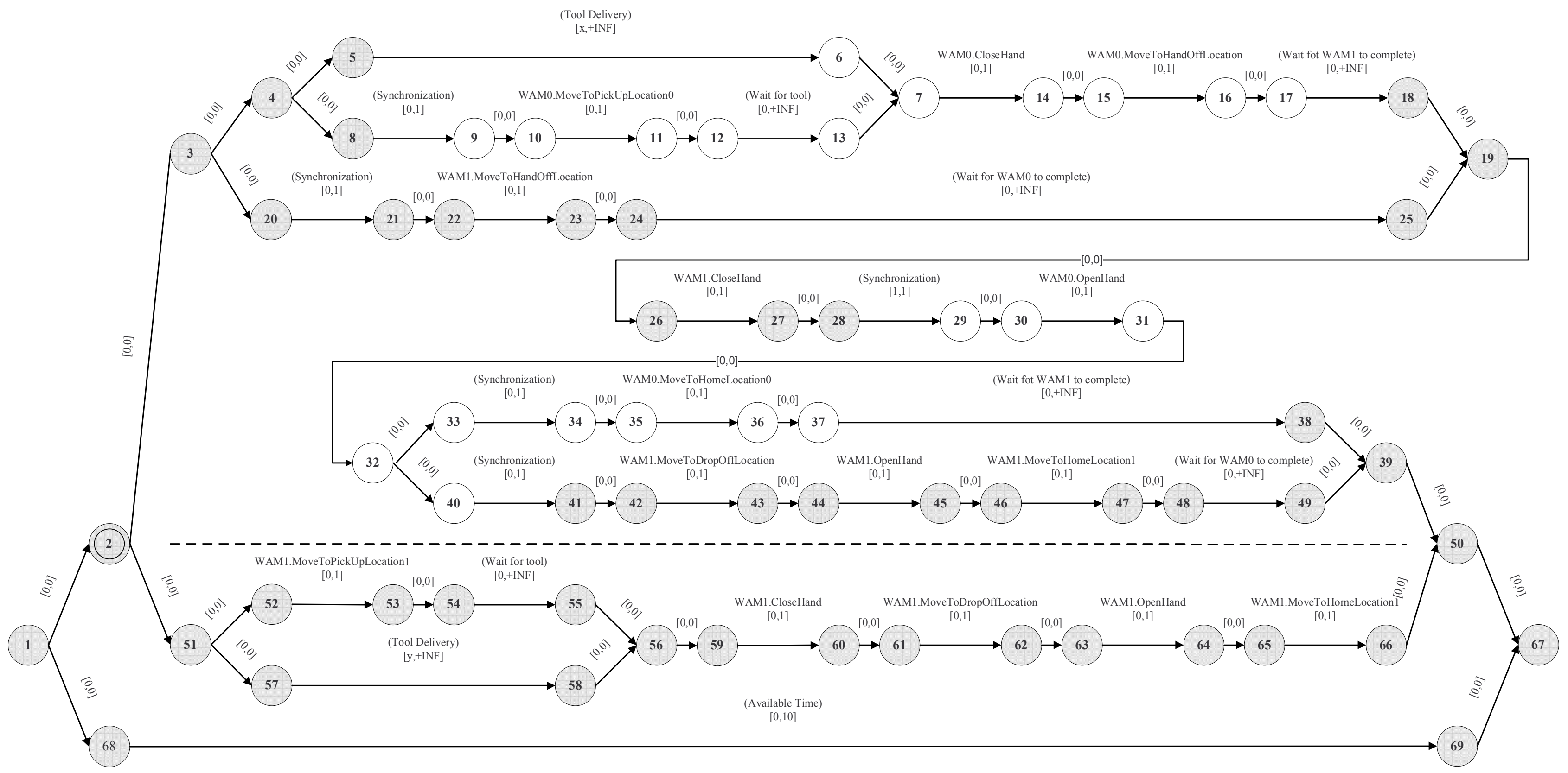


Figure 1-4: This figure shows the TPN after distribution between the agents involved in the execution. White nodes are assigned to WAM0 and gray nodes to WAM1.

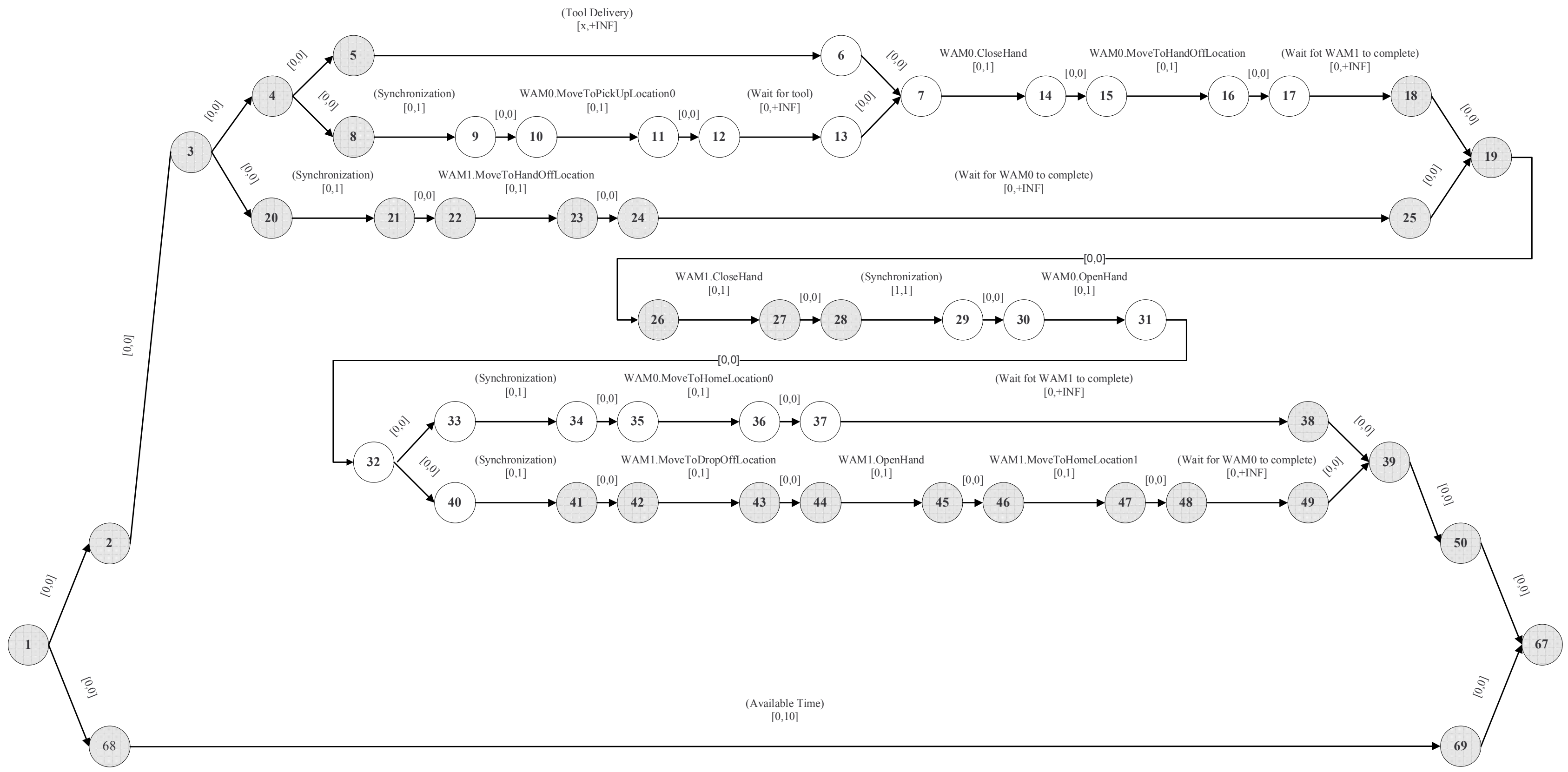


Figure 1-5: This figure shows the STN corresponding to the temporally consistent plan selected from the TPN.

nodes represent time events and directed edges represent a single temporal constraint. Specifically, each edge $AB$ in the distance graph has a length or weight $b(A, B)$ which specifies the constraint $t_B - t_A \leq b(A, B)$, where $t_A$ and $t_B$ are the execution times of nodes A and B respectively. Reformulation operates on the distance graph representation of the selected temporally flexible plan and the MDG is itself a distance graph. The MDG for the cooperative assembly task example is shown in Fig. 1-6.

An important feature of the MDG is that all Rigid Components (RCs), that is groups of nodes whose execution times are fixed relative to each other, have been rearranged. In particular, the member nodes of the RC are arranged in a chain, in order of execution time, linked by pairs of temporal constraints. The member node with the earliest execution time, at the head of the chain, is termed the RC leader. All temporal constraints between the RC's member nodes and the rest of the plan graph are re-routed to the leader. Also, RC member nodes linked by temporal constraints of zero duration, are collapsed to a single node. Members of these Zero-Related (ZR) components must execute simultaneously, so any start or end events are transferred to the remaining node. An example of a rearranged RC is nodes 27 through 30 in Fig. 1-5. Nodes 27 and 28 and nodes 29 and 30 are ZR components, so are collapsed to nodes 28 and 30 respectively in Fig. 1-6. Node 28 is the RC leader so the edges between node 30 and nodes outside the RC are rerouted to node 28.

In the case of the fast reformulation algorithm used in D-Kirk, rearrangement of rigid components in this way is required for the correct operation of the algorithm used to determine the distance graph edges that belong in the MDG, as described below. Also, the rearrangement improves the efficiency of reformulation.

Once RCs have been processed, the reformulation algorithm identifies all edges that belong in the MDG. Some of the original edges of the input STN remain in the MDG, while others are eliminated. In addition, the MDG contains new edges which are required to make explicit any implicit timing constraints required for dispatching. An example of such an edge is that of length 9 from node 1 to node 13 in Fig. 1-6. It ensures that node 13 is executed sufficiently early that the upper bound of 10 units on the overall plan length can be met, given that later in the plan (between nodes 28

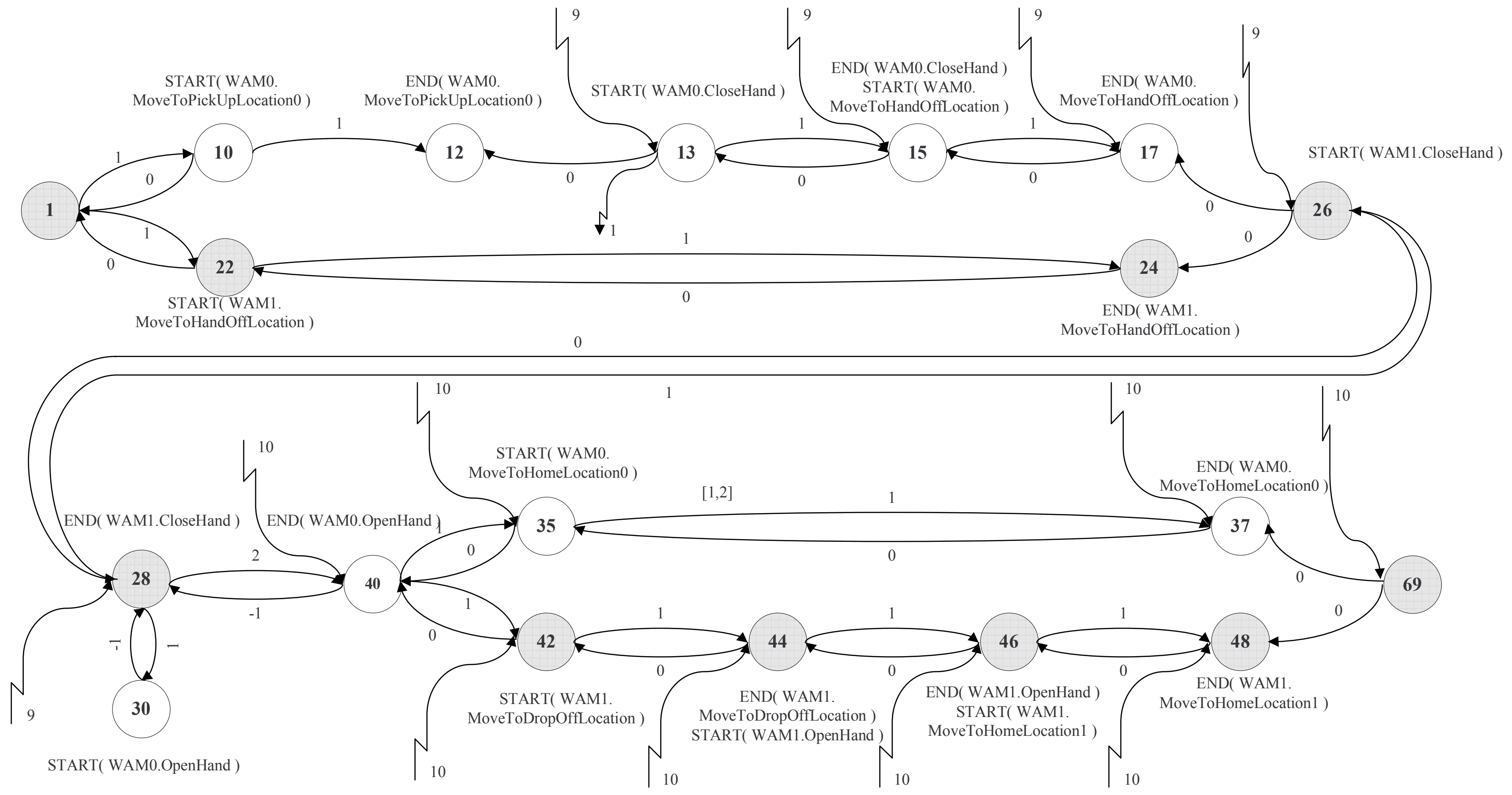

START( WAM0. MoveToPickUpLocation0 )
END( WAM0. MoveToPickUpLocation0 )
START( WAM0.CloseHand )
END( WAM0.CloseHand ) START( WAM0. MoveToHandOffLocation )
END( WAM0. MoveToHandOffLocation )
START( WAM1.CloseHand )
START( WAM1. MoveToHandOffLocation )
END( WAM1. MoveToHandOffLocation )
START( WAM0. MoveToHomeLocation0 )
END( WAM0. MoveToHomeLocation0 )
END( WAM1.CloseHand )
END( WAM0.OpenHand )
START( WAM0.OpenHand )
START( WAM1. MoveToDropOffLocation )
END( WAM1. MoveToDropOffLocation ) START( WAM1.OpenHand )
END( WAM1.OpenHand ) START( WAM1. MoveToHomeLocation1 )
END( WAM1. MoveToHomeLocation1 )
1
10
12
13
15
17
22
24
26
28
30
35
37
40
42
44
46
48
69
[1,2]

Figure 1-6: This figure shows the Minimal Dispatchable Graph for the cooperative assembly task plan. Many nodes have been eliminated as a result of collapsing Zero-Related groups, Rigid Components have been rearranged and the non-dominated edges have been calculated. Some edges to and from node 1 are shown as stepped lines for clarity.

and 40 in the MDG in Fig. 1-6) there exists a minimum duration of 1 unit.

During reformulation, the temporal constraints between the start and end nodes of an activity in the input plan may be removed if the constraints do not belong in the MDG, as described above. This means that activities no longer necessarily coincide with temporal constraints. For this reason, activities are instead represented as a pair of events: a start event at the start node and an end event at the end node. An example of this is the WAM0.OpenHand activity between nodes 30 and 31 in the STN in Fig. 1-5, or equivalently nodes 30 and 40 in the MDG in Fig. 1-6.

### 1.5.4 Dispatching the Selected Plan

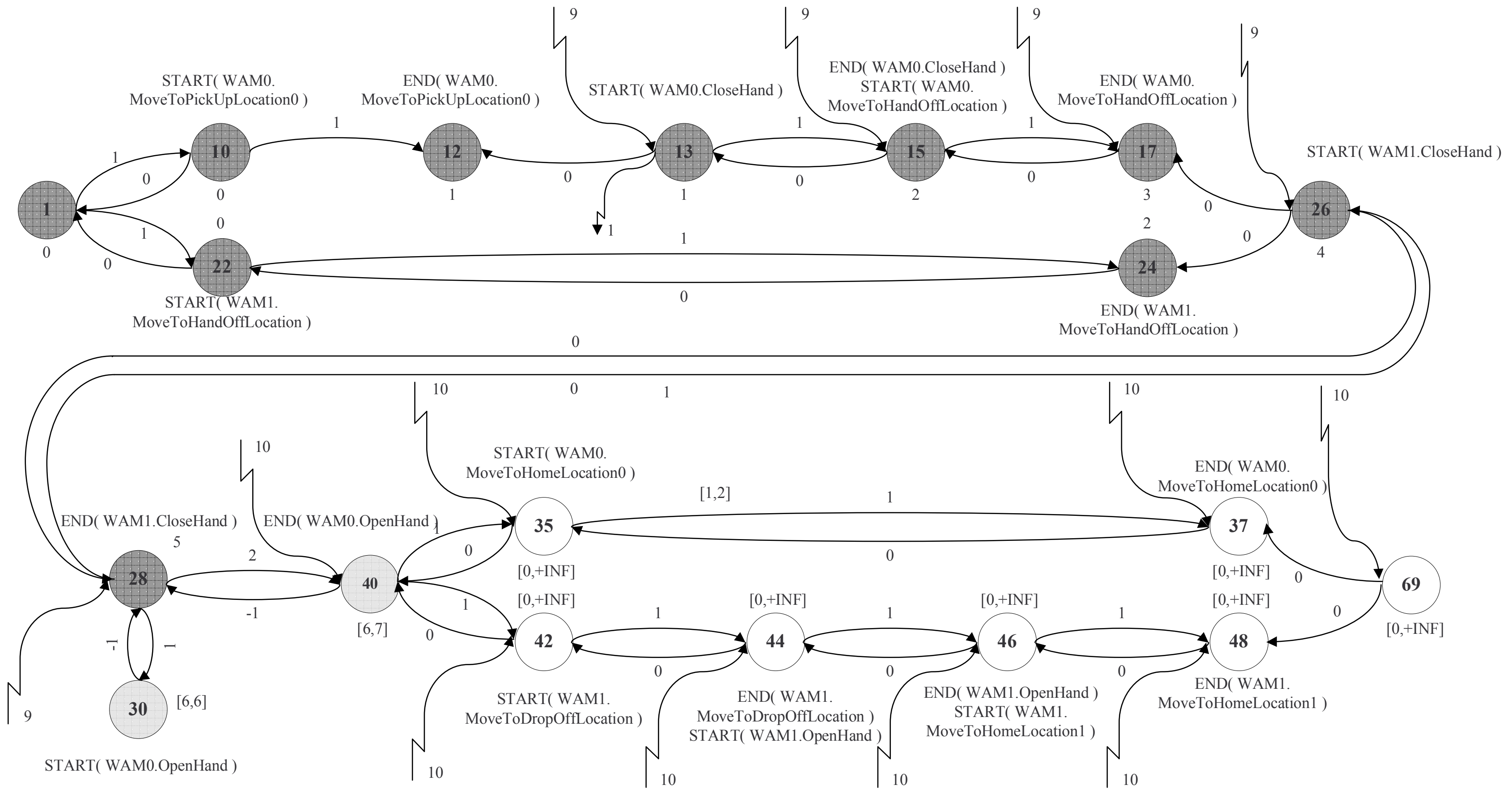
During dispatching each node maintains its execution window. The execution window is the range of possible execution times for the node, given the temporal constraints in the plan and the execution times of nodes that have already been executed. The start node, node 1 in the tool delivery example, is assigned an execution time of 0 by definition.

The dispatching algorithm uses an internal clock, the execution windows of the nodes and the temporal constraints from the MDG to determine when each node can be executed. We use a minimum execution time policy: where there exists a range of possible execution times, nodes are executed as soon as possible. When a node is executed the MDG edges are used to update the status of neighbor nodes.

A node must be both *alive* and *enabled* before it can be executed. A node is enabled when all nodes that must execute before it have executed. A node is alive when the current time lies within its execution window. In addition, *contingent* nodes, which represent the end event of uncontrollable activities, must also wait for the uncontrollable activities to complete.

Fig. 1-7 shows a snapshot of the MDG for the tool delivery example during dispatching at time 5. Executed nodes are shown in dark gray, with their execution times. Enabled nodes which are not yet alive are shown in light gray.

The distributed dispatching algorithm monitors the node for the enablement condition. The algorithm uses incoming EXECUTED messages from neighbor nodes



START( WAM0. MoveToPickUpLocation0 )
END( WAM0. MoveToPickUpLocation0 )
START( WAM0.CloseHand )
END( WAM0.CloseHand ) START( WAM0. MoveToHandOffLocation )
END( WAM0. MoveToHandOffLocation )
START( WAM1.CloseHand )
START( WAM1. MoveToHandOffLocation )
END( WAM1. MoveToHandOffLocation )
END( WAM1.CloseHand )
END( WAM0.OpenHand )
START( WAM0. MoveToHomeLocation0 )
END( WAM0. MoveToHomeLocation0 )
START( WAM0.OpenHand )
START( WAM1. MoveToDropOffLocation )
END( WAM1. MoveToDropOffLocation ) START( WAM1.OpenHand )
END( WAM1.OpenHand ) START( WAM1. MoveToHomeLocation1 )
END( WAM1. MoveToHomeLocation1 )
1
10
12
13
15
17
22
24
26
28
30
35
37
40
42
44
46
48
69
[6,6]
[6,7]
[1,2]
[0,+INF]

Figure 1-7: This figure shows the Minimal Dispatchable Graph for the cooperative assembly task plan at time 5 during dispatching. Executed nodes are shown in dark gray, with their execution times. Other nodes are labeled with their current execution window. Enabled nodes which are not yet alive are shown in light gray.

to update the enablement condition and the execution window. Once enabled, the algorithm waits for the current time to enter the execution window, while continuing to use EXECUTED messages to update the upper bound of the execution window. Once alive, and in the case of a contingent node, all uncontrollable activities ending at the node have completed, the node can be executed. Executing a node involves stopping all controllable activities that end at the node and starting all activities which start at the node. When a node is executed, EXECUTED messages are sent to its neighbor nodes so that they can update their execution window and enablement status.

For example, in Fig. 1-7, the last node to have been executed is node 28 at time 5. The non-positive MDG edges to node 28 from nodes 30 and 40 have been used to enable these nodes and to update the lower bound of their execution window to 6 in each case. The positive MDG edges from node 28 to nodes 30 and 40 have been used to update the upper bound of the execution window of these nodes to 6 and 7 respectively.

## 1.6 Key Technical Contributions

D-Kirk performs *distributed* execution of contingent temporally flexible plans. The technical insights employed by each phase of the algorithm are summarized below.

1. **Distribute the TPN across the processor network.** We arrange agents into a hierarchy to provide the required communication availability. We then map the TPN to the hierarchy, exploiting the hierarchical nature of the TPN to maximize efficiency.

2. **Select a temporally consistent plan from the TPN.** We use a distributed consistency checking algorithm and interleave candidate generation and consistency checking to exploit parallel processing.

3. **Reformulate the selected plan for efficient dispatching.** We use a distributed version of the fast reformulation algorithm, using parallelism wherever

possible. The algorithm uses a state machine approach with message passing that is robust to variable message delivery times.

4. **Dispatch the selected plan, while scheduling dynamically.** The algorithm uses a state machine approach with message passing that is robust to variable message delivery times.

## 1.7 Performance and Experimental Results

The main objective of distributed D-Kirk is to eliminate the communication bottleneck present at the master node of the centralized architecture when the plan is dispatched. The peak communication complexity of the D-Kirk dispatching algorithm is $O(e')$ per node or $O\left(\frac{Ne'}{A}\right)$ per agent on average, where $e'$ is the number of edges connected to each node in the plan after reformulation for dispatching, $A$ is the number of agents involved in the plan execution and $N$ is the number of nodes in the plan. In the centralized case, the peak message complexity is $O(N)$. This means that D-Kirk reduces the peak message complexity at dispatch time, when real-time operation is critical, by a typical factor of $O\left(\frac{A}{e'}\right)$ per agent. Note that $e'$ is typically a small constant, determined by the branching factor of the plan.

D-Kirk also improves the computational complexity of the reformulation algorithm relative to the centralized case. The computational complexity is reduced from $O(N^2E)$ in the centralized case to $O(N^2e)$ per node or $O\left(\frac{N^3e}{A}\right)$ per agent in the distributed case, where $E$ is the total number of edges in the input plan and $e$ is the number of edges connected to each node in the input plan. Typically, $E = O(Ne)$, so the improvement is a factor of $O(A)$.

Both of the above analytical results are verified by experimentation using a C++ implementation of D-Kirk operating on parameterized sets of plans of numerous types. In addition, we use the C++ implementation to demonstrate the applicability of D-Kirk in a realistic setting. We use D-Kirk to execute the manipulator tool delivery example plan in our robotic manipulator testbed. This consists of a pair of Barrett Technology's Whole Arm Manipulators (WAMs), each with four degrees of freedom and a three-fingered hand effector.

## 1.8 Thesis Layout

First, in Chapter 2, we review methods for the robust execution of contingent temporally flexible plans. We discuss how dispatchable execution is used to robustly execute a temporally flexible plan, present a definition of a TPN and give an overview of plan selection in the centralized case. In Chapter 3 we present the first stage of D-Kirk; formation of an agent hierarchy to provide the required communication between agents and the distribution of the TPN over the group of agents. Chapters 4 and 5 present the details of the two stage process required by the dispatchable execution approach to execute a temporally flexible plan. First, reformulation, where the plan is recompiled so as to reduce the real-time processing required at execution time. Second, dispatching, where the activities in the plan are executed. This is the core of the D-Kirk distributed executive. In Chapter 6 we extend the distributed executive to handle contingent plans. This completes the description of D-Kirk. Finally, in Chapter 7, we present experimental results which demonstrate the performance of Kirk and make comparisons with the centralized Kirk executive. We also demonstrate the real-world applicability of our solution by using D-Kirk to execute the manipulator tool delivery example plan in our manipulator testbed. Finally, we present conclusions and discuss directions for future work.

# Chapter 2

# Robust Execution

## 2.1 Introduction

This chapter serves as a review of methods used for robust execution of contingent temporally flexible plans in the centralized case. We first define a temporally flexible plan and its representation as a Simple Temporal Network (STN). We then review dispatchable execution, which provides a means to robustly execute a temporally flexible plan. We then present a definition of a Temporal Plan Network (TPN), which adds choices to a temporally flexible plan. This allows us to describe contingent plans, where each contingency is a choice between functionally equivalent threads of execution. We then review methods for selecting a temporally consistent plan from the TPN. Finally, we present related work on robust plan execution and distributed execution systems.

## 2.2 Temporally Flexible Plans

Traditional plans specify a single duration for each activity in the plan. A temporally flexible plan allows us to specify a range of durations, in terms of a lower and upper bound on the permissible duration. This allows us to model activities whose durations may be varied by the executive, or whose durations are unknown and are controlled by the environment.

A temporally flexible plan is built from a set of primitive activities and is defined recursively using `parallel` and `sequence` operators, taken from the Reactive Model-based Programming Language (RMPL) [32]. A temporally flexible plan encodes all executions of a concurrent, timed program, comprised of these operators.

A primitive element of a temporally flexible plan is of the form `activity`$[l, u]$. This specifies an executable command whose duration is bounded by a simple temporal constraint. A simple temporal constraint $[l, u]$ places a bound $t^+ - t^- \in [l, u]$ on the duration between the start time $t^-$ and end time $t^+$ of the `activity`, or more generally, any sub-plan to which it is applied.

Primitive activities are composed as follows.

- `parallel`$(\mathit{subnetwork}_1, \ldots, \mathit{subnetwork}_N)$ $[l, u]$ introduces multiple subnetworks to be executed concurrently.

- `sequence`$(\mathit{subnetwork}_1, \ldots, \mathit{subnetwork}_N)$ $[l, u]$ introduces multiple subnetworks which are to be executed sequentially.

Note that the RMPL language also allows us to apply a simple temporal constraint between the start and end of a `parallel` or `sequence` operator, but this is purely syntactic sugar. In the case of the `parallel` operator, it corresponds to an additional parallel branch, consisting of a single `activity` with the prescribed simple temporal constraint. In the case of the `sequence` operator, it corresponds to an additional `parallel` operator, of which one branch is the `sequence` and the other is a single `activity` with the prescribed simple temporal constraint. For simplicity, we do not use these syntactic mechanisms in this thesis.

The network of simple temporal constraints in a temporally flexible plan form a Simple Temporal Network (STN). An STN is visualized as a directed graph, where the nodes of the graph represent events and directed edges represent simple temporal constraints. Graph representations of the `activity` primitive and of the `parallel` and `sequence` constructs are shown in Fig. 2-1.

Fig. 2-2 shows a simplified version of the selected plan from the manipulator tool delivery scenario used in Chapter 1 as an example of a temporally flexible plan. The

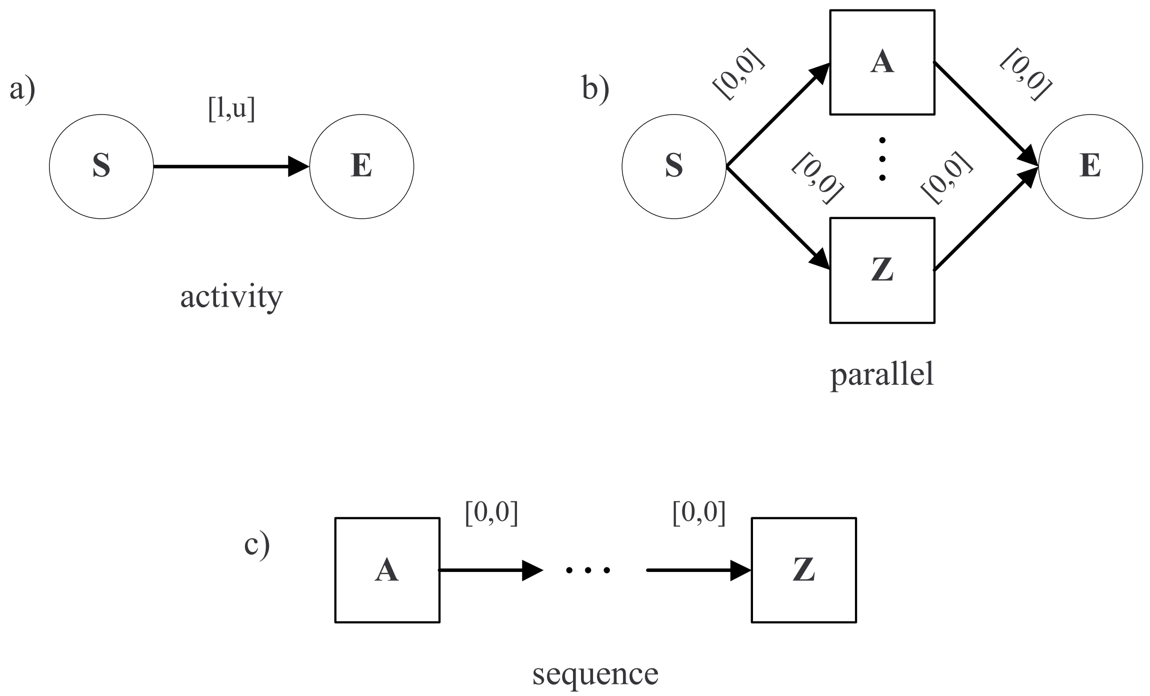


Figure 2-1: This figure shows graph representations of the `activity` primitive and the RMPL `parallel` and `sequence` constructs used to build a temporally flexible plan. (a) `activity`, (b) `parallel` and (c) `sequence`

first section of the plan has been collapsed to a single simple temporal constraint. The plan contains both the `sequence` and `parallel` constructs. For example, at the highest level, the plan consists of two subnetworks in parallel; a subnetwork between nodes 2 and 50 and another between nodes 68 and 69. The upper subnetwork, between nodes 3 and 39, is a series subnetwork, consisting of three activities (nodes 3 and 27, 28 and 29, and 30 and 31) followed by a `parallel` subnetwork (nodes 32 to 39). Note that nodes 2 and 50 are present as the remnants of the `choose` subnetwork from the complete TPN, which is introduced in Section 2.4.

In order for a temporally flexible plan to be executable, it must be *temporally consistent*. We discuss the methods used to test for temporal consistency in Section 2.5.

**Definition 1** *A temporally flexible plan is* ***temporally consistent*** *if there exists a* ***feasible schedule*** *for the execution of each time-point in the plan.*

**Definition 2** *A* ***feasible schedule*** *for the execution of a temporally flexible plan is*

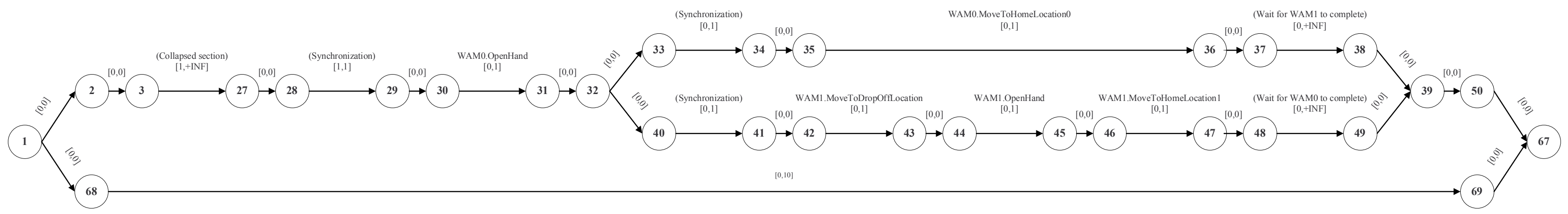

(Collapsed section) [1,+INF]
(Synchronization) [1,1]
WAM0.OpenHand [0,1]
(Synchronization) [0,1]
WAM0.MoveToHomeLocation0 [0,1]
(Wait for WAM1 to complete) [0,+INF]
(Synchronization) [0,1]
WAM1.MoveToDropOffLocation [0,1]
WAM1.OpenHand [0,1]
WAM1.MoveToHomeLocation1 [0,1]
(Wait for WAM0 to complete) [0,+INF]
[0,0]
[0,10]

Figure 2-2: This figure shows a simplified version of the plan from the manipulator tool delivery example used in Chapter 1 as an example of a temporally flexible plan.

*an assignment of execution times to each time-point in the plan such that all temporal constraints are satisfied.*

The example plan in Fig. 2-2 is temporally consistent because none of the temporal constraints conflict. This allows the executive to assign an execution time to each node without violating any temporal constraints. For example, consider node 32 in Fig. 2-2. The maximum duration specified by all of the edges between nodes 1 and 32 is +INF and the minimum duration is 2. Similarly, the maximum duration between nodes 32 and 67 is +INF and the minimum duration is 0. The simple temporal constraint between nodes 68 and 69 specifies a maximum duration of 10 between nodes 1 and 67 and a minimum duration of 0. This means that we can schedule node 32 to be executed at any time between times 1 and 10 and these constraints will be satisfied. All of the nodes in the plan can be scheduled in this way, so the plan is temporally consistent.

## 2.3 Dispatchable Execution

Given a temporally flexible plan to be executed, traditional execution schemes select a feasible execution schedule at planning time. This approach eliminates the temporal flexibility in the plan prior to execution, which leads to two problems. First, the fixed schedule lacks the flexibility needed to respond to variation in activity durations at execution time, so the plan is prone to failure. Second, if we generate a very conservative schedule to increase the likelihood of successful execution, then the execution becomes sub-optimal in terms of total execution time.

This limitation can be overcome through the methods of dispatchable execution [20], in which scheduling is postponed until dispatch time. This means that the plan retains temporal flexibility until it is dispatched. The dispatcher schedules events reactively, assigning execution times just-in-time using the observed execution times of previously executed events. This provides robustness to uncertain durations that lie within the temporally flexible bounds of the plan.

In traditional execution schemes, where events are scheduled at planning time, the job of the dispatcher is trivial: it simply executes events at the predetermined time. However, in the case of dispatchable execution, the dispatcher is required to perform a certain amount of computation to determine the execution time of each event. Since dispatching must be performed in real time, this extra computation can cause problems. To minimize the amount of computation that must be conducted in real-time, dispatchable execution uses a two-stage execution strategy. Prior to dispatching, the plan is compiled or *reformulated* to a form that allows straightforward dispatching. In particular, the plan is reformulated to a Minimal Dispatchable Graph (MDG), which requires the minimum amount of processing at dispatch time. This is followed by dispatching, when the plan is executed.

If a graph is dispatchable, then one can generate a feasible schedule using only *local* propagation of timing information to immediately succeeding activities. A *minimal* dispatchable graph is a dispatchable graph that contains the minimum possible number of edges. Dispatching a minimal dispatchable graph minimizes the amount of computation that must be performed by the dispatcher.

To form the MDG, reformulation identifies the *non-dominated edges* in the plan. These are the edges along which timing information must be propagated at dispatch time. The reformulation algorithm operates on the distance graph representation of the STN from the input temporally flexible plan.

Every STN has an equivalent representation as a distance graph [5]. The distance graph contains the same set of nodes as the STN. However, each edge represents a single temporal constraint, rather than the pair of constraints specified by a simple temporal constraint. Specifically, an edge from node A to node B with length or weight $b(A, B)$ specifies an upper bound of $b(A, B)$ on the execution time of node $B$ relative to node $A$, that is $t_B - t_A \leq b(A, B)$, where $t_A$ and $t_B$ are the execution times of nodes A and B respectively. Therefore, each edge in the STN is converted to a pair of opposed edges in the distance graph; one representing the lower and the other the upper bound of the simple temporal constraint. The distance graph representation of the STN from the example temporally flexible plan introduced in Section 2.2 is shown in Fig. 2-3.

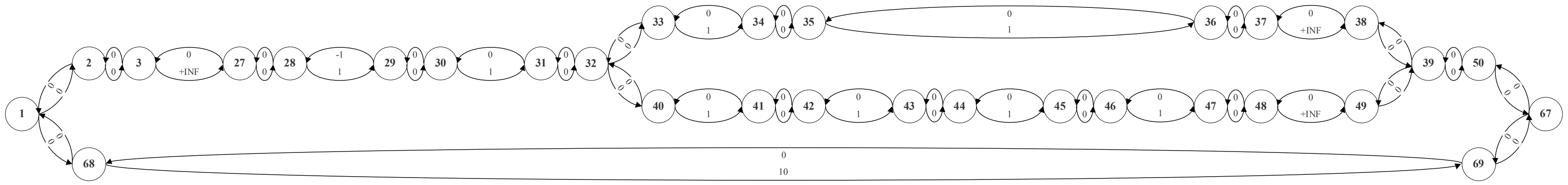


Figure 2-3: This figure shows the distance graph representation of the simple temporal network from the example temporally flexible plan introduced in Section 2.2.

For example, the simple temporal constraint of $[0, 1]$ between nodes 30 and 31 is represented by two edges. The lower bound of 0 is represented by an edge with weight 0 from node 31 to node 30. The upper bound of 1 is represented by an edge with weight 1 from node 30 to node 31.

The most simple reformulation algorithm [20] begins by forming the All Pairs Shortest Path (APSP) graph. The APSP graph is a distance graph that contains an edge from every node in the plan to every other node, the weight of which is equal to the shortest path distance between these nodes. We can not easily visualize the entire APSP graph for the example temporally flexible plan introduced in Section 2.2. Instead we show the APSP graph for a portion of the plan, as shown in Fig. 2-4.

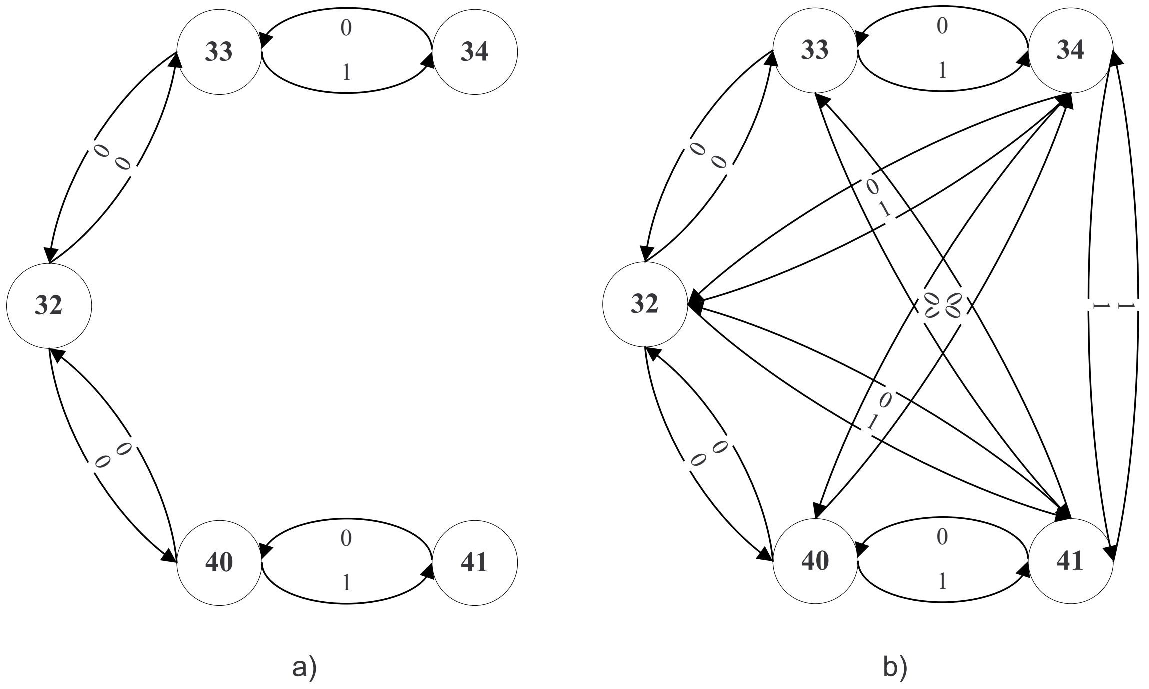


Figure 2-4: This figure shows (a) a portion of the distance graph for the example temporally flexible plan introduced in Section 2.2 and (b) its corresponding All Pairs Shortest Path graph.

The APSP graph is dispatchable but it is not minimal. Every time a node is executed the dispatcher must propagate timing information to all other nodes in the plan. This is computationally expensive and may prohibit real-time execution. The next step in the simple reformulation algorithm, therefore, is to trim the redundant edges from the APSP graph, leaving only those edges which belong in the MDG.

An edge is redundant if, in all possible execution schedules of the plan, there exists another edge which propagates a bound on the execution time of the node at the end of the edge which is at least as restrictive as that propagated by the edge in question. Such an edge is said to be *dominated* by the edge imposing the more restrictive bound. To determine if an edge is dominated we consider a triangle of nodes taken from the APSP graph. Since we consider the APSP graph, the edge between any two nodes A and B is a shortest path distance, written $|AB|$. There are two cases to consider, shown in Fig. 2-5.

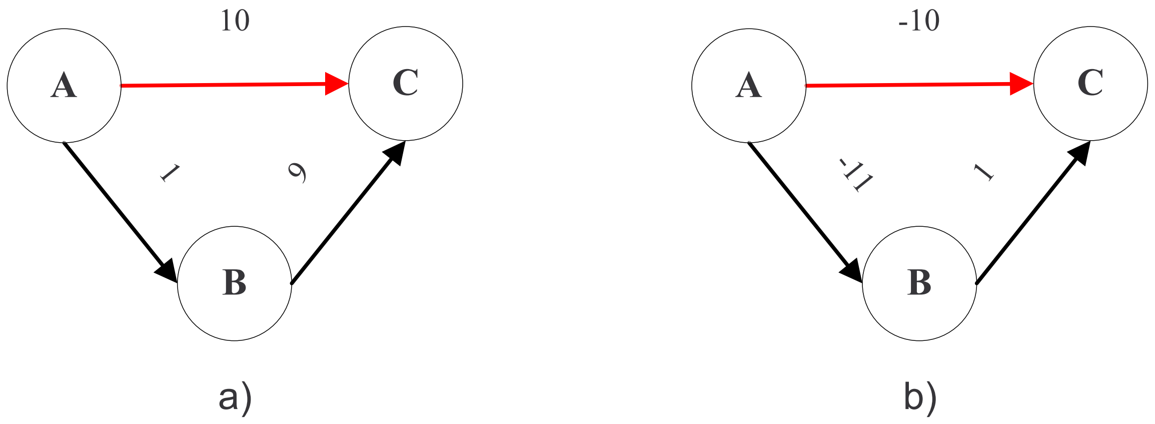


Figure 2-5: This figure shows an example of (a) an upper-dominated non-negative edge $|AB|$ and (b) a lower-dominated negative edge $|AB|$.

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

The simple reformulation algorithm is inefficient, because forming the APSP graph and searching it for non-dominated edges is expensive. Constructing the APSP graph requires $O(N^2)$ time complexity and $O(N^2)$ space complexity, where $N$ is the number of nodes in the plan. Searching the APSP graph for non-dominated edges requires

$O(N^3)$ time complexity. This gives an overall complexity for the simple reformulation algorithm of $O(N^3)$ in time and $O(N^2)$ in space.

The inefficiency of the simple reformulation algorithm is overcome by *fast reformulation* [27]. Fast reformulation identifies the non-dominated edges through a series of traversals of the graph, without forming the complete APSP graph. In the worst case, fast reformulation conducts a traversal from every node in the distance graph. This allows the non-dominated edges to be identified by a local test conducted at each node. The use of a local test is the source of the efficiency of the fast reformulation algorithm. In a traversal starting from node $A$, the fast reformulation algorithm applies a local test at node $B$ to determine whether the MDG should contain the shortest path edge $AB$.

We use the fast reformulation algorithm as the basis for the distributed plan reformulation phase of D-Kirk. The fast reformulation algorithm has space complexity linear in $N$, the number of nodes in the plan, and time complexity $O\left(NE + N^2 \ln N\right)$, where E is the number of edges in the plan, which is typically roughly proportional to $N$.

The fast reformulation algorithm further improves run time by exploiting Rigid Components (RCs). A rigid component is a set of nodes whose execution times are fixed relative to each other. The fast reformulation algorithm represents each RC by a single node, thus reducing the effective value of $N$ and hence reducing complexity. Note that the treatment of RCs in this way is also required for the dominance tests to function correctly. Note also that nodes whose execution times are not rigidly fixed relative to any other nodes in the plan can be considered as a special case of an RC, where the RC consists of a single node.

For example, consider the example plan introduced in Section 2.2. The plan is shown in Fig. 2-6. The edges between nodes 27 and 28 require them to execute at the same time. Similarly the edges between nodes 29 and 30 require them to execute at the same time. Furthermore, the edges between nodes 28 and 29 require node 29 to execute exactly 1 time unit after node 28. Therefore, the execution times of these four nodes are fixed relative to each other and they form a rigid component.

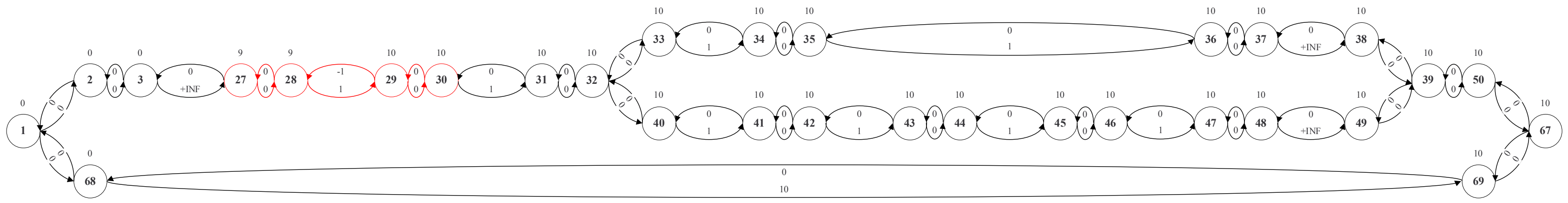


Figure 2-6: This figure shows the example plan with the RC consisting of nodes 27, 28, 29 and 30 highlighted in red.

The single node used to represent in the RC in the fast reformulation algorithm is referred to as the RC leader. The RC leader is the only member node involved in reformulation, so it is the only member of the RC which has MDG edges to other nodes in the plan. Therefore, during dispatching, it is the only member of the RC to which timing information is propagated from other nodes in the plan. Only once timing information has reached the RC leader can it be further propagated to the other members of the RC. For this reason, the RC leader must be the first member of the RC to be executed, so we choose as the RC leader the member node with the minimum Single Source Shortest Path (SSSP) distance from the start node of the plan. For example, consider the RC consisting of nodes 27, 28, 29 and 30 in Fig. 2-6. The nodes are labeled with the SSSP distance from the start node, node 1. Nodes 27 and 28 share the minimum SSSP distance of the RC members, so either could be selected as the RC leader.

The member nodes of an RC are connected with a doubly linked chain of edges. The nodes are linked in order of increasing SSSP distance from the start node, so that timing information can be propagated along the chain at dispatch time in order of execution time. A doubly linked chain of edges of this form is both minimal and dispatchable, so forms the MDG for the RC. This ensures that the RC members can be executed correctly at dispatch time.

A special case exists when multiple nodes in an RC are constrained to be executed at exactly the same time. These nodes are identified by the fact that they share the same SSSP value from the start node. A set of such nodes is referred to as a Zero-Related (ZR) component. It is required for the correct operation of the dominance tests used in the fast reformulation algorithm that ZR components are collapsed to a single node. It does not matter to which member of the ZR component the nodes are collapsed. Collapsing involves transferring all activity start and end events from the member nodes to the single surviving node. All edges to other nodes are also transferred. The example temporally flexible plan used earlier in this section is shown in Fig. 2-7. In this figure the ZR components of the RC containing nodes 27, 28, 29 and 30 have been collapsed. In particular, nodes 27 and 28 have been collapsed to node 28, and nodes 29 and 30 have been collapsed to node 30.

The RC leader represents all members of the RC for the purposes of reformulation. It is necessary, therefore, to express all temporal constraints between the member nodes of the RC and other nodes in the plan in terms of temporal constraints between the RC leader and the other nodes in the plan. To achieve this, the edges to all RC members are relocated to the RC leader. When the edges are rerouted their lengths must be modified to reflect the relocation. The amount by which edge lengths are adjusted is equal to the SSSP distance between the RC member from which they are being relocated and the RC leader. In particular, when relocating an outgoing edge from node A to a leader node L, its length must be increased by the SSSP distance of node A relative to that of node L. When relocating an incoming edge from node A to a leader node L, its length must be decreased by the SSSP distance of node A relative to that of node L.

For example, the temporally flexible plan used earlier in this section is shown in Fig. 2-8. In this figure the RC containing nodes 27, 28, 29 and 30 has been fully processed. The RC leader (node 28) has been selected, ZR nodes (nodes 27 and 28 and nodes 29 and 30) have been collapsed, the remaining RC members are joined by a doubly linked chain of edges and edges to other nodes in the plan graph have been relocated to the RC leader. In particular, the pair of edges between nodes 30 and 31 have been relocated to connect nodes 28 and 31. The outgoing edge from node 30 of length 1 has been increased in length by 1 unit to give a new length of 2. The incoming edge from node 30 of length 0 has been decreased in length by 1 unit to give a new length of -1.

Once the rigid components in the plan have been processed as described above, the fast reformulation algorithm performs a series of traversals of the graph to determine which edges should be included in the MDG. A traversal is conducted from every RC leader node (including the leaders of singleton RCs) in the graph. Each traversal follows the Reverse Post Order (RPO) for a Depth First Search (DFS) of the predecessor graph rooted at the start node of the traversal. The predecessor graph and the reverse post order, as well as details on how they are calculated, are

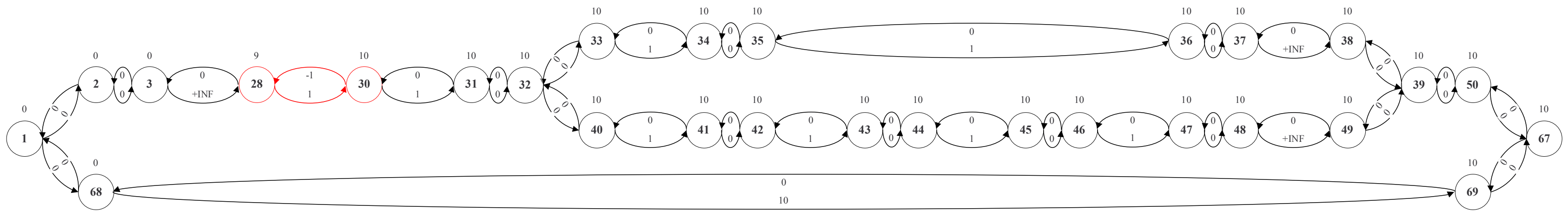


Figure 2-7: This figure shows the example plan with zero-related components collapsed in the RC consisting of nodes 27, 28, 29 and 30.

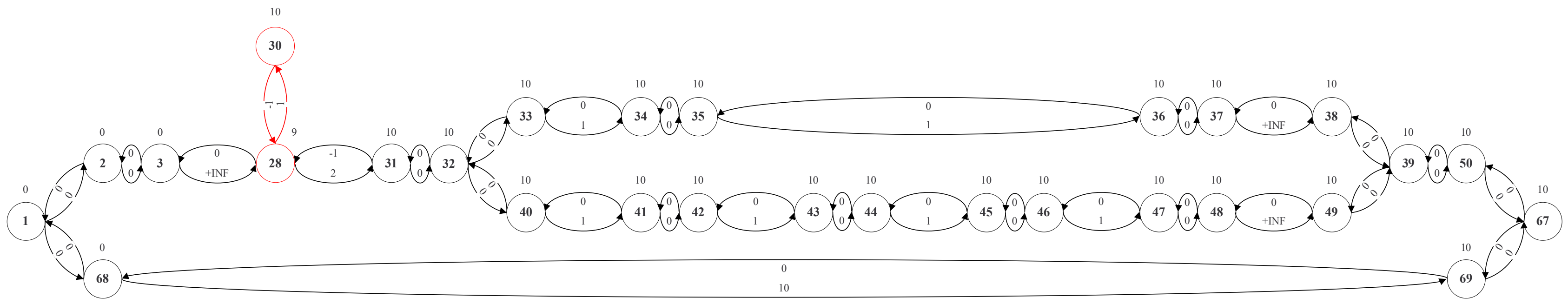


Figure 2-8: This figure shows the example plan with the RC consisting of nodes 27, 28, 29 and 30 processed.

given in Chapter 4. The predecessor graph is built on the set of RC leader nodes. At a given node B on the traversal started from node A, the fast reformulation algorithm performs a local test to determine whether or not the shortest path edge AB is dominated. Only non-dominated edges feature in the MDG. The test requires the following two pieces of data.

- ***minimum*** The minimum SSSP distance (from the start node of the traversal) of the nodes visited so far in this traversal, excluding the start and current nodes.

- ***non-positive*** Whether or not a node with a non-positive SSSP distance (from the start node of the traversal) has been encountered so far in this traversal, excluding the start and current nodes.[1]

The first dominance test is carried out at the second node in the RPO, as a dominance test at the first node in the RPO (the start node) is meaningless. It is for this reason that the start node is excluded in the above definitions.

Note that not all nodes are necessarily reachable from every other node in the plan, so the predecessor graph for a particular node may not span the entire set of nodes. As a result, a given traversal may not visit all of the RC leader nodes in the plan. This behavior is correct, because the shortest path edge from a given node to another node not in its predecessor graph must be of infinite length, so is always dominated and can not be a member of the MDG.

These two pieces of data are used as follows to perform the dominance test. The logic behind these tests is given by Tsamardinos et. al. [27].

- If $|AB|$ is non-negative, then the edge AB is upper-dominated if and only if *minimum* is less than the SSSP distance of B.[2]

[1]Tsamardinos et. al. [27] test for a negative edge, rather than a non-positive edge. Use of a non-positive edge allows correct dispatching in the case of nodes separated by simple temporal constraints with a lower bound of zero.

[2]Tsamardinos et. al. [27] use the condition 'less than or equal to', rather than 'less than'. Use of the condition 'less than or equal to' allows correct dispatching in the case of nodes separated by simple temporal constraints with a lower bound of zero.

- If $|AB|$ is negative, then the edge AB is lower-dominated if and only if *non-positive* is true.

For example, consider once again the example temporally flexible plan introduced in Section 2.2. Fig. 2-9 shows the distance graph for this plan once all rigid components have been processed. The figure represents a snapshot during the traversal from node 1. Each node is labeled with the SSSP distance from node 1 and the reverse post order is shown by the blue arrows. The node being traversed in this snapshot is node 42. This node has SSSP distance 10, so the shortest path edge from node 1 to node 42 is 10, which is non-negative, so we use the test for upper-dominance. The value of *minimum* is 10, which is not less than the SSSP distance of node 42, so the edge from node 1 to node 42 is not upper-dominated. We therefore include the APSP edge from node 1 to node 42, of length 10, shown in red in Fig. 2-9, in the MDG.

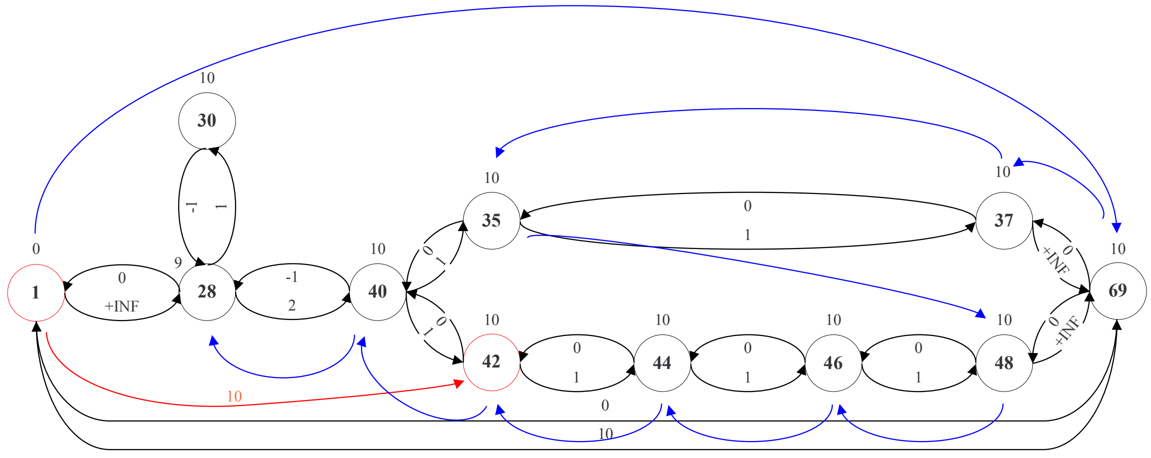


Figure 2-9: This figure shows a snapshot of the dominance test traversal from node 1. The node being traversed is node 42. The dominance test applied at this node determines that the APSP edge of length 10 from node 1 to node 42 is a member of the MDG.

Once a traversal has been conducted from every RC leader in the graph, and the non-dominated edges have been recorded, the MDG is complete. The MDG makes explicit all of the temporal constraints in the plan, such that it can be executed using only local propagation of timing information. The MDG can then be dispatched using the methods of Muscettola et. al. [20].

Recall that we use temporally flexible plans to model activities of uncertain duration. These uncertain durations arise for one of two reasons. First, it may be acceptable to conduct the activity for any period of time within a given range, or second, it may be the case that the duration of the activity is beyond the control of the executive. In the first case, the activity is termed *controllable* and its duration is dictated by the executive. In the second case, the activity is termed *uncontrollable* and its duration is dictated by nature. The fast reformulation algorithm does not guarantee that the MDG it produces is dispatchable if any of the activities in the plan are uncontrollable. This is because the dispatcher can not adapt the execution schedule to account for the uncertain completion time of activities whose durations are determined by nature.

Vidal and Ghallab introduced the notion of uncontrollable activities [29] and proposed methods to correctly execute plans containing such activities. A Simple Temporal Problem under Uncertainty (STPU) allows an uncontrollable activity to be distinguished from a controllable activity, so that it can be handled correctly. Vidal and Ghallab also introduce the notion of *controllability*; the ability for the executive to select a temporally consistent execution schedule irrespective of the *situation* imposed by the environment. In particular, they introduce *strong controllability*, where a single execution schedule is consistent for any situation; *weak controllability*, where a consistent execution schedule exists for each complete situation; and *dynamic controllability*, where the executive can generate a consistent execution schedule dynamically, given the observed situation up the current to point in time.

Work by Vidal and Fargier produced algorithms to check weak and strong controllability [28], while a dynamic controllability algorithm [18] was produced by Morris et. al.. The dynamic controllability algorithm extends the reformulation algorithm provided by Tsamardinos et. al. [27] to ensure correct dispatching of a plan containing uncontrollable activities. The algorithm tightens temporal constraints elsewhere in the plan to ensure that the plan can be correctly dispatched irrespective of the duration of the uncontrollable activities.

Prior to the dynamic controllability algorithm, Morris and Muscettola presented ways to apply existing plan verification, reformulation and dispatching algorithms to a STPU [17]. This work also applied the concept of an STPU to an STN, to define the Simple Temporal Network with Uncertainty (STNU).

Work by Stedl [25] introduced the Temporal Plan Network under Uncertainty (TPNU). This is a TPN which includes uncontrollable activities, where the temporal constraints of the TPNU are described by an STNU, rather than by an STN. Stedl produced a plan execution system which allows a group of agents to robustly execute a plan encoded as a TPNU when communication is limited at execution time. To achieve this, agents are clustered into groups based on their communication availability. A novel hierarchical reformulation algorithm is then used to rearrange the plan into two layers. At the group level, group plans are decoupled so that a static schedule can be used that does not require communication at execution time. At the agent level, each group plan is reformulated using a new fast version of the dynamic controllability algorithm to allow agents to respond to the uncertain durations of uncontrollable activities.

## 2.4 Temporal Plan Networks

Dispatchable execution provides robustness to execution uncertainty by preserving temporally flexibility throughout the planning stage and exploiting it at dispatching time. This allows the executive to use the temporal flexibility in the plan to respond to uncertain activity durations at dispatch time. A Temporal Plan Network [12] (TPN) extends a temporally flexible plan by adding contingencies. These contingencies encode functionally equivalent methods for achieving the same task. This provides the executive with alternate methods of achieving a goal, and hence adds robustness to plan failure.

For example, in the manipulator tool delivery task introduced in Chapter 1 we must deliver a tool to a specified location within a certain time limit. In this example, the times at which a tool will become in each of two initial locations is not known *a priori*. We wish to encode in our plan two alternative methods for delivering the tool to the desired location. The two methods represent the two different strategies that are required to deliver the tool within the specified time limit, depending on the time at which the tool becomes available in each of the two initial locations. The inclusion in the plan of both contingencies allows the executive to successfully execute the task even if one alternative method fails due to insufficient time being available for delivering the tool from that initial location.

A TPN encodes contingencies by augmenting the temporally flexible plan representation with a `choose` operator, taken from the RMPL programming language [32]. The `choose` operator allows us to specify nested choices in the plan, where each choice is an alternative sub-plan that performs the same function. The `choose` operator is defined below.

- `choose`$(TPN_1, \ldots, TPN_N)$ introduces multiple sub-plans of which only one is to be chosen. A choice variable is used to encode the currently selected subnetwork.

As with the other RMPL constructs, the RMPL language allows a simple temporal constraint to be applied between the start and end of the `choose` operator, as syntactic sugar. Here this corresponds to an additional `parallel` operator, of which one branch is the choice and the other is a single `activity` with the prescribed simple temporal constraint. For simplicity, we do not use these syntactic mechanisms in this thesis.

The RMPL representation of the manipulator tool delivery task is shown in Fig. 2-10. A graph representation of the `choose` construct is shown in Fig. 2-11. The choice node is shown as an inscribed circle.

An example of a TPN is that representing the manipulator tool delivery scenario, shown in Fig. 1-2 and repeated in Fig. 2-12 below. This example includes all three TPN constructs; `parallel`, `sequence` and `choose`. Node 2 is a choice node and introduces the two possible methods for delivering the tool. The upper path through the TPN involves WAM0 picking up the tool and passing it to WAM1 for delivery. The lower path uses WAM1 to pick up and deliver the tool alone.

```
parallel
  choose
    sequence
      parallel
        sequence
          parallel
            (Tool delivery) [x,+INF]
            sequence
              (Synchronization) [0,1]
              WAM0.MoveToPickupLocation0 [0,1]
              (Wait for tool) [0,+INF]
            end-sequence
          end-parallel
          WAM0.CloseHand [0,1]
          WAM0.MoveToHandOffLocation [0,1]
          (Wait for WAM1 to complete) [0,+INF]
        end-sequence
        sequence
          (Synchronization) [0,1]
          WAM1.MoveToHandOffLocation [0,1]
          (Wait for WAM0 to complete) [0,+INF]
        end-sequence
      end-parallel
      WAM1.CloseHand [0,1]
      (Synchronization) [1,1]
      WAM0.OpenHand [0,1]
      parallel
        sequence
          (Synchronization) [0,1]
          WAM0.MoveToHomeLocation0 [0,1]
          (Wait for WAM1 to complete) [0,+INF]
        end-sequence
        sequence
          (Synchronization) [0,1]
          WAM1.MoveToDropOffLocation [0,1]
          WAM1.OpenHand [0,1]
          WAM1.MoveToHomeLocation1 [0,1]
          (Wait for WAM0 to complete) [0,+INF]
        end-sequence
      end-parallel
    end-sequence
    sequence
      parallel
        sequence
          WAM1.MoveToPickupLocation1 [0,1]
          (Wait for tool) [0,+INF]
        end-sequence
        (Tool delivery) [y,+INF]
      end-parallel
      WAM1.CloseHand [0,1]
      WAM1.MoveToDropOffLocation [0,1]
      WAM1.OpenHand [0,1]
      WAM1.MoveToHomeLocation1 [0,1]
    end-sequence
  end-choose
  (Available time) [0,10]
end-parallel
```

Figure 2-10: This figure shows the RMPL representation of the manipulator tool delivery task.

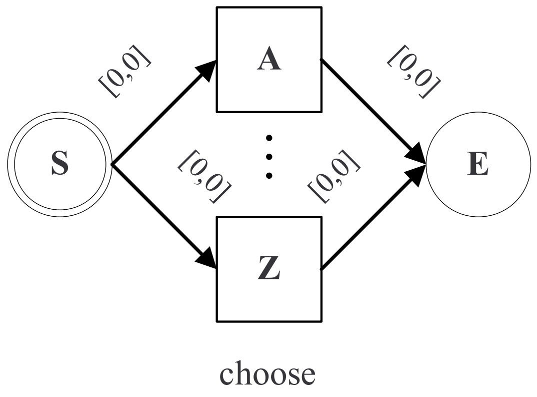


Figure 2-11: This figure shows the graph representation of the RMPL `choose` construct used to build a Temporal Plan Network. The choice node is shown as an inscribed circle.

## 2.5 Plan Extraction

A TPN allows us to encode alternative options in a plan. At execution time, the executive must select from the TPN a plan for which all temporal constraints are satisfied. Recall that for each set of alternative contingencies in the TPN, the currently selected contingency is encoded in the corresponding choice variable . Selecting a plan, therefore, is a matter of assigning values to the TPN's choice variables. In particular, we require a *feasible* plan. Note that a temporally consistent plan was defined in Section 2.2.

**Definition 3** *A **feasible plan** is a* complete choice assignment *to the choice variables of a TPN such that the selected plan is* temporally consistent*.*

**Definition 4** *A **complete choice assignment** is an assignment to the choice variables of a TPN such that 1) all active choice variables are assigned 2) all inactive choice variables are unassigned. The plan denoted by a complete choice assignment includes all activities and temporal constraints in the selected portion of the TPN.*

**Definition 5** *An **active choice variable** is a choice variable whose corresponding choice node is included in the currently selected portion of the TPN.*

In general, plan selection consists of two stages: generation of candidate plans and testing of candidate plans for temporal consistency. We detect temporal inconsistency

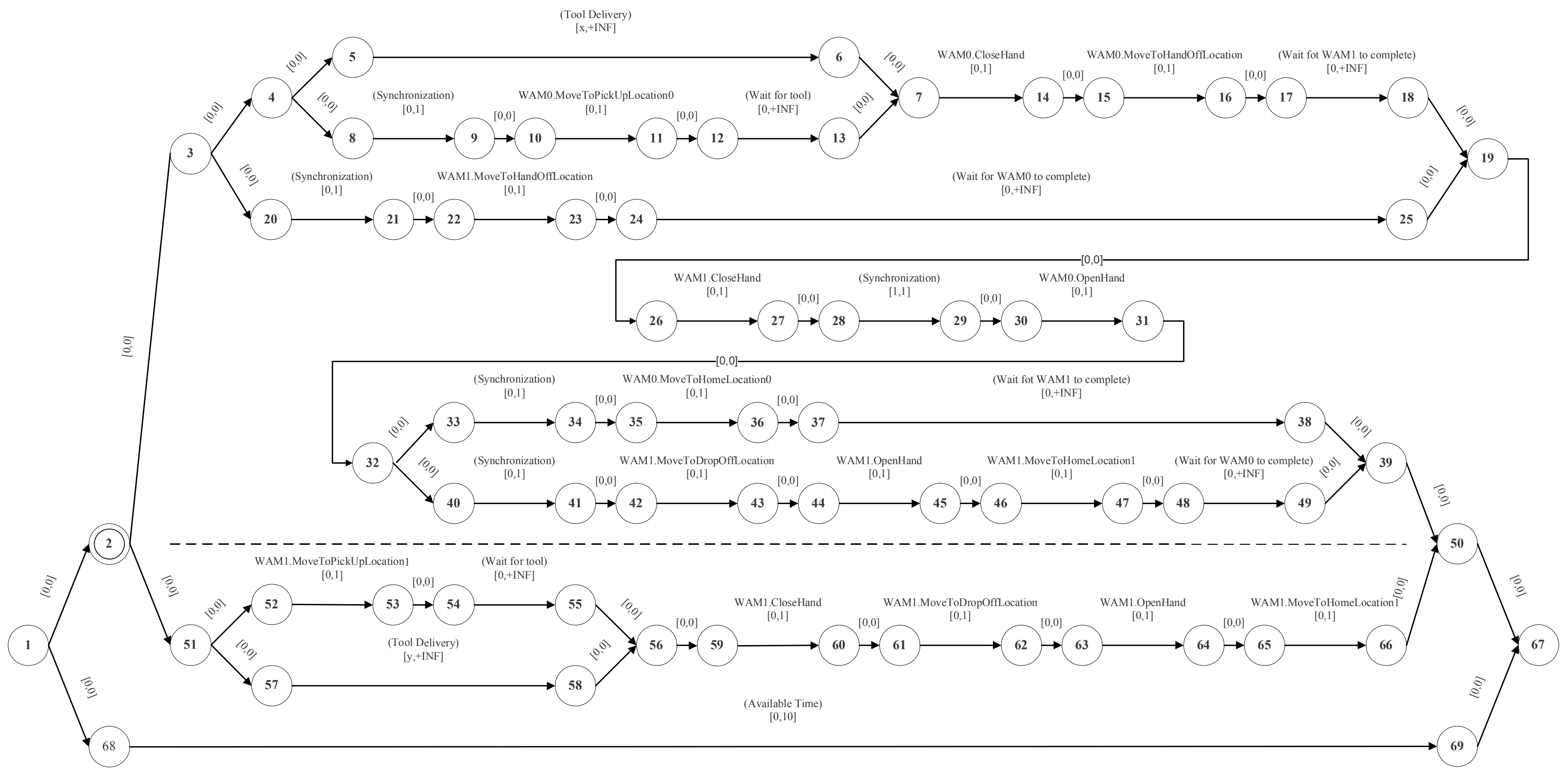


Figure 2-12: This figure shows the TPN representing the manipulator tool delivery scenario.

as a negative cycle in the distance graph. For example, consider the distance graph in Fig. 2-13. This plan is inconsistent because the upper temporal bound for the thread of execution ABD is 9 units, whereas the lower temporal bound for the thread of execution ACD is 10 units. Clearly these two constraints are incompatible, so no feasible assignment of execution times exists and the plan is temporally inconsistent. This inconsistency manifests itself as a negative cycle in the distance graph: the sum of the edges around the loop ABDCA is -1.

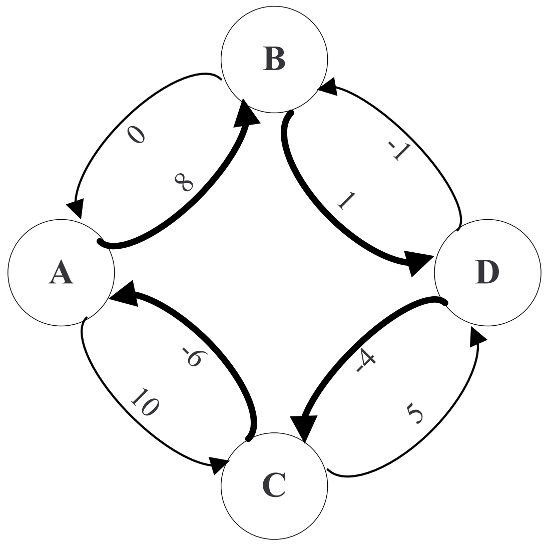


Figure 2-13: This figure shows an inconsistent plan. The distance graph contains a negative cycle ABDCA.

The simplest method to test for negative cycles is to use an All Pairs Shortest Path (APSP) algorithm, for example the Floyd-Warshall algorithm or the matrix-multiplication-based APSP algorithm. The Floyd-Warshall algorithm takes $O(N^3)$ time and requires $O(N^2)$ space to store the distance matrix, and the matrix-multiplication-based APSP algorithm takes $O(N^3 \log N)$ time and requires $O(N^2)$ space as well. However, there are methods of detecting negative cycles that are both faster and require less space. The Bellman-Ford algorithm is used to compute single-source shortest paths but also can be used to check for negative cycle in $O(NE)$ time, where E is the number of edges in the distance graph. In addition, this algorithm only needs to maintain one distance label at each node, which only takes $O(N)$ space.

The *Kirk* executive [12] generates candidate plans by performing a Depth First Search (DFS) through the possible assignments to the choice variables. Plan extraction therefore has exponential time complexity in the worst case. Kirk frames consistency checking as a Single Source Shortest Path (SSSP) calculation. It uses the First In First Out (FIFO) implementation of the generic label correcting SSSP algorithm, which is similar to the Bellman-Ford SSSP algorithm.

Work by Walcott [30] extends the plan selection phase of Kirk to handle both activity costs and the problem of path planning. This planner uses A* search to extract an optimal plan from a TPN which is extended to include activity costs. In addition, the planner is integrated with a road-map based path planner.

Work by Effinger [7] extends the plan selection phase of Kirk to use dynamic backtracking to improve the efficiency of search. The planning problem is posed as a Constraint Satisfaction Problem (CSP).

Recent work by Wehowsky [31] produced a distributed version of the plan selection phase of Kirk. It uses a distributed depth-first search for candidate plan generation and the distributed Bellman-Ford algorithm for consistency checking. This is the plan selection algorithm used in D-Kirk.

## 2.6 Related Work

In this section we provide an overview of other work related to the execution of temporally flexible plans and to distributed algorithms in general.

Work by Tsamardinos introduced the Probabilistic Simple Temporal Problem (PSTP) [26]. In this work, uncontrollable events are modeled using random variables following conditional probability distributions. This overcomes two limitations of the STPU formalism. First, the duration of a real uncontrollable activity is not limited by lower and upper bounds, but there exists a finite probability of any duration. Second, there is no principled way for the modeler to select these bounds. Tsamardinos presents an algorithm which schedules controllable activities such that the probability that the temporal constraints in the plan are satisfied by the durations of the uncontrollable activities is maximized.

Work by Khatib et. al. introduced the Simple Temporal Problem with Preferences (STPP) [11]. STPPs allow us to model plans with temporal constraints and to express preference criteria which dictate the relative importance of the satisfaction of each constraint. Khatib et. al. present an algorithm, based on a simple generalization of the SSSP algorithm, for finding globally best solutions to STPPs with restricted preference functions.

Schwartz and Pollack developed the Disjunctive Temporal Partial-Order Planner (DT-POP) [24]. This work uses a partial-order planning approach to generate plans from actions which include disjunctions of temporal constraints.

Model-based programming elevates the level of interaction between human operators and hidden-state, under-actuated systems. A model-based executive reasons on a model of the plant to determine the required sequence of commands to drive the plant through the required states. Williams introduced a model-based executive called Titan [32], which allows the operator to specify the behavior to be executed in terms of intended plant state evolution, instead of specific command sequences.

Work by Léauté and Williams addressed the problem of plan execution for under-actuated systems [14]. Their model-based executive takes as input a qualitative state plan. This is a temporally flexible plan augmented with state constraints, which specify the intended state evolution of the system. The executive dynamically generates a near-optimal control sequence by framing planning as a disjunctive linear programming problem. To achieve robustness to disturbances and tractability, planning is folded within a receding horizon, continuous planning framework.

Qualitative state plans are also used by Hofmann and Williams for the control of bipedal walking devices [10]. These bipedal walking devices are hybrid systems with high-dimensional non-linear dynamics, so a feedback linearizing controller is used to provide an abstraction of the system. The qualitative state plan specifies the input to the system as a set of gait poses, including foot placement constraints, linked by flexible timing constraints. The executive uses the temporal and state flexibility to provide robustness to disturbances and controls the system by tuning the control parameters of the linearized abstraction.

A great deal of work has been done on controlling groups of autonomous agents to perform collaborative tasks. Des Jardins [6] asserts that to effectively execute real world problems we require a planner that is both distributed and continual, where planning and execution are interleaved. This work presents a survey of work in this field and draws some general conclusions. In particular, distributed planning is broadly categorized into two distinct approaches. The first is Cooperative Distributed Planning (CDP), where a group of agents collaborate to formulate a plan which is optimal, or at least good, for the collective objectives of the group as a whole. Similarly, the agents execute the plan in a coherent, effective manner. The second approach is Negotiated Distributed Planning (NDP), where agents seek to construct a plan which is optimized for their individual objectives. Where other agents place constraints upon the planning process, an agent seeks to persuade them to accommodate its own preferences. Our executive takes a collaborative group plan and executes it in a distributed fashion, so fits the CDP approach.

The RoboCup Rescue competition [13] highlights the need for distributed execution systems to achieve complex, real-world tasks.

The CASPER system [8] provides a system for coordinating a group of rovers in collecting planetary surface data. A central planner generates an abstracted group plan and assigns activities to each rover. Each rover is then responsible for planning its own activities and provides a detailed, executable plan. However, this is a leader-follower type of distributed system and there is no communication between followers. This means that agents do not interact when formulating their individual plans, so the system can not handle plans with tight coupling between activities owned by different agents. Each agent executes its plan independently, within a re-planning architecture.

The Multi-Rover Integrated Science Understanding System (MISUS) [9] uses a similar approach.

The CAMPOUT [21] system provides an architecture for the tightly coupled coordination of groups of agents. The actions of each agent are determined by behaviors, which represent the mapping from a set of observations to the activity performed by the agent. A behavior has an assigned preference value which describes how desirable the action is. This system of preferential behaviors is extended to the case of multiple agents through multi-objective problems [22]. This allows agents to take into account the preferences of other agents and thus reach a mutually preferred plan.

Temporally flexible plans allow us to model tightly coupled sets of activities. None of the above work provides methods for the distributed execution of such tightly coupled plans. However, work by Riley and Veloso provide a system for generating and executing plans for a group of agents [23] which partially addresses this problem. First, planning is performed in a centralized fashion. The output of the planning phase is a STN. The centralized planner then decides which agent will conduct each activity and assigns to the agent the corresponding nodes of the STN. Each agent then executes its portion of the STN in a fully distributed fashion, without access to a global state. This work differs from our approach in that planning is centralized rather than distributed. Also, complete reformulation is not used: instead the APSP graph is formed from the STN and used in execution. Riley and Veloso state that for the relatively small plans they use, the APSP graph is manageable and full reformulation is not required. Furthermore, formation of the APSP graph is centralized also.

# Chapter 3

# Plan Distribution

## 3.1 Introduction

D-Kirk distributes both data and processing between all of the agents involved in the plan and this means that all agents must be able to communicate throughout the execution of the plan. This chapter addresses the problems of establishing the required communication routes between the agents involved in the plan and of distributing the plan data between those agents. First we discuss exactly how the plan is distributed and the requirements that this places upon the structure of the plan. We then introduce the algorithm we use to establish the required communication routes between agents.

## 3.2 Distribution

One of the advantages of using a group of agents, rather than a single agent, is that more complex tasks can be achieved. One reason for this is that agents can be specialized such that each lends its particular skills to a specific portion of the plan. As a result, each activity in a multi-agent plan is intended to be executed by a specific agent. For example, a rover equipped with a robotic manipulator is able to take a soil sample. A second rover, with a camera but no manipulator, can not take soil samples, but can capture images. These two rovers will therefore be used in distinct roles in an activity plan.

In a centralized architecture a single agent conducts all of the processing related to dispatching the plan. However, as discussed above, each activity must be executed by a specific agent. This means that when a given activity is to be executed, the master agent must send a message to the agent which will carry out that activity, instructing it to begin execution of the activity. In a plan involving many agents this can lead to a communication bottleneck at the master node, which renders the system vulnerable to communication latency. These delays may mean that the system can not operate in real-time, causing dispatching failure.

The main objective in distributing the dispatching process is to eliminate the communication bottleneck present in a centralized architecture. We achieve this by distributing both data and processing between all of the agents involved in the plan. In particular, we assign to each agent the portion of the plan corresponding to the activities it will carry out, as well as the responsibility for processing this part of the plan.

D-Kirk uses a Temporal Plan Network (TPN) to represent the plan to be executed, so the task of distributing the plan data between the available agents is one of assigning parts of the TPN to each agent. A TPN consists of nodes and edges, where nodes represent points in time and are connected by edges, which represent activities whose durations are bounded by simple temporal constraints. The dispatching process used to execute the activities in the plan involves assigning execution times to each node. For the purposes of dispatching, therefore, we represent each activity by two events; a start event at the activity's start node and an end event at its end node. Details of the dispatching algorithm are given in Chapter 5. Therefore, we assign nodes to agents based on the ownership of the activities corresponding to the start and end events they contain.

For example, consider the plan fragment shown in Fig. 3-1, which is based on the manipulator tool delivery scenario introduced in Chapter 1. The plan fragment contains three activities, two of which are to be carried out by WAM0 and third by WAM1. The figure shows how start and events are assigned to the relevant nodes for each activity and how nodes are then assigned to agents.

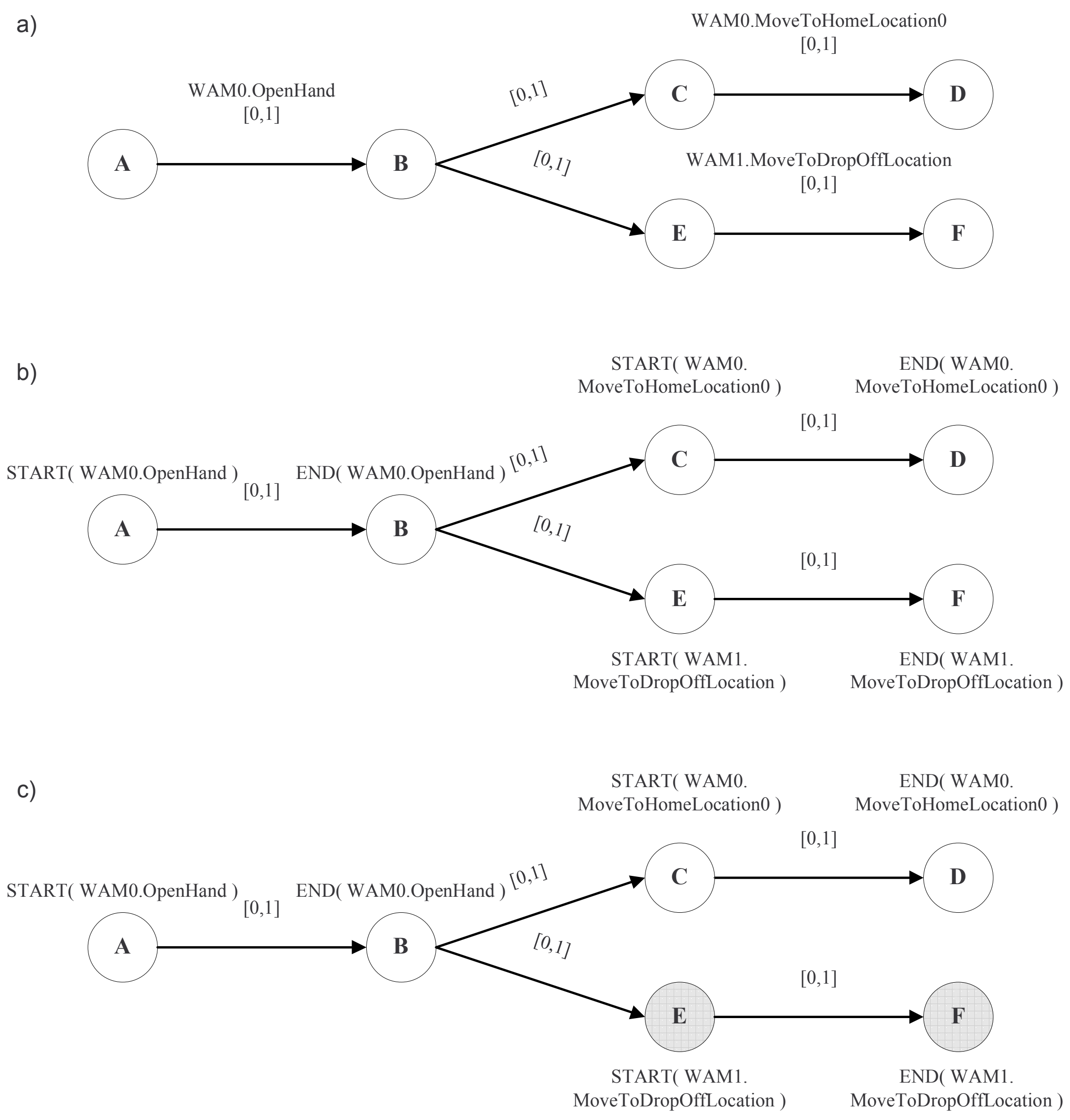

a)
WAM0.OpenHand
[0,1]
[0,1]
[0,1]
WAM0.MoveToHomeLocation0
[0,1]
WAM1.MoveToDropOffLocation
[0,1]
A
B
C
D
E
F
b)
START( WAM0.OpenHand )
END( WAM0.OpenHand )
START( WAM0. MoveToHomeLocation0 )
END( WAM0. MoveToHomeLocation0 )
START( WAM1. MoveToDropOffLocation )
END( WAM1. MoveToDropOffLocation )
c)

Figure 3-1: This figure shows an example plan fragment and how nodes are assigned to agents based on the activities they will perform. (a) TPN with activities. (b) Activity start and end events assigned to nodes. (c) Nodes assigned to agents based on activity start and end events. Nodes assigned to WAM1 are shown in gray.

It is possible to construct a TPN in which a given node contains multiple activity start or end events. If all of the events in a single node correspond to activities to be carried out by the same agent, the distribution proceeds as described above. However, the events may correspond to activities to be carried out by multiple agents. Such a plan requires multiple agents to execute events simultaneously. This is not possible in any realizable system, so execution of the plan is not possible and distribution fails. For example, consider the TPN fragment in Fig. 3-2. Node B owns multiple activity events but both correspond to activities to be carried out by WAM0, so the node is successfully assigned to WAM0. However, node C owns one activity event corresponding to WAM0 and another corresponding to WAM1. This means that the plan can not be executed and the distribution fails.

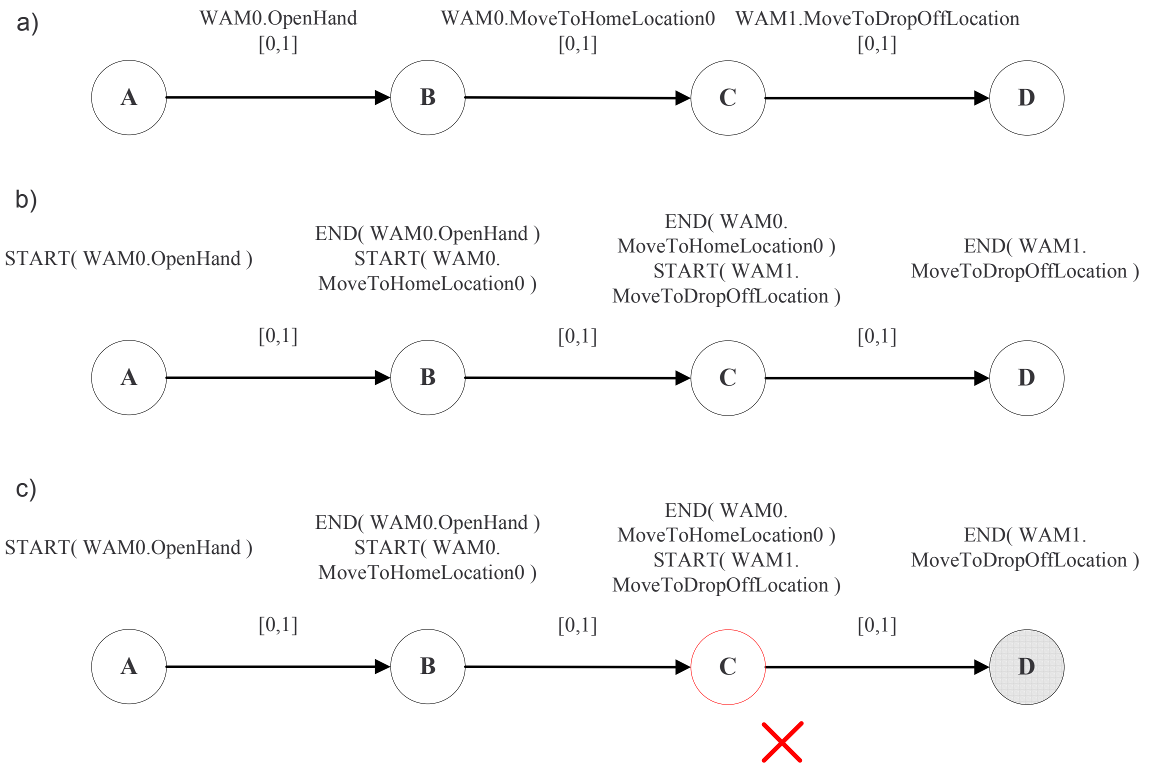

a)
WAM0.OpenHand [0,1]
WAM0.MoveToHomeLocation0 [0,1]
WAM1.MoveToDropOffLocation [0,1]
A
B
C
D
b)
START( WAM0.OpenHand )
END( WAM0.OpenHand ) START( WAM0. MoveToHomeLocation0 )
END( WAM0. MoveToHomeLocation0 ) START( WAM1. MoveToDropOffLocation )
END( WAM1. MoveToDropOffLocation )
[0,1]
[0,1]
[0,1]
A
B
C
D
c)
START( WAM0.OpenHand )
END( WAM0.OpenHand ) START( WAM0. MoveToHomeLocation0 )
END( WAM0. MoveToHomeLocation0 ) START( WAM1. MoveToDropOffLocation )
END( WAM1. MoveToDropOffLocation )
[0,1]
[0,1]
[0,1]
A
B
C
D

Figure 3-2: This figure shows an example plan fragment that can not be distributed because it requires events to be carried out simultaneously by multiple agents. (a) TPN with activities. (b) Activity start and end events assigned to nodes. (c) Nodes assigned to agents based on activity start and end events. Nodes assigned to WAM1 are shown in gray. Node C can not be assigned because it owns activity events corresponding to activities to be carried out by both WAM0 and WAM1.

For practical modeling purposes, this situation can be avoided by inserting a ‘wait’ duration between the activities to be executed by each agent. This removes the requirement for the events to be executed simultaneously and allows the plan to be executed and therefore distributed successfully. An example of this technique is the waits between nodes B and C and nodes B and E Fig. 3-1.

Section 2.3 introduced the concept of a Zero-Related (ZR) component. A ZR component is a group of plan nodes which are constrained to execute simultaneously and must be collapsed to a single node for the correct operation of the fast reformulation algorithm. When ZR nodes are collapsed, the resulting node takes all of the activity start and end events. In the case where member nodes of the ZR component own activity start and end events corresponding to multiple agents, this again leads to the situation described above, where multiple agents must execute an event simultaneously. To avoid this, we check for this condition after collapsing each ZR component and if it is detected the algorithm fails.

Fig. 3-3 shows an example plan in which two ZR components must be collapsed. First, nodes are assigned to agents based on their activity start and end events. The ZR component on the right is collapsed to node E, which owns activity events corresponding to activities to be executed by WAM0 only. However, the ZR component on the left is collapsed to node C, which owns activity events corresponding to activities to be executed by WAM0 and WAM1. This means that the plan can not be executed and ZR component collapsing fails.

When an agent is assigned a node it records the activity start and end events belonging to that node as well as information relating to the edges connected to the node. For each edge, the agent records the value of the simple temporal constraint it represents and the remote node to which it is connected. This means that agents at both ends of each edge have knowledge of the edge.

Once distribution is complete, each agent is responsible for conducting all processing relating to the plan nodes it owns. This means that each agent conducts planning, reformulation and dispatching for the plan fragment made up of the nodes it owns. These operations involve creating, removing and modifying the edges between nodes. Throughout the processing, agents send messages to each other to notify them of changes to the set of edges. This is the basis of the distributed approach to plan execution.

This section described the way in which a plan is distributed among the agents

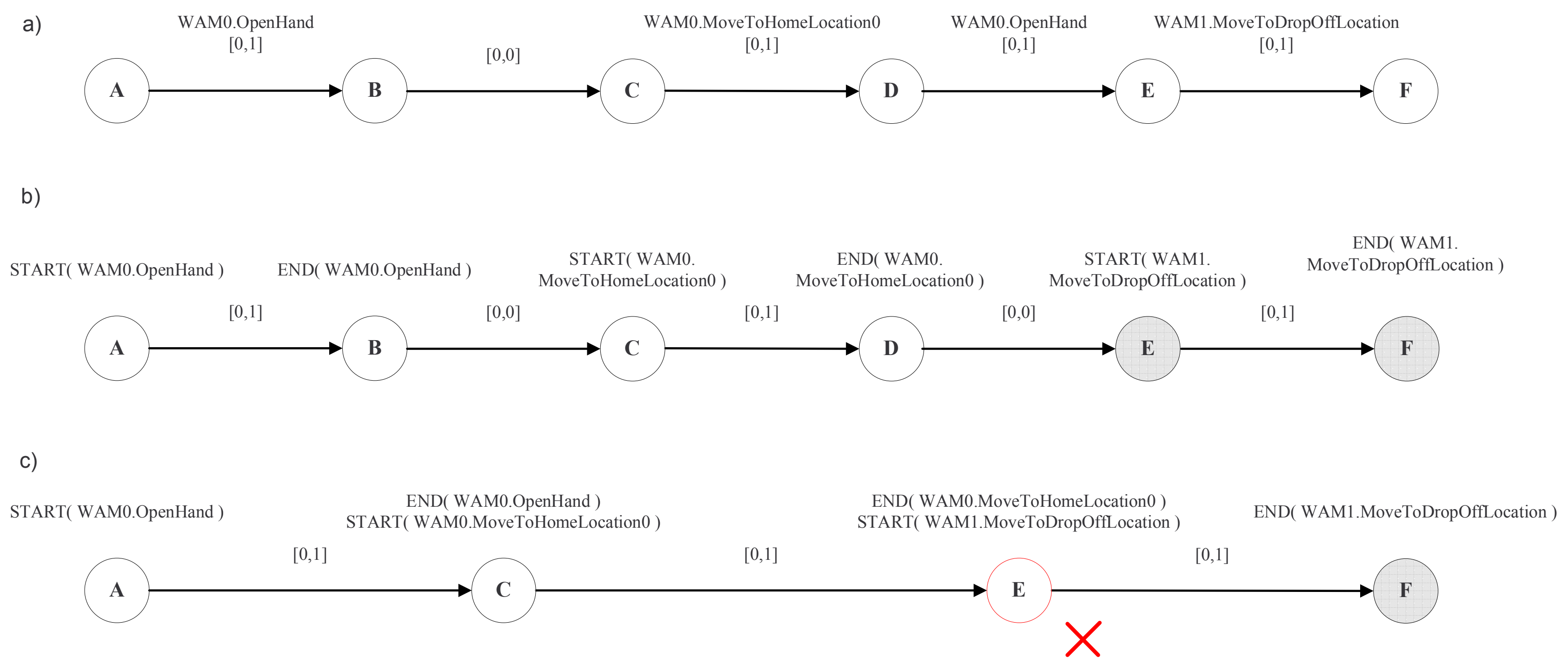

a)
WAM0.OpenHand
[0,1]
[0,0]
WAM0.MoveToHomeLocation0
[0,1]
WAM0.OpenHand
[0,1]
WAM1.MoveToDropOffLocation
[0,1]
A
B
C
D
E
F
b)
START( WAM0.OpenHand )
END( WAM0.OpenHand )
START( WAM0. MoveToHomeLocation0 )
END( WAM0. MoveToHomeLocation0 )
START( WAM1. MoveToDropOffLocation )
END( WAM1. MoveToDropOffLocation )
[0,1]
[0,0]
[0,1]
[0,0]
[0,1]
A
B
C
D
E
F
c)
START( WAM0.OpenHand )
END( WAM0.OpenHand ) START( WAM0.MoveToHomeLocation0 )
END( WAM0.MoveToHomeLocation0 ) START( WAM1.MoveToDropOffLocation )
END( WAM1.MoveToDropOffLocation )
[0,1]
[0,1]
[0,1]
A
C
E
F

Figure 3-3: This figure shows an example plan in which two ZR components must be collapsed. (a) Before node assignment. (b) After node assignment. Nodes assigned to WAM1 are shown in gray. (c) After collapsing. Node E owns activity events corresponding to activities to be executed by WAM0 and WAM1 so ZR component collapsing fails.

which will carry out its activities and discussed the limitations this places upon the structure of the plan.

## 3.3 Communication Availability

In general, a given agent is able to communicate directly with only a subset of the other agents involved in the plan. For example, a rover may use radio or infrared transmission for communicating with other agents and such a system has limited range. In the case of a group of rovers operating over a wide area, therefore, each rover can communicate only with neighbor rovers that are within a certain range.

D-Kirk requires all agents to be able to communicate with each other throughout the execution. This means that we must establish communication pathways between all agents. In particular, we need to establish which agents are able to communicate with each other directly and then use these communication pathways to provide communication between all agents. To achieve this, we arrange the available agents into an organized structure. We establish a hierarchical, ad-hoc communication network, using a leader election algorithm [2].

The first stage in the formation of the agent hierarchy is to form the agents into groups, based on the available communication between them. Each group is known as a club, and comprises a single leader agent and number of followers. By definition, a leader is available to communicate directly with all of its followers, but not with any other leaders.

The leader election algorithm used to form the clubs is shown as pseudo-code in Fig. 3-4 and line numbers in the following description refer to this figure. The leader election algorithm is run simultaneously by every agent and uses a countdown from a randomly generated number to arbitrarily select which agents are to become leaders. First, the agent assigns to its countdown variable $r$ a random integer between 1 and an upper bound $R$ (line 1). Details of how to best choose the value of $R$ are given by Coore et. al. [2]. The algorithm then performs a series of iterations, decrementing the value of $r$ at each iteration (line 3). At each iteration, the agent checks to see if $r$ has reached zero (line 4). If so, this agent declares itself a leader (line 5) and broadcasts a RECRUIT message to all agents within its communication range (line 6). When an agent receives a RECRUIT message (line 8) then it declares itself a follower (line 9) and records the identity of its leader (line 10). In this way, each follower is by definition in direct communication with its leader. If an agent does not receive the broadcast RECRUIT message it must not be in direct communication with the leader which broadcast it. Such an agent continues to listen for other RECRUIT messages, or if it does not receive any before its countdown variable $r$ reaches zero, it declares itself a leader. In this way, it is guaranteed that leaders are unable to communicate.

Once an agent has received a RECRUIT message and declared itself a follower it continues to listen for subsequent messages until $r$ reaches zero (line 2). If such a message is received, the agent records the sender of this message as a secondary leader. In this way an agent can be a member of multiple clubs. Such an agent can by definition communicate with the leaders of all of the clubs to which it belongs, so is used to route messages between these clubs. This defines the notion of neighboring clubs, which is used below.

Details of how the algorithm can be modified to handle the situation where two agents within communication range of each other broadcast RECRUIT messages simultaneously are given by Coore et. al. [2].

```
 1: $r \leftarrow$ random$(1, R)$
 2: while $r > 0$ do
 3:     $r \leftarrow r - 1$
 4:     if $r = 0$ then
 5:         $leader =$ true
 6:         broadcast RECRUIT
 7:     end if
 8:     if received message RECRUIT then
 9:         $follower =$ true
10:         record sender of message as leader
11:     end if
12: end while
```

Figure 3-4: This figure shows the pseudo-code for the leader election algorithm used to form agent clubs.

Fig. 3-5 shows the operation of the club formation algorithm for a team of six rovers. For simplicity, we assume that the value of $R$ for each rover is equal to the rover's ID number, as shown in the figure. The communication range of each rover is shown as a dashed red circle (a). At time 1, rover 1's countdown reaches zero and it declares itself a leader and broadcasts a RECRUIT message. The only rover in range is rover 2, which receives the message and declares itself a follow with rover 1 as its primary leader (b). At time 3, rover 3's countdown reaches zero and it broadcasts a RECRUIT message which is received by rovers 2 and 4. Rover 3 declares itself a follow with rover 3 as its primary leader and rover 2 records rover 3 as a secondary leader (c). At time 5, rover 5's countdown reaches zero and it broadcasts a RECRUIT message which is received by rovers 4 and 6. Rover 6 declares itself a follow with rover 5 as its primary leader and rover 4 records rover 5 as a secondary leader (d).

In this example, agent 2 is used to route messages between the clubs centered on agents 1 and 3. Agent 4 does so for the the clubs centered on agents 3 and 5.

Once clubs have been formed, the next stage in the formation of the agent hierarchy involves establishing a tree structure between the club leaders. This is achieved with a countdown algorithm very similar to that described above, known as *tree regions* [2]. The significant difference is that once an agent is recruited by a leader, it in turn recruits other agents as its followers. In this way, a tree structure is constructed, up to a maximum specified depth $h$.

As before, each agent counts down from a randomly generated integer. Once the countdown reaches zero the agent declares itself the root of a tree and recruits the leaders of its neighboring clubs as its children by broadcasting a RECRUIT message. The RECRUIT message is augmented with the current depth in the tree, which for this initial message is 1. The message is broadcast to the leaders of neighboring clubs and is routed via the agents that are members of both this club and the neighboring clubs. When an agent receives a RECRUIT message it records the sender as its parent in the tree structure. If the value of the current depth recorded in the received message is less than the specified maximum depth, the child agent then recruits the leaders of its neighboring clubs by broadcasting RECRUIT messages with an incremented

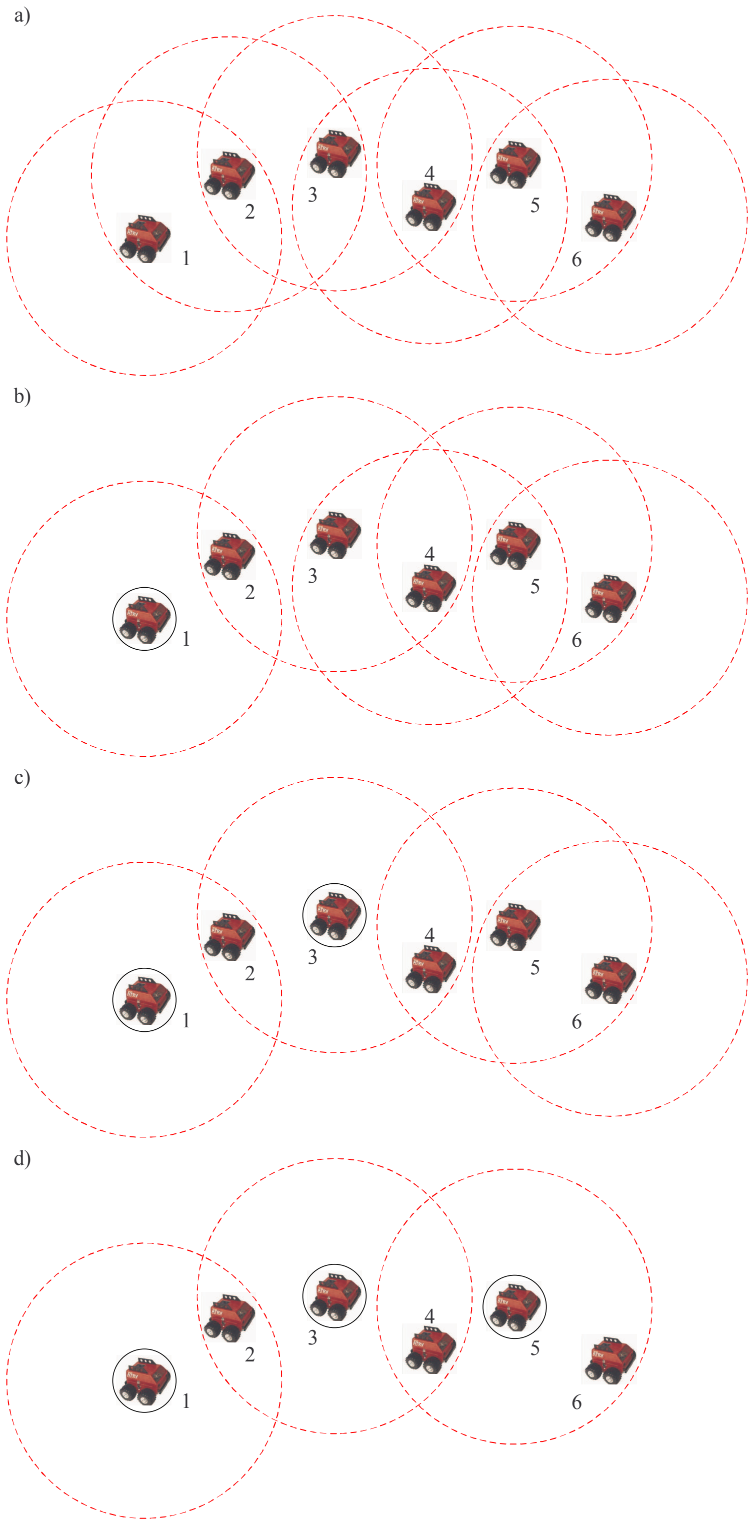


Figure 3-5: This figure shows the operation of the club formation algorithm on a group of rovers. (a) Agents with communication ranges. (b) At time 1, agent 1 declares itself a leader and agent 2 becomes its follower. (c) At time 3, agent 3 declares itself a leader and agents 2 and 4 become its followers. (d) At time 5, agent 5 declares itself a leader and agents 4 and 6 become its followers.

depth value.

The result of the tree regions algorithm is a tree hierarchy consisting of the club leaders. Since an agent in the hierarchy can communicate with its parent and children this structure provides a means for routing communication between any pair of agents. An example tree structure for the group of six rovers from Fig. 3-5 is shown in Fig. 3-6. In this example the root of the tree structure is rover 3 and rovers 1 and 5 are its children.

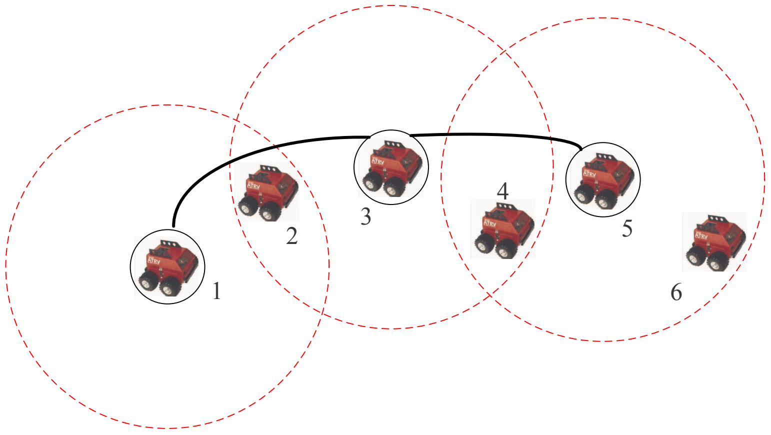


Figure 3-6: This figure shows the result of the tree hierarchy algorithm on a group of rovers. The root of the tree structure is rover 3 and rovers 1 and 5 are its children.

This chapter described the way in which a TPN is distributed between the agents in a plan, such that each agent is responsible for the portion of the plan relating to the activities it will carry out. We then presented the club formation and tree hierarchy algorithms, which form the agents into an ordered structure, such that communication routing can be established between all agents.

# Chapter 4

# Plan Reformulation

## 4.1 Introduction

The D-Kirk executive performs distributed execution of a temporally flexible plan. We use the techniques of dispatchable execution, described in Section 2.3, to retain temporal flexibility in the plan until dispatch time. The dispatcher uses this temporal flexibility to respond to the uncertain durations of activities in the plan. Dispatching must be done in real-time, so the system is susceptible to failure if the complexity of computation at this stage causes it to fall behind real-time. However, dispatchable execution requires significant computation at dispatch time. We therefore reformulate the plan to minimize the amount of computation that must be done at dispatch time. The output of the reformulation phase is a Minimal Dispatchable Graph (MDG), which makes explicit the temporal constraints in the plan such that correct dispatching requires only local propagation of timing information. This chapter describes the distributed reformulation algorithm used by D-Kirk, which is based on the fast reformulation algorithm by Tsamardinos et. al. [27].

First, we introduce some basic structures and algorithms which will be used repeatedly in the distributed reformulation algorithm. We then present the distributed reformulation algorithm itself and discuss its complexity.

## 4.2 Basic Structures and Algorithms

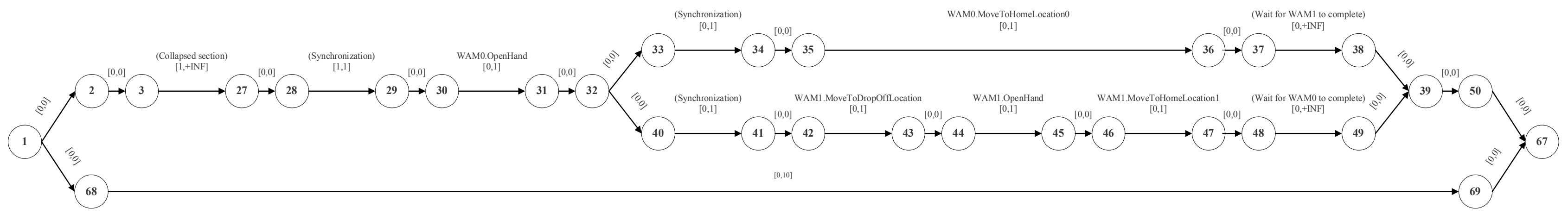
This section summarizes some basic structures and algorithms which are used repeatedly by the distributed reformulation algorithm. First we review a Simple Temporal Network (STN). We then review the distance graph representation of a STN. Next we introduce the distributed Bellman-Ford algorithm used to calculate the Single Source Shortest Path (SSSP) distances in a graph. We then introduce the predecessor graph and show how this can be formed from the SSSP distances. Next we discuss how a Depth First Search (DFS) is conducted in a distributed framework. Finally we introduce the post order and Reverse Post Order (RPO) for a graph search and describe how these can be recorded by extending the algorithm for distributed DFS.

### 4.2.1 Simple Temporal Network

We represent a temporally flexible plan as a Simple Temporal Network (STN). An STN was introduced in Section 2.3 and is a graph structure, where nodes represent time events and directed edges represent simple temporal constraints. The STN representation of the simplified plan from the manipulator tool delivery scenario was first shown in Fig. 2-2 and is repeated below in Fig. 4-1.

### 4.2.2 Distance Graph

The reformulation algorithm operates on the distance graph representation of the STN. A distance graph was introduced in Section 2.3 and takes the set of nodes from an STN but replaces each edge with a pair of edges in opposing directions. Each edge represents an upper bound on the relative execution times of the nodes it connects. One edge represents the upper bound and the other the lower bound of the simple temporal constraint. The distance graph representation of the STN in Fig. 4-1 was first shown in Fig. 2-3 and is repeated below in Fig. 4-2.



(Collapsed section) [1,+INF]
(Synchronization) [1,1]
WAM0.OpenHand [0,1]
(Synchronization) [0,1]
WAM0.MoveToHomeLocation0 [0,1]
(Wait for WAM1 to complete) [0,+INF]
(Synchronization) [0,1]
WAM1.MoveToDropOffLocation [0,1]
WAM1.OpenHand [0,1]
WAM1.MoveToHomeLocation1 [0,1]
(Wait for WAM0 to complete) [0,+INF]
[0,0]
[0,10]
1
2
3
27
28
29
30
31
32
33
34
35
36
37
38
39
50
40
41
42
43
44
45
46
47
48
49
67
68
69

Figure 4-1: This figure shows the Simple Temporal Network representation of the simplified plan from the manipulator tool delivery scenario and was first shown in Fig. 2-2.

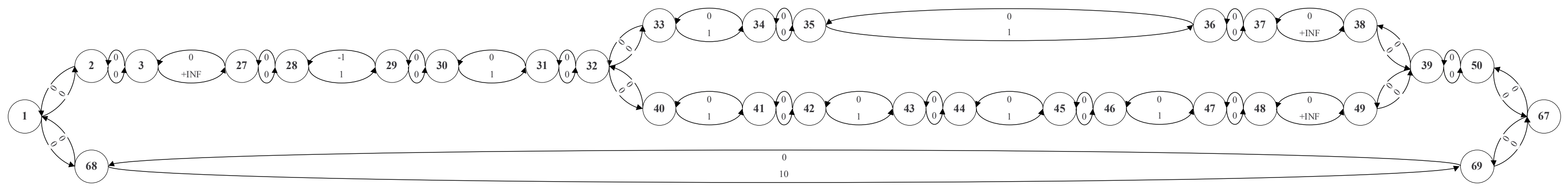


Figure 4-2: This figure shows the distance graph representation of the Simple Temporal Network in Fig. 4-1 and was first shown in Fig. 2-3.

### 4.2.3 Distributed Bellman-Ford

The reformulation algorithm requires calculation of the Single Source Shortest Path (SSSP) distance to each node on multiple occasions. We use the distributed Bellman-Ford algorithm [15] to calculate the SSSP distances because it offers a run time linear in the number of nodes in the graph. The Bellman-Ford algorithm is a type of label-correcting algorithm, where an initial estimate of the SSSP distance at each node is refined over a number of iterations. At each iteration, the estimated distance at a given node is used to update the estimates at its neighboring nodes, where a neighbor node is one with which the node shares an edge. If no negative cycles exist, the estimates converge to the true SSSP distance values. If the algorithm is run synchronously, where all nodes complete their $i$th distance estimate update before any node begins update $i+1$, then convergence will be achieved after $N-1$ updates, where $N$ is the number of nodes in the graph.

The distributed Bellman-Ford algorithm requires only local knowledge of the graph at each node, hence allowing the SSSP calculation to be performed locally. This algorithm is fully parallel.

The edges in the distance graph are *directed*, in that they impose an upper bound on the execution time of the target node relative to that of the source node. We must therefore distinguish between *outgoing* and *incoming* edges. In the Bellman-Ford algorithm, nodes send SSSP distance estimate updates to neighbor nodes on outgoing edges and receive updates from nodes on incoming edges. We refer to these nodes as outgoing an incoming neighbors respectively. For example, in the example distance graph in Fig. 4-3(a), node B has nodes A and D as incoming neighbors and node A as its outgoing neighbor.

In the distributed Bellman-Ford algorithm, a node exchanges messages with its neighbor nodes to update their SSSP distance estimates. Each node knows the values of $e_i$, the edge length to its neighbor $i$, for all outgoing neighbors $i$. It also maintains an estimate *dist* of its SSSP distance. Each node sends its distance estimate to its outgoing neighbors and receives distance estimates from its incoming neighbors.

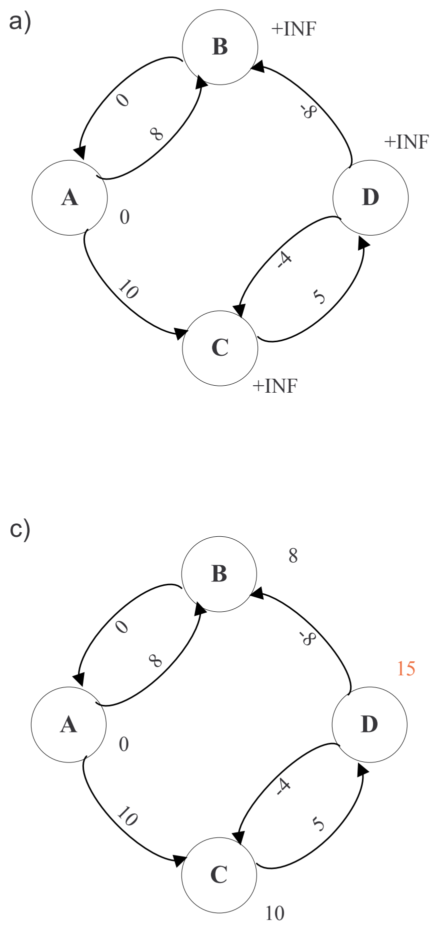


Figure 4-3: This figure shows the operation of the distributed Bellman-Ford algorithm on an example distance graph without any negative cycles. The start node is node A. (a) Input distance graph with initial distance estimates. (b) In round 1, nodes B and C update their distance estimates. (c) In round 2, node D updates its distance estimate. (d) In round 3, node B updates its distance estimate and the values have converged.

The pseudo-code for the distributed Bellman-Ford algorithm is shown in Fig. 4-4. Line numbers in the following description refer to this pseudo-code.

```
 1: $round \leftarrow 1$
 2: if start node then
 3:     $dist \leftarrow 0$
 4: else
 5:     $dist \leftarrow +\mathrm{INF}$
 6: end if
 7: while $round < N$ do
 8:     Send $dist$ to outgoing neighbors
 9:     Wait for distance update $d_i$ from all incoming neighbors $i$
10:     $dist \leftarrow min_i\,(dist, d_i + e_i)$
11:     $round \leftarrow round + 1$
12: end while
```

Figure 4-4: This figure shows the pseudo-code for the distributed Bellman-Ford algorithm.

Consider the example distance graph in Fig. 4-3, for which we wish to compute the SSSP distances from node A using the distributed Bellman-Ford algorithm. The distance graph has four nodes and does not contain negative cycles, so the distributed Bellman-Ford algorithm will converge in three rounds. Fig. 4-3(a) shows the distance graph with the initial distance estimates *dist*. The initial values of *dist* are all +INF (line 5) except for that of the start node, node A, which has a SSSP distance of zero by definition (line 3). In each round, each node sends it current distance estimate to its outgoing neighbors (line 8), such that each node receives distance estimates from all of its incoming neighbors (line 9). Each node then uses this information to update its SSSP distance estimate (line 10). For example, in round 1, node B receives distance estimates of zero and +INF from nodes A and D respectively. The shortest path distance to node B implied by the distance estimate from node A is 8, because the edge from A to B has length 8. That from node D is +INF. The value of $min_i\,(dist, d_i + e_i)$ for node B in round 1 is therefore 8 and node B updates its value of *dist* to 8. In round 2, node B again receives distance estimates of zero and +INF from nodes A and D respectively. Again, the value of $min_i\,(dist, d_i + e_i)$ for node B in round 1 is 8, so its value of *dist* remains unchanged. However, node D receives a distance estimate of 10 from node C in round 2, so updates its distance estimate to 15. In round 3, node B receives distance estimates of zero and 15 from nodes A and D respectively. The value of $min_i\,(dist, d_i + e_i)$ is now 7, so node B updates its value of $dist$ accordingly. The values have now converged (line 7).

### 4.2.4 Predecessor Graph

For a given start node, the predecessor graph contains only the edges which define the shortest path to each node. The predecessor graph for the example distance graph in Fig. 4-3 is shown in Fig. 4-5. For example, the SSSP distance of node C is 10, which is due to the edge of length 10 from node A. Node A is therefore node C's predecessor and the edge AC is in the predecessor graph. It is possible for a node to have multiple predecessors, including itself. This occurs in the case of rigid components, which were introduced in Section 2.3, and causes the predecessor graph to become cyclic. This is discussed in detail in Section 4.3.2.

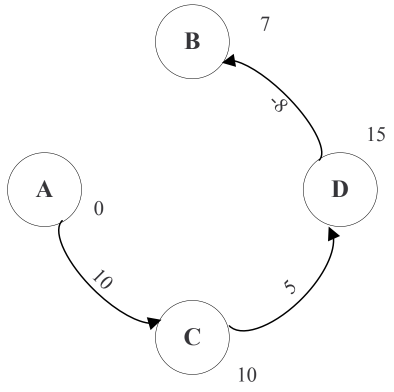


Figure 4-5: This figure shows the predecessor graph for the example distance graph in Fig. 4-3.

We use the predecessor graph for identifying rigid components and for establishing the order in which nodes are visited in the dominance test traversals. We obtain the predecessor graph by adding another round of updates to the distributed Bellman-Ford algorithm. Since this round follows the $N-1$ rounds required for convergence, the SSSP distances are accurate. If a node receives from a node $i$ a SSSP distance $d_i$ such that $d_i + e_i = dist$ then node $i$ is a predecessor and the edge from node $i$

belongs in the distance graph. For example, in the distance graph in figure Fig. 4-3, in a fifth round of updates, node B receives SSSP distances of zero and 15 from nodes A and D respectively. This gives values of $d_i + e_i$ of 8 and 7 respectively. Therefore, node D is a predecessor of node B, but node A is not, and edge DB is a member of the predecessor graph, as shown in Fig. 4-5. The computational complexity for each node is $O(e)$, where $e$ is the number of edges per node.

### 4.2.5 Distributed Depth First Search

In a Depth First Search (DFS) a node waits for the search in a given neighbor node to complete before instigating search in the next neighbor. As a result, a DFS must be performed sequentially, rather than in parallel. In a distributed DFS, a node sends a SEARCH message to instigate search in a single neighbor and then waits for a reply. When the neighbor node has completed its part in the DFS it returns a DONE message to the original node, which in turn sends a message to instigate search in the next neighbor or to signal that it too is done.

Following this simple scheme it is possible for a node to send SEARCH messages to instigate search in nodes which have already been explored. We wish to avoid avoid this inefficiency, so the algorithm maintains a list of visited nodes and nodes instigate search only in neighbor nodes that have not been visited.

In the case of a directed graph, such as a distance graph, SEARCH messages are sent to outgoing neighbors and received from incoming neighbors only. Similarly, DONE messages are sent to incoming neighbors and received from outgoing neighbors only. Pseudo-code for the distributed DFS algorithm is shown in Fig. 4-6 and line numbers in the following description refer to this pseudo-code.

We demonstrate the operation of the distributed DFS algorithm on the example distance graph in Fig. 4-7. Nodes are colored light gray once they have been visited and dark gray once their part in the search is complete. The initial state is shown in Fig. 4-7(a). The start node is node A, which does not wait for the receipt of a SEARCH message (line 2) and begins the DFS. Node A has two outgoing neighbors, B and D and begins by sending a SEARCH message with to node B, passing a visited

```
 1: $visitedList \leftarrow ID$
 2: if This is not start node then
 3:     Wait for SEARCH message $msg$
 4:     $visitedList \leftarrow (visitedList, msg.visitedList)$
 5: end if
 6: for All outgoing neighbors $i$ not in $visitedList$ do
 7:     Send SEARCH message to neighbor $i$ with parameter $visitedList$
 8:     Wait for DONE message $m$ from neighbor $i$
 9:     $visitedList \leftarrow m.visitedList$
10: end for
11: if This is not start node then
12:     Send DONE message to node $m.sender$
13: end if
```

Figure 4-6: This figure shows the pseudo-code for the distributed Depth First Search algorithm.

list of $< A >$ (line 7). Node A then waits for a response from node B (line 8). Node B receives the SEARCH message from node A (line 3) and appends the received visited list to its own list, giving a list of $< B, A >$. Node B has one outgoing neighbor, node A, but this has already been visited, so it does not send any SEARCH messages. It sends a DONE message back to node A (line 12) and search of node B is now complete. This is shown in Fig. 4-7(b). Node A receives the DONE response from node B (line 8). It then sends a SEARCH message to its other neighbor, node C, passing a visited list of $< B, A >$ (line 7). Node C receives the SEARCH message (line 3) and appends the received visited list to its own list, giving a list of $< C, B, A >$. This is shown in Fig. 4-7(c). Node C has one outgoing neighbor, node D, to which it sends a SEARCH message, passing a visited list of $< C, B, A >$ (line 7). Node D receives the SEARCH message (line 3) and appends the received visited list to its own list, giving a list of $< D, C, B, A >$. Node D has one outgoing neighbor, node B, but this has already been visited, so it does not send any SEARCH messages. It sends a DONE message back to node C (line 12) and search of node D is now complete. This is shown in Fig. 4-7(d). Node C receives the done RESPONSE from node D (line 8). Node C has completed search in all of its outgoing neighbors, so it sends a DONE message back to node A (line 12). Search of node C is now complete. This is shown in Fig. 4-7(e). Node A has now completed search in all of its outgoing neighbors. It is the start node of the search, so it does not send a DONE message (line 11) and search of node A and the entire DFS are now complete. This is shown in Fig. 4-7(f).

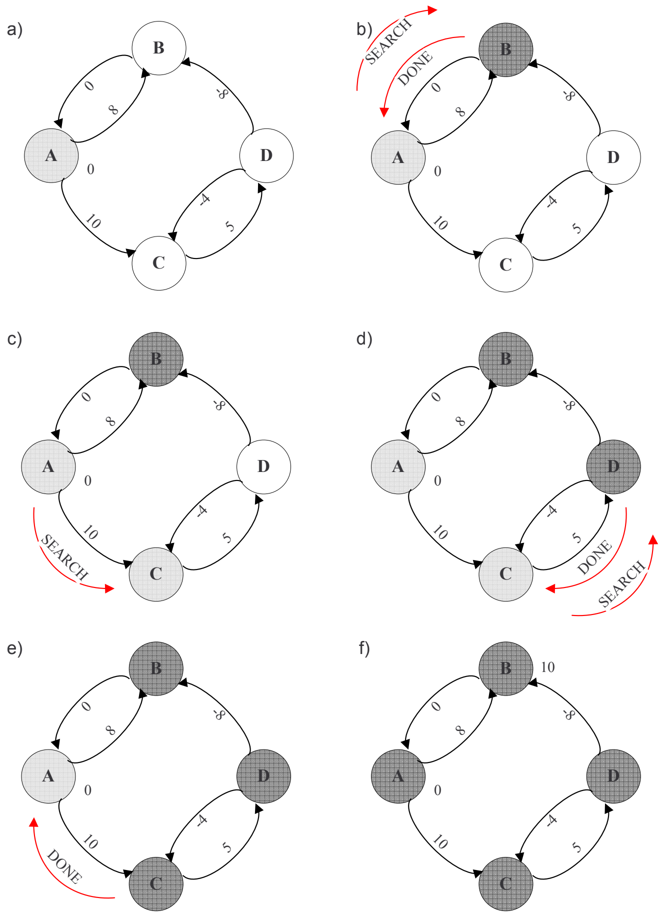


Figure 4-7: This figure shows the operation of the distributed Depth First Search Algorithm on the example distance graph from Fig. 4-3. Nodes are colored light gray once they have been visited and dark gray once their part in the search is complete. (a) The search begins at node A. (b) Node A sends a SEARCH message to its first outgoing neighbor, node B, which has no outgoing neighbors that have not been visited, so replies with a DONE message. (c) Node A then sends a SEARCH message to its other neighbor, node C. (d) Node C sends a SEARCH message to its only outgoing neighbor, node D. Node D has no outgoing neighbors that have not been visited, so replies with a DONE message. (e) Node C sends a DONE message back to node A. (f) The DFS is complete.

We use a depth first search in a number of places in the distributed reformulation algorithm. First, we use DFS to obtain the list of nodes reachable by following edges from a given node. This is required for correct operation of the Bellman-Ford algorithm, which is used throughout the distributed reformulation algorithm. Also, we use DFS to determine the order in which nodes should be visited in each dominance test traversal.

### 4.2.6 Post Order

The post order is the order in which nodes are removed from the search queue during search. In the distributed case, since there is no global search queue, this corresponds to the order in which nodes complete their part in the search. For example, in the DFS example in Fig. 4-7, the post order is $< B, D, C, A >$. Note that the post order is unrelated to the order in which nodes are added to the visited list.

In the distributed reformulation algorithm we make use of the Reverse Post Order (RPO). The RPO is simply the reverse of the post order. We use the RPO from a DFS on the transposed predecessor graph of the input distance graph in the algorithm used to identify rigid components. Also, each dominance test traversal visits nodes in the order given by the RPO from a DFS on the predecessor graph rooted at a given node in the distance graph.

In the distributed framework we obtain the RPO by adding to the SEARCH and DONE messages used in the distributed DFS algorithm the ID of the last node to *post*, or complete its part in the search. When a node sends a SEARCH message, it sends the ID of the last posted node from the last SEARCH or DONE message it received. When a given node posts, it records the value of the last node to post, and sends its own identity with the DONE message that it sends to signal its completion. In this way, when the search is complete, each node has a record of the node that posted immediately before it. The start node of the search is always the last node to post, and therefore the first in the RPO, so we can follow these pointers from the start node, through the graph, to trace out the RPO.

For example, Fig. 4-8 shows the example distance graph used in the DFS example in Fig. 4-7, but with the last posted node values added to the messages.

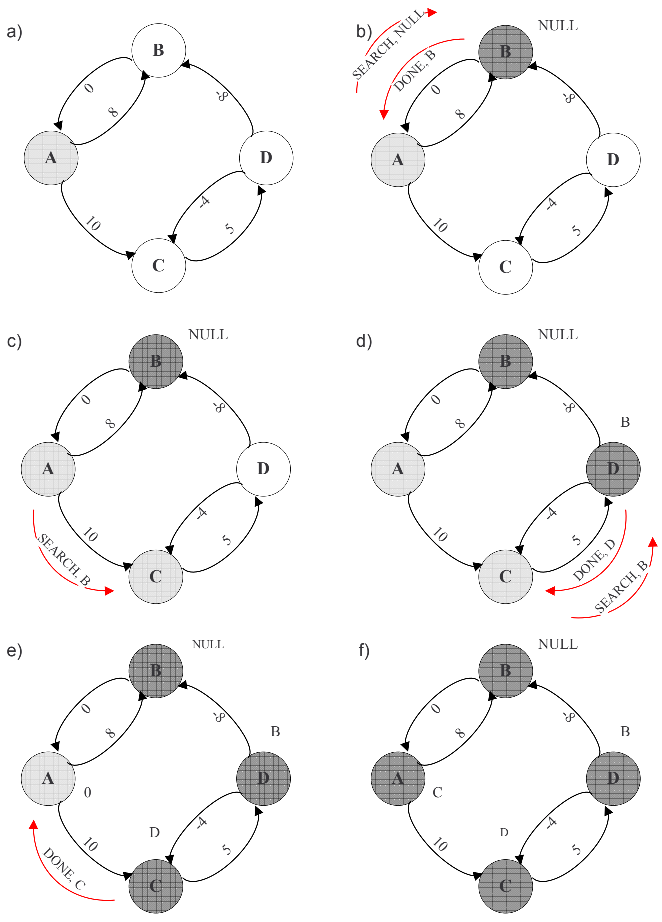


Figure 4-8: This figure shows the operation of the distributed Depth First Search Algorithm, with the addition of RPO recording. The ID of the last posted node is included with each messages. Nodes are colored light gray once they have been visited and dark gray once their part in the search is complete and are labeled with the last posted node.

## 4.3 Distributed Reformulation

The D-Kirk executive uses the techniques of dispatchable execution to maintain temporal flexibility in the plan until dispatch time. This increases the complexity of plan execution. To minimize computational load on the dispatcher, which must operate in real-time, we reformulate the plan prior to dispatching. This section describes the details of the distributed reformulation algorithm. We also present the computational complexity of each phase of the algorithm, for comparison with the centralized case.

The distributed reformulation algorithm takes as input a temporally flexible plan in the form of a Simple Temporal Network (STN) and produces a Minimal Dispatchable Graph (MDG). A dispatchable graph makes explicit all of the timing constraints present in the TPN, such that the dispatcher needs to make only local propagation of execution times to ensure a consistent execution. An MDG also minimizes the number of propagations required, to further reduce dispatching complexity.

The All Pairs Shortest Path (APSP) graph formed from the distance graph representation of the plan is dispatchable, but not minimal. The most straightforward reformulation algorithm [20] constructs the entire APSP before trimming redundant edges to obtain the MDG. However, formation of the APSP graph is inefficient in terms of both time and space. D-Kirk uses a distributed version of the fast reformulation algorithm [27], which obtains the MDG without forming the APSP graph.

Recall that the fast reformulation algorithm does not provide an MDG for which correct execution is guaranteed if the input plan contains uncontrollable activities. To make this guarantee we must check for the dynamic controllability of the plan, as described in Section 2.3. Extension of the D-Kirk distributed reformulation algorithm to provide dynamic controllability checking is an avenue for future work and is discussed in Section 7.5.

### 4.3.1 Algorithm Structure

We distribute the fast reformulation algorithm between all nodes, using a message passing scheme. The algorithm is event driven, where an event is the receipt of a message. However, since the algorithm makes repeated use of a number of techniques, described in Section 4.2, the action to take on receipt of a given message depends upon the node's current state within the algorithm. It is therefore natural to use a state machine approach to express the structure of the reformulation algorithm and to allow each node to remember its progress through the computation.

This state information allows a node to act only on received messages relevant to the phase of the computation with which it is currently concerned, while postponing processing of any messages for future phases for which the node is not yet ready. Similarly, we can detect erroneous messages, which provides significant error checking. This means that the algorithm is robust to delays in message delivery and ensures correct operation even when messages arrive in an order different from that in which they were sent.

Throughout the algorithm, we exploit parallel processing wherever possible. In the following sections, we discuss the approaches used in each section of the distributed reformulation algorithm, and show how efficiency is maximized.

Fig. 4-9 shows the high-level pseudo-code for the distributed fast reformulation algorithm. Details of the centralized fast reformulation algorithm were presented in Section 2.3. The way in which each phase of the algorithm is implemented in a distributed framework is presented in the following two subsections.

```
1: Process rigid components
2: Identify non-dominated edges
```

Figure 4-9: This figure shows the high-level pseudo-code for the distributed reformulation algorithm.

Throughout the following discussions, we illustrate the operation of the distributed fast reformulation algorithm using the simplified version of the manipulator tool delivery scenario introduced in Chapter 1. Fig. 4-10 shows the distance graph representation of the plan describing this scenario, which was first shown in Fig. 2-3.

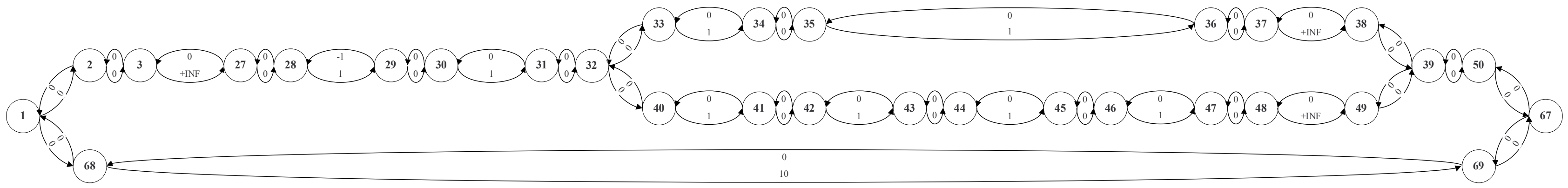


Figure 4-10: This figure shows shows the distance graph representation of the plan describing the simplified manipulator tool delivery scenario and was first shown in Fig. 2-3.

### 4.3.2 Processing of Rigid Components

The first section of the fast reformulation algorithm processes Rigid Components (RCs) to improve efficiency. Recall that an RC is a set of nodes whose execution times are rigidly fixed relative to each other. The algorithm represents each RC in the plan as a single node for the purposes of identifying non-dominated edges. This reduces the effective size of the plan and hence improves efficiency. Processing of rigid components in this way is also required for the correct operation of the graph traversals used to identify non-dominated edges. It is also required that Zero-Related (ZR) nodes are collapsed to a single node. Zero-related nodes are a special-case subset of the members of an RC and are constrained to execute simultaneously.

Pseudo-code for the RC processing phase of the distributed reformulation algorithm is shown in Fig. 4-11. It consists of two main phases. First we identify the RCs in the graph. Second, we rearrange the edges and activities in each RC.

```
1: Compute SSSP distances from phantom node using synchronous distributed
   Bellman-Ford
2: Form predecessor graph using SSSP distances
3: Perform DFS on predecessor graph and record RPO
4: for Each node in the graph, taken in RPO order do
5:   Perform search on transposed predecessor graph to extract members of this RC
     and their edges
6:   Determine member node with minimum SSSP distance and set as RC leader
7:   for Each member node do
8:     if Member of a ZR component then
9:       Collapse with ZR component leader
10:    else
11:      Add to MDG doubly linked chain edges
12:      Relocate edges connected to nodes outside the RC to the leader
13:      Delete all other edges
14:    end if
15:  end for
16: end for
```

Figure 4-11: This figure shows the pseudo-code for the rigid component processing phase of the distributed reformulation algorithm.

**RC Identification**

Before we can process the rigid components we must identify them. We identify a rigid component using the Strongly Connected Component (SCC) identification algorithm [3]. This involves the following steps. Line numbers refer to the pseudo-code for the RC processing algorithm in Fig. 4-11.

1. **Calculate the SSSP distance of every node from a phantom node and construct the predecessor graph.**

   The SCC identification requires the SSSP distance to each node from a *phantom* node, a virtual node which has edges of zero length to every other node in the plan (line 1). The distance graph with phantom node is shown in Fig. 4-12.

   We use the distributed Bellman-Ford algorithm to find the SSSP distance to each node. In practice, we simulate the presence of the phantom node by initializing each node's distance estimate in the Bellman-Ford calculation to zero, rather than +INF. This equates to modifying line 5 in Fig. 4-4. This means that after tightening the SSSP distance estimates in each round, each node's SSSP distance is at most zero, which is exactly the constraint imposed by the phantom node and its additional edges.

   We form the predecessor graph (line 2) by conducting an additional round in the synchronized distributed Bellman-Ford algorithm, as described in Section 4.2. The SSSP distances and predecessor graph for the example plan are shown in Fig. 4-13.

2. **Conduct a DFS on the predecessor graph from the phantom node and record the RPO.**

   The next step is to conduct a DFS on the distance graph and to record the RPO (line 3). Since the phantom node is a virtual node, we must simulate the fact that the DFS starts there. We achieve this by starting a DFS from every node that has a zero SSSP distance. The order in which we choose start nodes is arbitrary, as this corresponds to the arbitrary choice between neighbor nodes

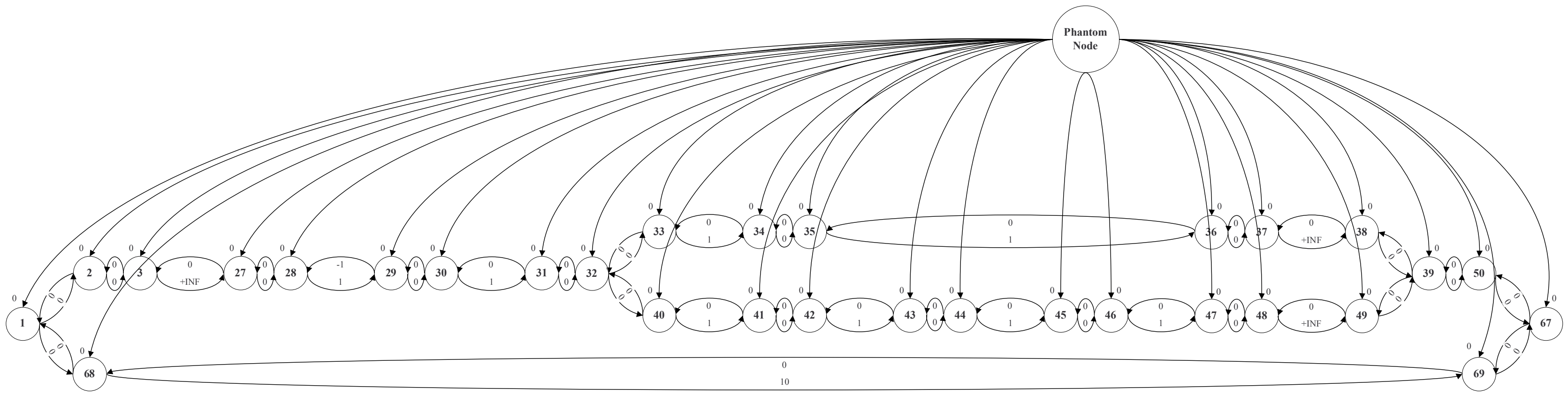


Figure 4-12: This figure shows the distance graph with phantom node and corresponding zero length edges added.

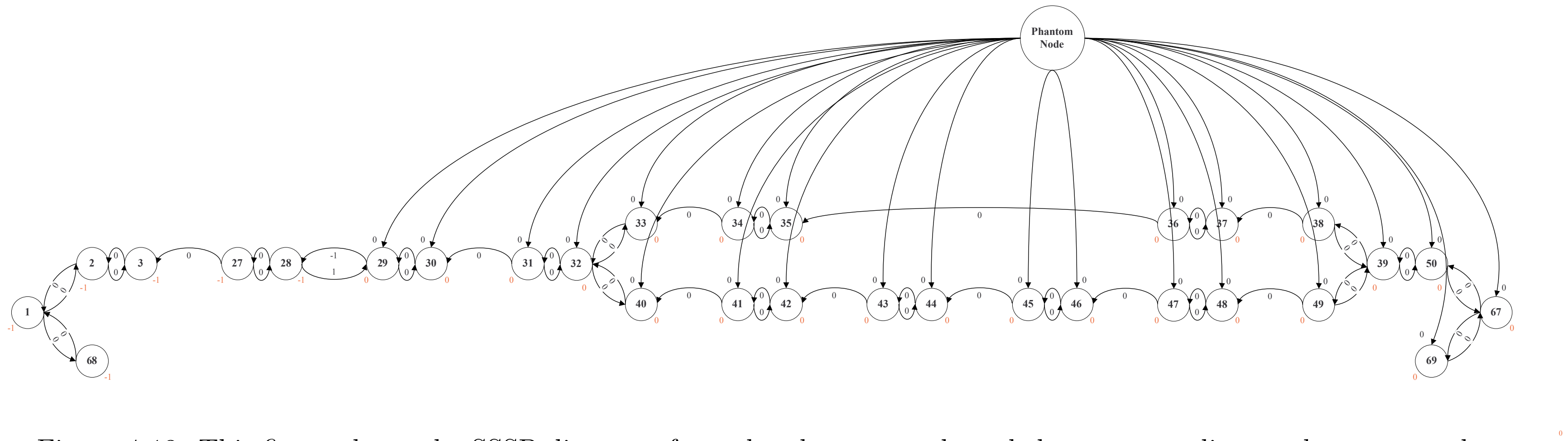


Figure 4-13: This figure shows the SSSP distances from the phantom node and the corresponding predecessor graph.

at the phantom node. Recall that in the distributed case we store the RPO in the form of a pointer at each node to the lats posted node.

We illustrate the DFS and RPO recording on the example plan. For simplicity, we choose start nodes for each DFS in ascending numerical order of their ID. The first DFS begins at node 29, and records a post order of $<$ 68, 1, 2, 3, 27, 28, 30, 29 $>$. Fig. 4-14 shows the predecessor graph, with the nodes visited in this in DFS in red, and each node labeled with the ID of the last posted node.

Since node 30 has already been visited, the second DFS begins at node 31 and records a post order of $<$ 33, 40, 32, 31 $>$. This is shown in Fig. 4-15.

Since nodes 32 and 33 have already been visited, the third DFS begins at node 34 and records a post order of $<$ 35, 34 $>$. This is shown in Fig. 4-16.

Since node 35 has already been visited, the fourth DFS begins at node 36 and records a post order of $<$ 37, 36 $>$. This is shown in Fig. 4-17.

Since node 37 has already been visited, the fifth DFS begins at node 38. Node 40 has already been visited, so is not visited by this search. The DFS records a post order of $<$ 69, 67, 50, 41, 42, 43, 44, 45, 46, 47, 48, 49, 39, 38 $>$. This is shown in Fig. 4-18. The complete RPO is therefore $<$ 38, 39, 49, 48, 47, 46, 45, 44, 43, 42, 41, 50, 67, 69, 36, 37, 34, 35, 31, 32, 40, 33, 29, 30, 28, 27, 3, 2, 1, 68 $>$.

A given node must complete its part in forming the predecessor graph before it can take part in RPO extraction, but we do not require all nodes to have completed the predecessor graph for the RPO extraction to begin. The RPO extraction process is begun as soon as the start node has completed its part in forming the predecessor graph, so it occurs concurrently with formation of the predecessor graph, waiting for a node to finish its part in forming the predecessor graph where necessary.

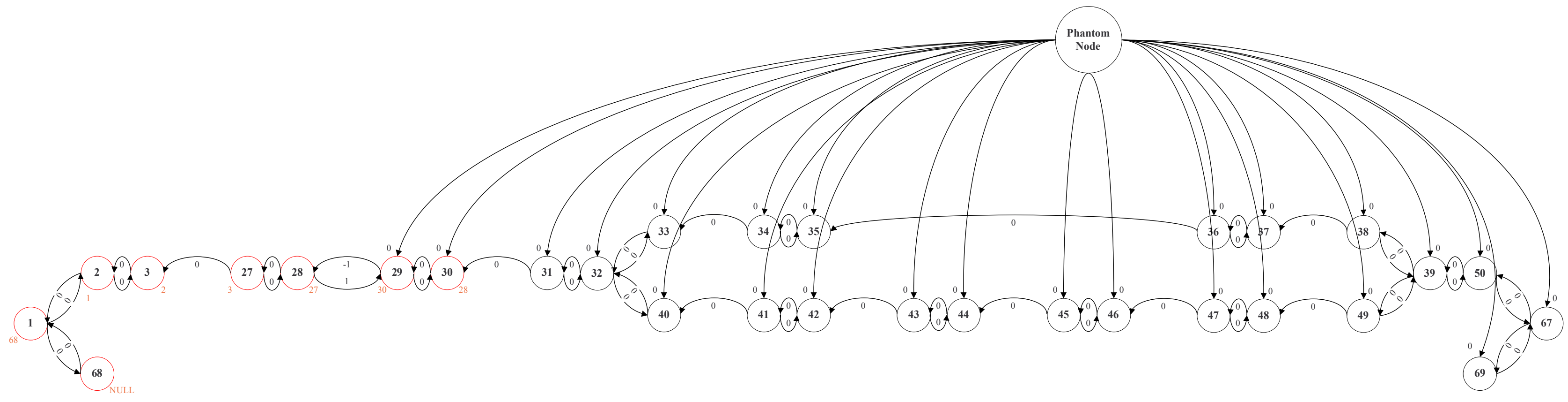


Figure 4-14: This figure shows the predecessor graph for the first DFS, which starts at node 29 and records a post order of < 68, 1, 2, 3, 27, 28, 30, 29 >.

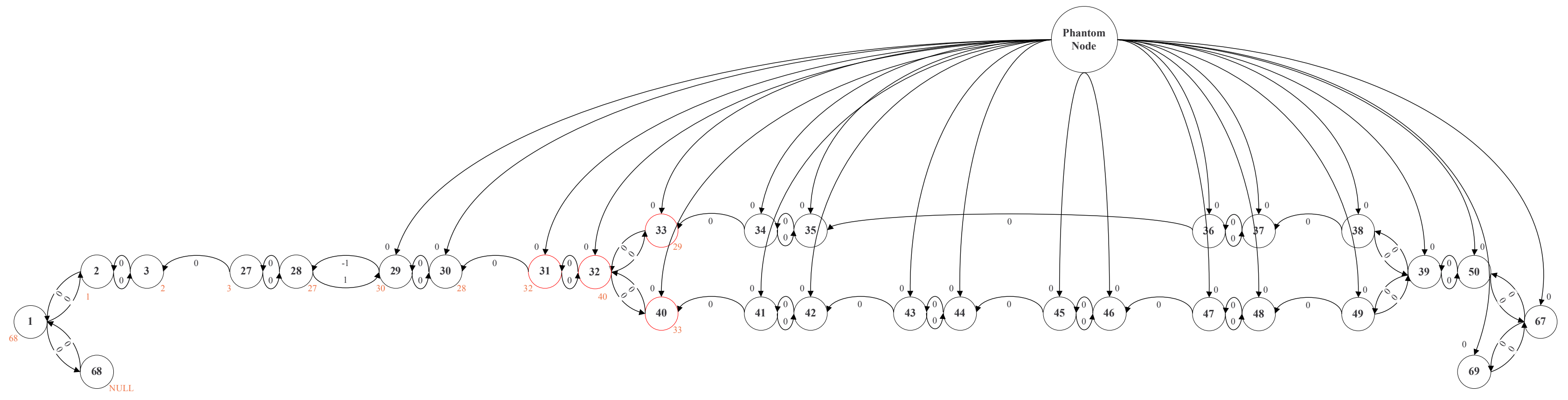


Figure 4-15: This figure shows the predecessor graph for the second DFS, which starts at node 31 and records a post order of $< 33, 40, 32, 31 >$.

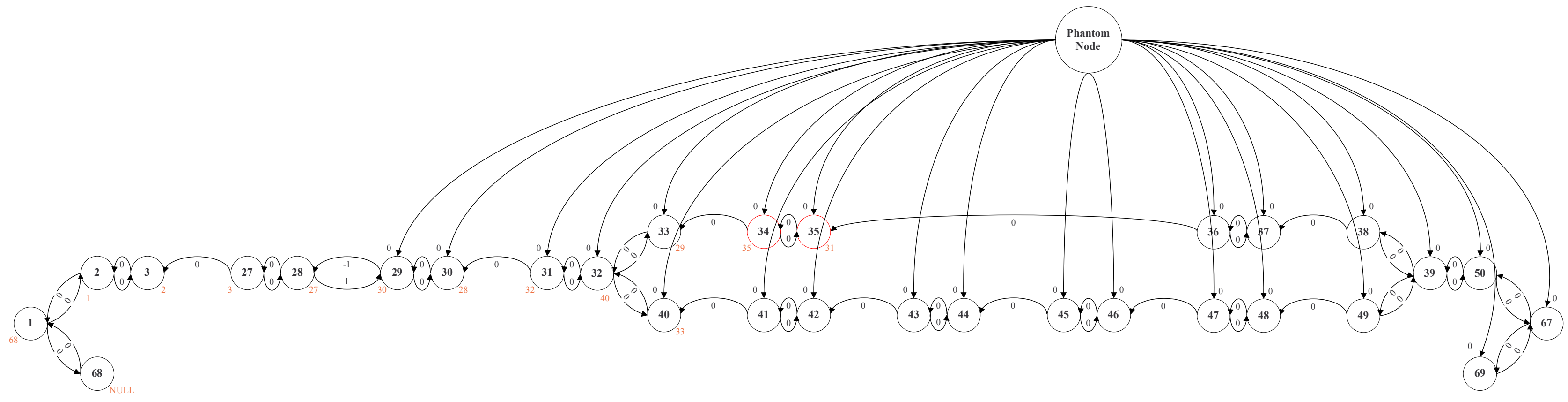


Figure 4-16: This figure shows the predecessor graph for the third DFS, which starts at node 34 and records a post order of < 35, 34 >.

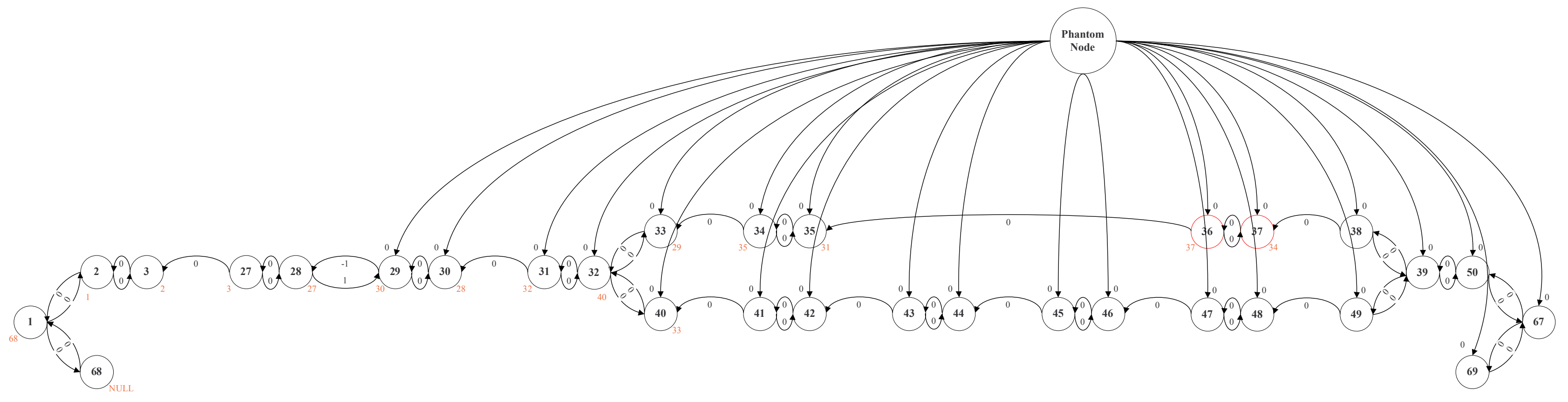


Figure 4-17: This figure shows the predecessor graph for the fourth DFS, which starts at node 36 and records a post order of < 37, 36 >.

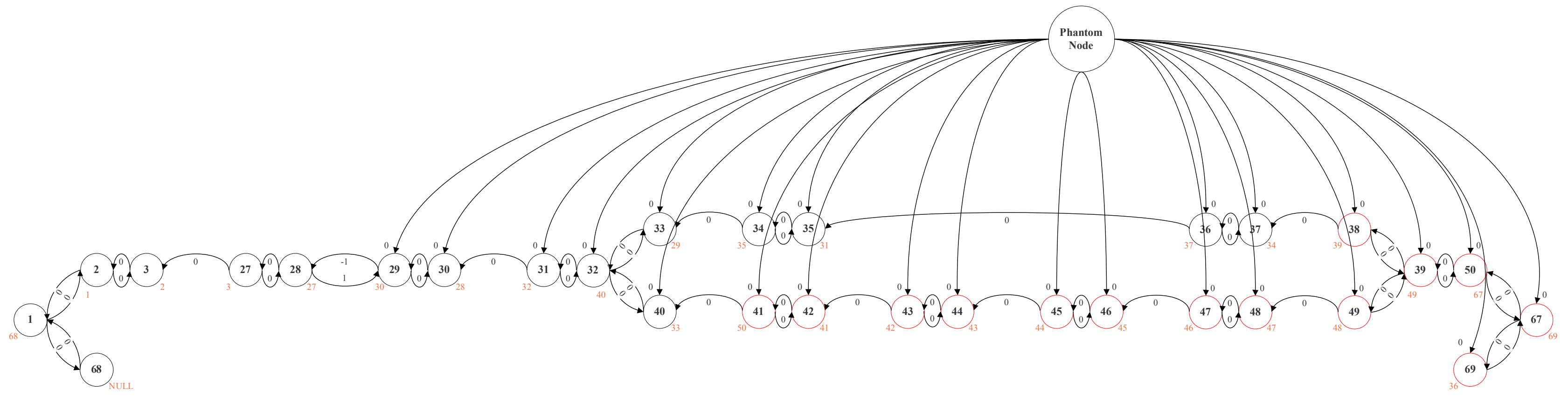


Figure 4-18: This figure shows the predecessor graph for the fifth DFS, which starts at node 38 and records a post order of < 69, 67, 50, 41, 42, 43, 44, 45, 46, 47, 48, 49, 39, 38 >.

3. **Conduct a series of searches on the *transposed* predecessor graph, selecting start nodes in the order given by the RPO just calculated, to identify the members of each RC.**

   The final step in identifying rigid components is to conduct a series of searches on the transposed predecessor graph, without the phantom node (line 5). We start a search from every node in the graph, with the order determined by the RPO for DFS on the predecessor graph (line 4). The nodes visited in each search belong to a single RC.

   The purpose of each search is to identify the members of each RC: the order in which nodes are visited is unimportant. We therefore use a parallel search, in which a node instigates search in its neighbors simultaneously. We achieve this by using a modified version of the distributed DFS described in Section 4.2, where a node does not wait for a DONE reply from a given neighbor before sending a SEARCH message to the next neighbor. Instead, a node sends all SEARCH messages simultaneously and waits to receive all DONE messages before replying to the node which instigated the search. Note, however, that the order in which the searches are carried out relative to each other *is* important. The searches, therefore, are carried out sequentially, following the RPO for DFS on the predecessor graph to determine the selection order of start nodes.

   Within each search, we record the members of the RC by adding to each SEARCH and DONE message a list containing the members of the RC so far discovered. The search terminates at the node which started it, which then has the complete list of RC members. Note that we also include in the list the SSSP distance from the phantom node and the list of distance graph edges for each member node. The member node deletes its record of these edges at this time. These edges are relocated to the RC leader when the RC is rearranged, as described below.

   For our example plan, the first search begins at node 38, as this is first in the RPO calculated above. This search visits nodes 38, 39, 49, 50, 67 and 69, which form the members of an RC. This RC is highlighted in Fig. 4-19, which shows the transposed predecessor graph for the example plan.

The nodes following node 38 in the RPO are 39, 49 and 48. Nodes 39 and 49 have already been visited, so the second search begins at node 48 and visits nodes 47 and 48. This RC is shown in Fig. 4-20.

The next five RCs, with members 45 and 46; 43 and 44; 41 and 42; 36 and 37; and 34 and 35 are extracted in a similar way.

The next search, number eight, starts at node 31. Nodes 34 and 41 have already been visited, so this search visits nodes 31, 32, 33 and 40. This RC is shown in Fig. 4-21.

The nodes following node 31 in the RPO are 32, 40, 33 and 29. Nodes 32, 40 and 33 have already been visited, so the ninth search begins at node 29 and visits nodes 27, 28, 29 and 30. This RC is shown in Fig. 4-22.

The nodes following node 29 in the RPO are 30, 28, 27 and 3. Nodes 30, 28 and 27 have already been visited, so the tenth search begins at node 3 and visits nodes 1, 2, 3 and 68. This RC is shown in Fig. 4-23, which shows the transposed predecessor graph with RC identification complete.

### RC Rearrangement

Having identified the members of each RC, we process the RC. This involves the following steps. Line numbers refer to the pseudo-code for the RC processing algorithm in Fig. 4-11.

1. **Select as the RC leader the member node with minimum SSSP distance.**

   Each search used to extract the members of an RC terminates at the node which started it. This start node assigns the RC leader (line 6). Recall that the RC leader represents the RC in the remainder of the reformulation algorithm. MDG edges are created to the leader, so the leader must be the first node in the RC

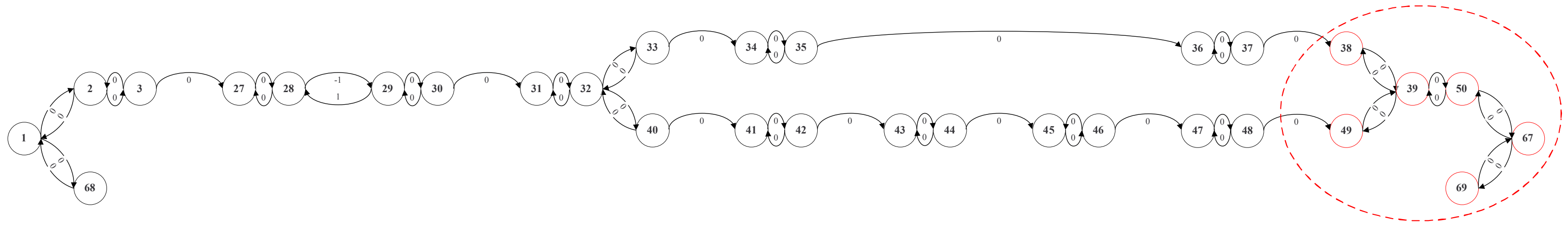


Figure 4-19: This figure shows the transposed predecessor graph for the example plan, with the RC extracted by the first search, from node 38, highlighted.

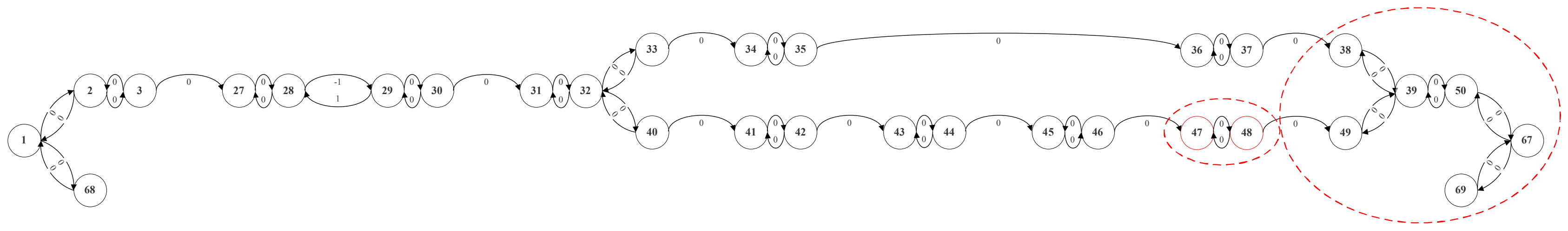


Figure 4-20: This figure shows the transposed predecessor graph for the example plan, with the RC extracted by the second search, from node 48, highlighted.

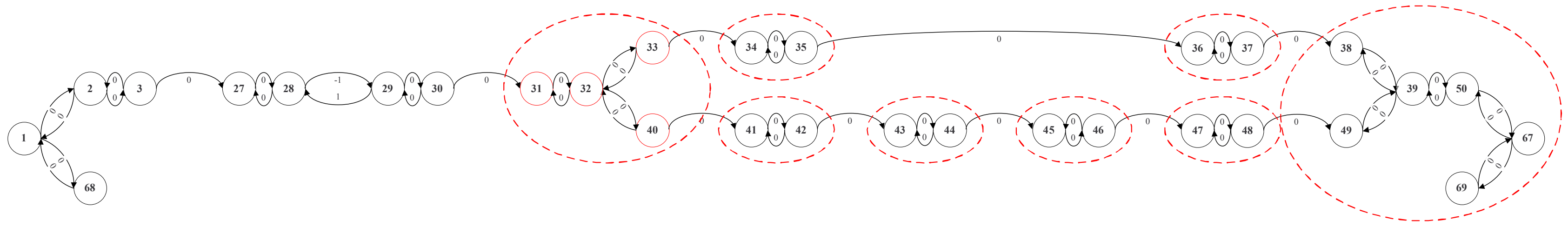


Figure 4-21: This figure shows the transposed predecessor graph for the example plan, with the RC extracted by the eighth search, from node 31, highlighted.

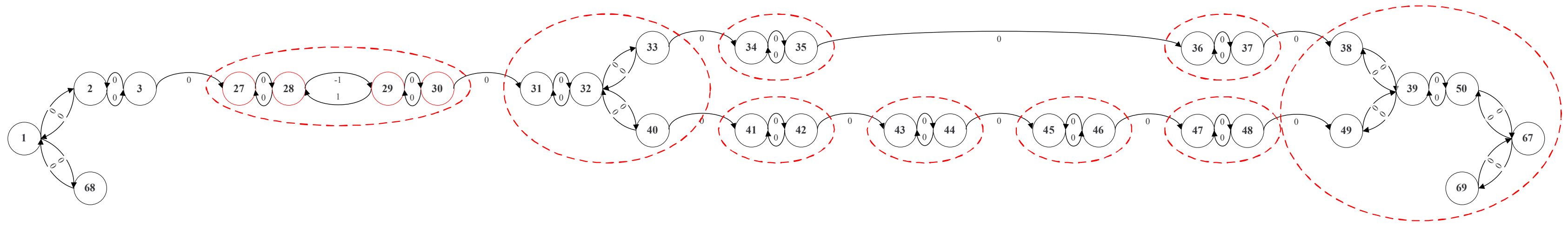


Figure 4-22: This figure shows the transposed predecessor graph for the example plan, with the RC extracted by the ninth search, from node 29, highlighted.

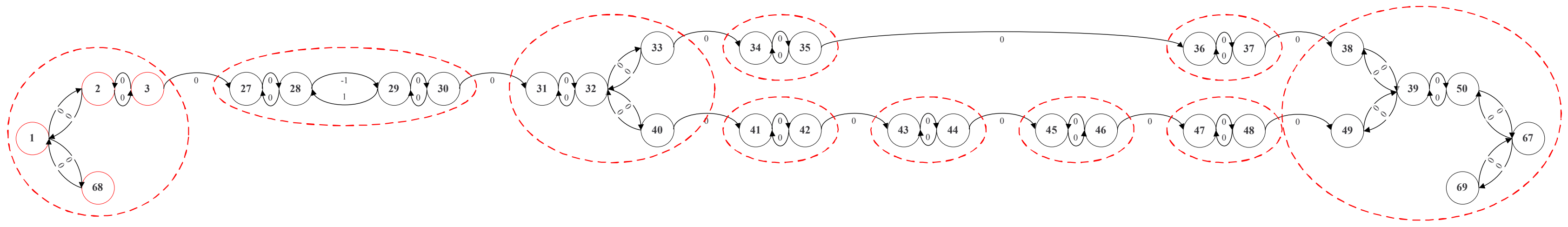


Figure 4-23: This figure shows the transposed predecessor graph for the example plan, with the RC extracted by the tenth search, from node 3, highlighted.

to be executed, to ensure that the plan can be dispatched correctly. The start node also initiates rearrangement of the edges and activities within the RC. To do this, the start node makes use of the following three pieces of data, which were gathered during the search used to identify the RC members.

(a) The RC membership list

(b) The members' SSSP distances

(c) The members' distance graph edges

The data collected during the search includes all distance graph edges for all member nodes. However, we require only those distance graph edges that connect member nodes to non-member nodes, as only these edges must be rearranged.

For example, the search from node 29 visits nodes 27, 28, 29 and 30, so the membership list contains these four node IDs. The SSSP distances for these four nodes are -1, -1, 0 and 0 respectively. Fig. 4-24 shows the distance graph for the example plan, with RCs circled and SSSP distances for each node. Node 27 has four distance graph edges, of which only two are connected to nodes outside the RC. These are an edge of length +INF from node 3, and an edge of length 0 to node 3. Nodes 28 and 29 do not have any distance graph edges connected to nodes outside the RC. Node 30 has four distance graph edges, of which only two are connected to nodes outside the RC. These are an edge of length 0 from node 31, and an edge of length 1 to node 31. The list of distance graph edges used by node 29 on completion of the search therefore consists of these four edges. In all that follows, we represent a directed edge from node *start* to node *end* with length $l$ as $< start, end, l >$. The distance graph edge list is therefore $< 3, 27, +INF >$, $< 27, 3, 0 >$, $< 30, 31, 1 >$, $< 31, 30, 0 >$.

The RC leader must be the first node in the RC to be executed, so the start node of the search assigns as RC leader the member node with minimum SSSP distance from the phantom node. In this example node 29 identifies nodes 27 and 28 as those with the minimum SSSP distance and (arbitrarily) chooses node 28 as the RC leader. RC leaders are shown highlighted in Fig. 4-24.

To initiate the remainder of the RC rearrangement process, the start node of the search sends a `RC_SET_LEADER` message to all members of the RC. This message includes the RC membership list and their SSSP distances and distance graph edges, as well as the ID of the RC leader.

Part of the rearrangement process is the rerouting of the RC members' distance graph edges to the RC leader (line 12). The RC leader records the modified edges when it receives the `RC_SET_LEADER` message from the start node, as described below. However, in the distributed framework, the nodes at both ends of an edge maintain a record of that edges. We must therefore update the record of the edge maintained by the node outside the RC to reflect the modification to the edge length. To achieve this, the start node sends `RC_UPDATE` messages to all nodes at the far end of the distance graph edges that will be relocated. A message is sent for each edge to be rearranged. For example, in the example plan, node 29, the start node of the search for the RC consisting of nodes 27 through 30, sends four `RC_UPDATE` messages, one for each of the distance graph edges to non-member nodes that it records from the RC members.

The start node for each RC assigns the RC leader, as described above, so can determine the modified parameters for each edge that will be relocated. Each `RC_UPDATE` message includes two parameters, the edge to be relocated and the relocated edge. On receipt of a `RC_UPDATE` message, a node confirms that it has a record of the edge to be relocated, deletes its record of the edge, and records the relocated edge. For example, the distance graph edge $< 30, 31, 1 >$ will be relocated to connect node 31 and the RC leader, node 28. The SSSP distance of node 30 relative to node 28 is 1, so the length of the edge is increased by 1 by relocation, giving $< 28, 31, 2 >$ as the relocated edge. Similarly, the distance graph edge $< 31, 30, 0 >$ will be relocated to connect node 31 and node 28. The length of the edge is decreased by 1 by relocation, giving $< 31, 28, -1 >$

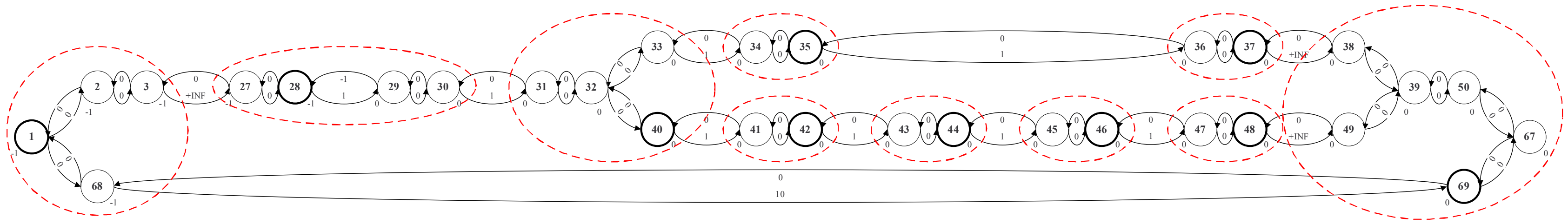


Figure 4-24: This figure shows the distance graph for the example plan, with RCs circled, RC leaders highlighted and SSSP distances for each node.

as the relocated edge. Therefore, node 25 sends 2 `RC_UPDATE` messages to node 31. The first has parameters $< 30, 31, 1 >$ and $< 28, 31, 2 >$, the second has parameters $< 31, 30, 0 >$ and $< 31, 28, -1 >$.

2. **Collapse zero-related components to a single node.**

   On receipt of the `RC_SET_LEADER` message from the start node of the search, each member of the RC takes steps to rearrange the RC. The first step is to collapse each Zero-Related (ZR) component to a single node. This is required for the correct operation of the traversals used to identify non-dominated edges and involves replacing each ZR component with a single node that owns the start and end activities for all members of the ZR component.

   Note that the members of a ZR component must be executed simultaneously, but this can not be realized in practice if the nodes are assigned to different agents. Therefore, we must check that all ZR member nodes belong to the same agent, as discussed in Section 3.2.

   First, each node determines which RC member nodes are also members of its ZR component. It does this by searching the lists of RC member nodes and their SSSP distances received with the `RC_SET_LEADER` message for nodes with the same SSSP distances. Once it has obtained this list, it selects a node as the ZR component leader. The choice is arbitrary, but all nodes must choose consistently, so we use the node with the highest ID. For example, in the RC consisting of nodes 27 through 30 in the example plan, nodes 29 and 30 determine that they belong to a ZR component and select node 30 as the ZR leader.

   If a node is not a ZR component leader, it collapses with its leader (lines 8-10). To do this, it sends a `ZR_COLLAPSE` message to its ZR component leader, passing with it its activity start and end activities. For example, node 29 passes a `ZR_COLLAPSE` message to node 30. Such a node has no further involvement in the D-Kirk algorithm, so terminates execution and can be removed from the plan graph.

If a node is the ZR component leader, it determines how many `ZR_COLLAPSE` messages it should receive and waits for them to arrive. On receipt of each message, it adds the activity start and end events to its list of events. It also removes all ZR component non-leader nodes from the list of RC members, as these nodes play no further part in the D-Kirk algorithm. Fig. 4-25 shows the example distance graph with ZR component non-leaders shown dashed. Note that edges to these nodes still exist, because edges have not yet been relocated.

Note that a node which is not zero-related to any nodes in the RC determines that it is a member of a singleton ZR component in which it is the leader. It therefore does not expect to receive any `ZR_COLLAPSE` messages and is unaffected by the ZR component collapsing process.

3. **Form the doubly-linked chain of edges between RC member nodes.**
   The next step on receipt of the `RC_SET_LEADER` message is to create the doubly linked chain of edges that joins members of the RC (line 11). Each node takes the RC membership list, with ZR component non-leaders removed, and sorts it in increasing SSSP distance from the phantom node. It can then determine its neighbors in the sequence and record edges of an appropriate length. In our example, the membership list, sorted by increasing SSSP distance, is $< 28, 30 >$. Node 28 identifies that its neighbor in the RC is node 30, with SSSP distance 0. Node 28 has a SSSP distance of -1, so it records incoming and outgoing edges of length -1 and 1 respectively to node 30. It also deletes its record of all other edges (line 13). All of the other nodes in the RC do likewise. Fig. 4-26 shows the distance graph for the example plan, with the doubly linked chains of edges complete.

4. **Relocate all edges between member nodes and non-member nodes to the RC leader, adjusting edge lengths accordingly.**
   If the receiving node is the RC leader, it must create edges between itself and nodes outside the RC to replace those edges that are connected to RC member nodes (line 12). This edge relocation is required so that all of the temporal

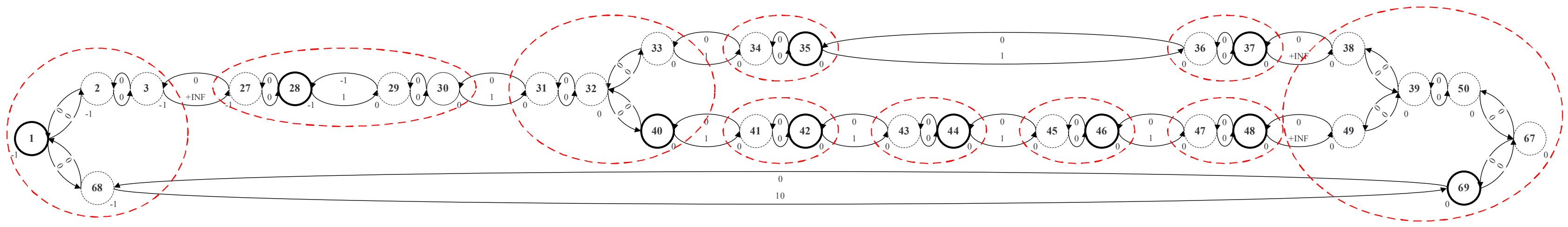


Figure 4-25: This figure shows the distance graph for the example plan, with zero-related component non-leaders shown dashed.

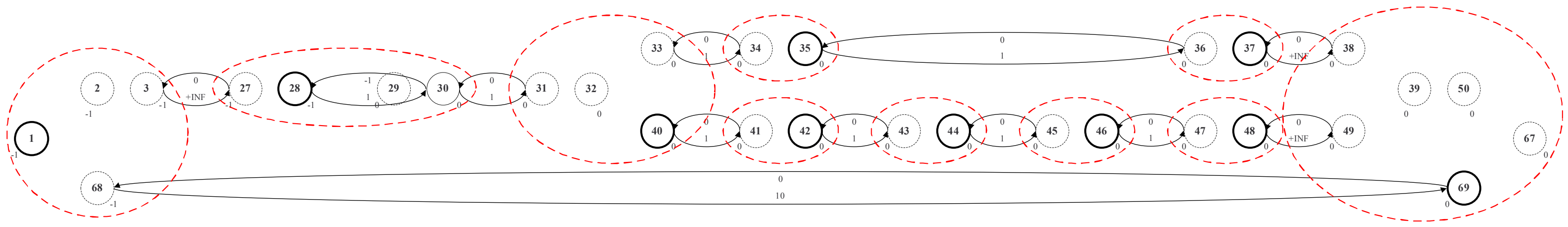


Figure 4-26: This figure shows the distance graph for the example plan, with the doubly linked chains of edges complete.

constraints between the RC and the rest of the plan are expressed through the RC leader, which represents the RC for the remainder of the reformulation process.

The RC leader uses the list of distance graph edges from the RC members to create the relocated edges. Each edge length is modified to reflect the relocation to the RC leader. This is done using the relative SSSP distances and is discussed in Section 2.3.

For example, when node 28 receives the `RC_SET_LEADER` from node 29, it recognizes that it is the RC leader and relocates the RC's four distance graph edges to non-member nodes. The length of edge $< 3, 27, +INF >$ is unchanged when it is relocated because nodes 27 and 28 have the same SSSP distance: the relocated edge is $< 3, 28, +INF >$. Likewise, edge $< 27, 3, 0 >$ becomes $< 28, 3, 0 >$. The length of edge $< 30, 31, 1 >$ is increased by 1, the difference between the SSSP distances of nodes 30 and 28, when it is relocated: the relocated edge is $< 28, 31, 2 >$. Similarly, the length of edge $< 31, 30, 0 >$ is decreased by 1, so the relocated edge is $< 31, 28, -1 >$. The updated edges for this RC are shown in Fig. 4-27.

If the node receiving the `RC_SET_LEADER` message is not the RC leader, it needs to take no action regarding the relocation of edges, since it deleted its record of the relocated edges when it was visited in the search to identify RC members.

Once the RC identification and rearrangement processes are complete for all nodes, RC processing is complete. The fully processed distance graph for the example plan is shown in Fig. 4-28.

The RC extraction process begins at the first node in the RPO. By definition, this is the last node to complete the RPO extraction phase, so RC extraction does not begin until RPO extraction is complete. Since in the presence of the phantom node the overall start node is not necessarily the first in the RPO, the RC processing phase is initiated by a message from the overall start node to the first node in the RPO. The complexity of the RC processing phase of the distributed reformulation algorithm is

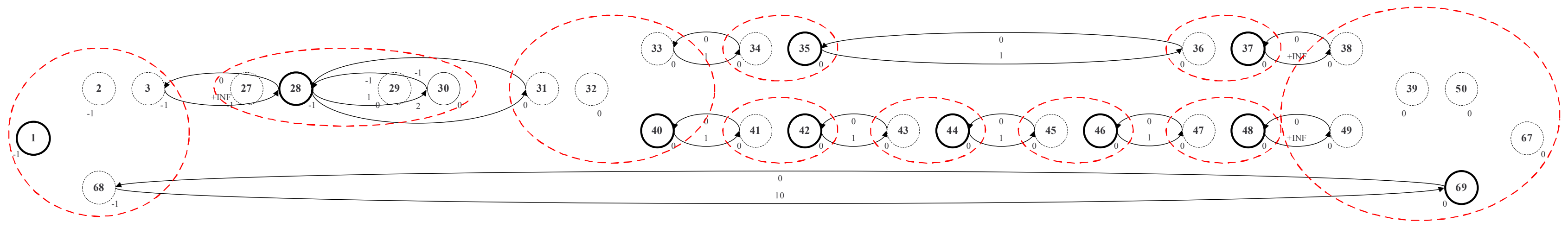


Figure 4-27: This figure shows the distance graph for the example plan after relocation of edges for the rigid component consisting of nodes 27 through 30.

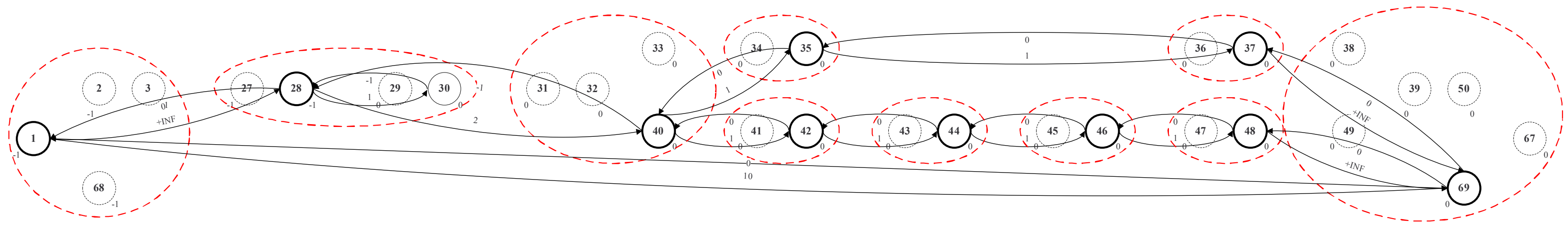


Figure 4-28: This figure shows the distance graph for the example plan with RC processing complete.

dominated by the extraction of the members of the RC and by rearranging edges to the leader, giving a computational complexity of $O(e)$, where $e$ is the number of edges per node.

### 4.3.3 Identification of Non-Dominated Edges

Once the rigid components in the plan have been processed, we can proceed with the main part of the fast reformulation; identification of non-dominated edges. Recall that dispatchable execution requires that we reformulate the plan to a dispatchable graph, to reduce computational load during dispatching. A dispatchable graph makes explicit temporal constraints in the plan such that only local propagation of timing information is required at dispatch time. The All Pairs Shortest Path (APSP) graph for the plans distance graph is a dispatchable graph. However, we require a *Minimal* Dispatchable Graph (MDG), which is a dispatchable graph that contains the minimum number of edges. To obtain the MDG, we test for whether each edge in the APSP graph is *dominated* by another APSP edge. Only non-dominated edges belong in the MDG.

The fast reformulation algorithm forms the MDG without forming the APSP graph. It achieves this by performing a series of traversals of the plan graph. A traversal is conducted from each RC leader in the plan and determines which of the APSP edges from that node belong in the MDG. In a traversal started at node $A$, the dominance test applied as we traverse node $B$ determines whether or not the APSP edge $AB$ is a member of the MDG. The fast reformulation algorithm obtains its efficiency through the fact that the dominance test at each node requires only local computation. This requires that the traversal visits a node's successors in the distance graph before visiting the node itself. The Reverse Post Order (RPO) for Depth First Search (DFS) on the predecessor graph rooted at the start node of the traversal has this property, so we use this order for the traversal.

Since the RPO is inherently ordered, this part of the algorithm can not be conducted in parallel and is entirely serial. A given node must complete its part in RC processing before it can begin the dominance test procedure. However, we do not need all nodes to have completed the RC phase before we start the dominance tests, so these two phases run concurrently, waiting for nodes to complete RC processing where required.

Pseudo-code for this phase of the distributed reformulation algorithm is shown in Fig. 4-29. The algorithm runs on the RC leader nodes in the plan, including singleton RCs. At the highest level, it is a loop over each RC leader, corresponding to the fact that we start a traversal from each of these nodes. The order with which we select start nodes is unimportant.

```
1: for Each RC leader do
2:     Select this node as the start node of the traversal
3:     Identify the nodes involved in this traversal by DFS from the traversal start
       node
4:     Compute SSSP distances from the traversal start node for the graph of RC
       leaders using synchronous distributed Bellman-Ford
5:     Form the predecessor graph for the traversal start node using SSSP distances
6:     Perform DFS on predecessor graph from the traversal start node and record
       RPO
7:     Traverse graph in RPO from the traversal start node
8:     for Each node traversed do
9:         Use minimum and non-positive data to apply dominance tests
10:        if APSP edge is non-dominated then
11:            Record APSP edge as member of MDG
12:        end if
13:        Update minimum and non-positive data
14:        Propagate minimum and non-positive data to successors
15:    end for
16:    Record non-dominated APSP edges as members of MDG
17: end for
```

Figure 4-29: This figure shows the pseudo-code for the dominance test traversal phase of the distributed reformulation algorithm.

We describe the operation of the algorithm in terms of the following steps. Line numbers refer to the pseudo-code in Fig. 4-29. The steps are conducted for every traversal start node in turn (line 2). For the purposes of illustration, we consider the traversal from node 1 of our example plan.

1. **Determine which nodes will be involved in this traversal**

   The traversal visits every RC leader node which is a member of the predecessor graph rooted at the start node of the traversal. In general, some RC leaders are not members of the predecessor graph because they are not *reachable* from the start node of the traversal. A node is reachable if there exists a pathway of finite length along the directed edges of the network from the start node to the node in question, i.e. the node's SSSP distance is finite. This means that a particular node may be unreachable either because there are no edges that connect it to the start node, or because those edges are of infinite length. Therefore, APSP edges from the start node to nodes which are not members of the predecessor graph have infinite length, so can not possibly be part of the MDG and we do not need to use the dominance tests to determine whether or not they belong in the MDG.

Note that when forming the predecessor graph for the purpose of processing RCs, all nodes are reachable because we add zero length edges from the phantom node to every node in the network.

We use the synchronous distributed Bellman-Ford algorithm to calculate the SSSP distances of each node in order to form the predecessor graph. To run this algorithm efficiently, we must identify the set of nodes that are reachable from the start node. This is because a node identifies the nodes from which it expects to receive update messages, and uses this to synchronize itself with the other nodes in the network.

Therefore we conduct a DFS from the start node of the traversal to identify the reachable nodes (line 3). The set of reachable nodes is simply the visited list once the DFS terminates at the start node. We then pass this list to every reachable node so that it can use it in the Bellman-Ford algorithm as mentioned above. In the example plan every node is reachable for the traversal from node 1. Indeed for this plan, all nodes are reachable from every other node.

2. **Calculate the SSSP distance of every node and construct the predecessor graph.**

   We use the extended synchronized distributed Bellman-Ford algorithm to calculate the SSSP distance from the start node and to form the predecessor graph (lines 4 and 5). The computational complexity per traversal is $O(Ne)$, where $N$ is the number of nodes in the plan and $e$ is the number of edges per node. Fig. 4-30 shows the SSSP distances and predecessor graph from node 1 for the example plan. Note that ZR component non leaders have been removed and the graph redrawn for clarity.

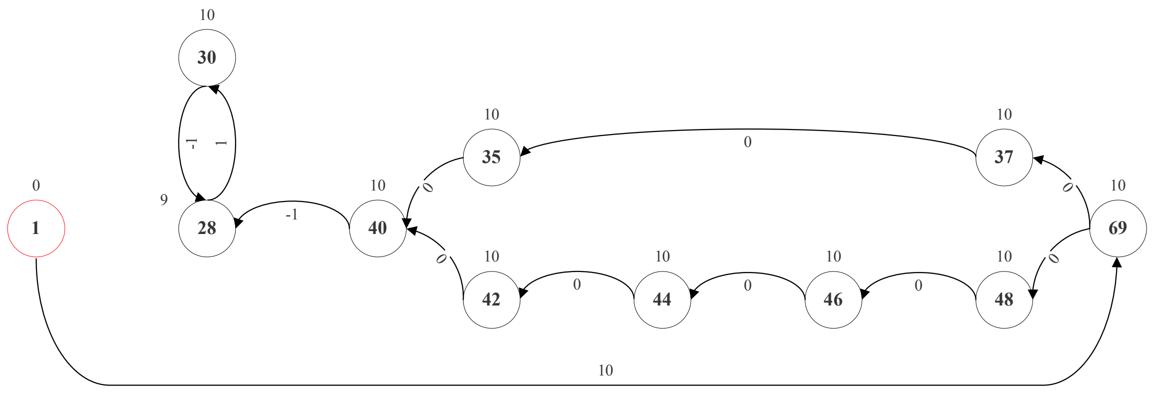


Figure 4-30: This figure shows the SSSP distances and predecessor graph from node 1 for the example plan.

3. **Extract reverse post order.**

   We extract the RPO as described previously (line 6). The computational complexity per traversal is $O(e)$, where $e$ is the number of edges per node. Fig. 4-31 shows the predecessor graph from node 1 for the example plan, with the ID of the last posted node for DFS from node 1 recorded for each node.

4. **Traverse and apply dominance tests.**

   We conduct the traversal in the order given by the RPO (line 7). The traversal start node, the first in the RPO, is the last node to complete the RPO extraction phase, so the traversal does not begin until RPO extraction is complete.

   The dominance test at each node requires two pieces of information, *minimum* and *non-positive*, as detailed in Section 2.3 (line 9). If the edge is not dominated, it is recorded locally (line 11). It is also recorded in a list which is passed back to the start node of the traversal for recording there (line 16).

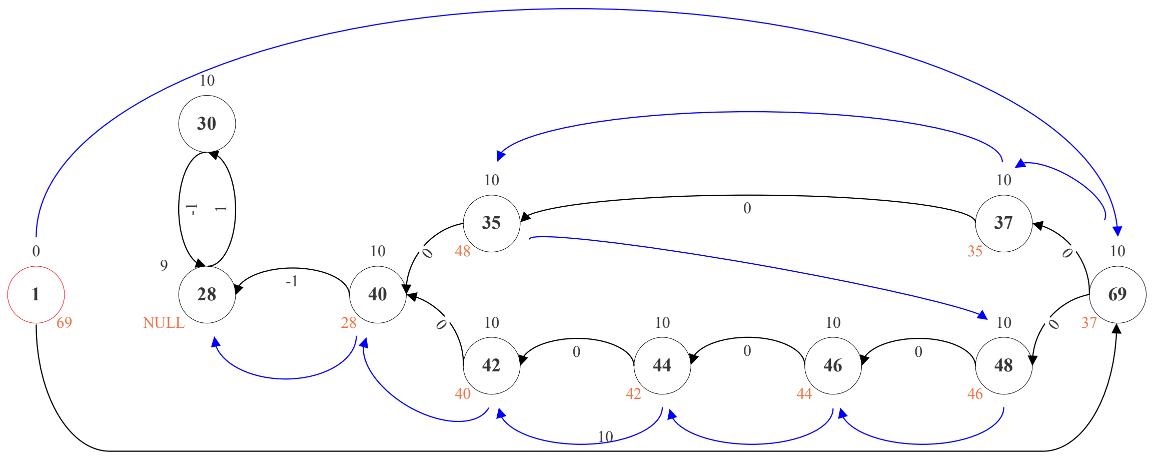


Figure 4-31: This figure shows the predecessor graph from node 1 for the example plan, with the ID of the last posted node for DFS from node 1 recorded for each node.

Once these pieces of data have been used by the node they are updated (line 13) and propagated to the node's successors (line 14), as these are visited after the node itself. We propagate the data by sending `MDG_DATA` messages to the successor nodes, with the values of *minimum* and *non-positive* included as message parameters. Once the data has been propagated, the node then instructs the next node in the RPO to conduct its dominance test, by sending a `MDG_EXTRACT` message to the node in question.

For example, consider the traversal from node 1 in the example plan. Recall that the start node does not conduct a dominance test, so the first step is the start node sending a `MDG_DATA` message to its successor, node 69. Recall also that the definitions of *minimum* and *non-positive* are such that we exclude the start node, so the start node uses the default initial values of +INF and *false* respectively. The start node then sends a `MDG_EXTRACT` message to the next node in the RPO, node 69.

Node 69 has a SSSP distance from the start node of 10, so the APSP edge from node 1 to node 69 has length 10. This is non-negative, so we test for upper-dominance. The value of *minimum* is +INF, which is not less than 10, so the edge is not upper-dominated. Node 69 therefore records an MDG edge of length 10 from node 1. It then updates the values of *minimum* and *non-positive* using its own SSSP distance of 10, to give values of 10 and *false* respectively. It then sends `MDG_DATA` messages to its successors, nodes 37 and 48, with these values. Finally, it sends a `MDG_EXTRACT` message to the next in the RPO, node 37, to instruct it to conduct its dominance test. The network at this stage in the traversal is shown in Fig. 4-32. MDG edges are shown in red.

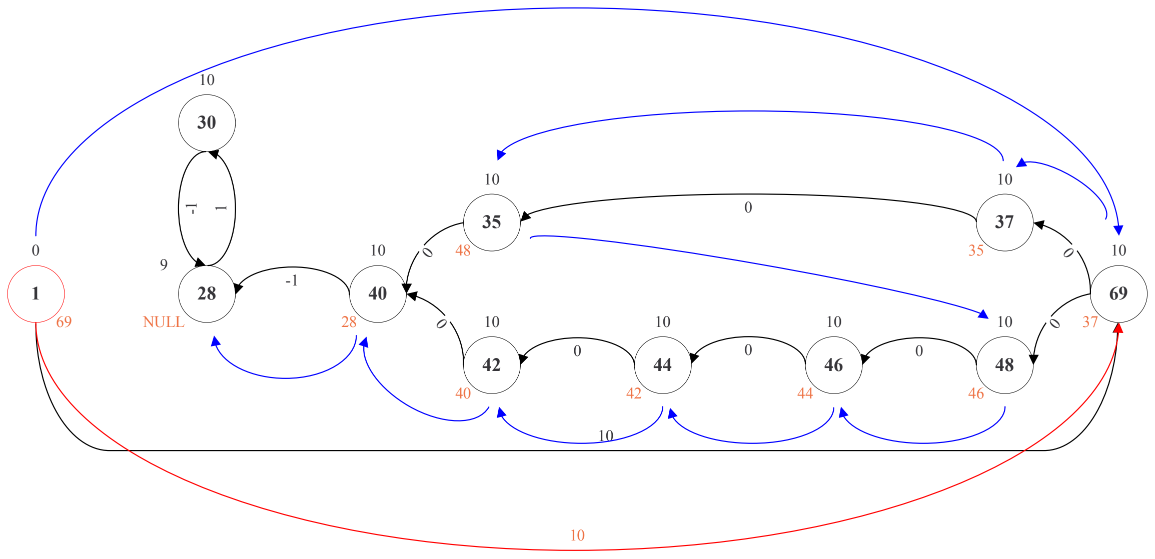


Figure 4-32: This figure shows the predecessor graph from node 1 for the example plan, at the point where the traversal from node 1 has just passed node 69. MDG edges are shown in red.

The complexity is dominated by the need to propagate the data used in the dominance tests to all predecessors, giving a computational complexity per traversal of $O(e)$, where $e$ is the number of edges per node.

Once a traversal is complete, the start node of the traversal uses a message to initiate a traversal from the next node in the graph. In this way, successive traversals are conducted serially.

Once a traversal has been conducted from all RC leader nodes, the MDG is complete. The complete MDG for the example plan is shown in Fig. 4-33. Edges from node 1 are shown as stepped lines for clarity.

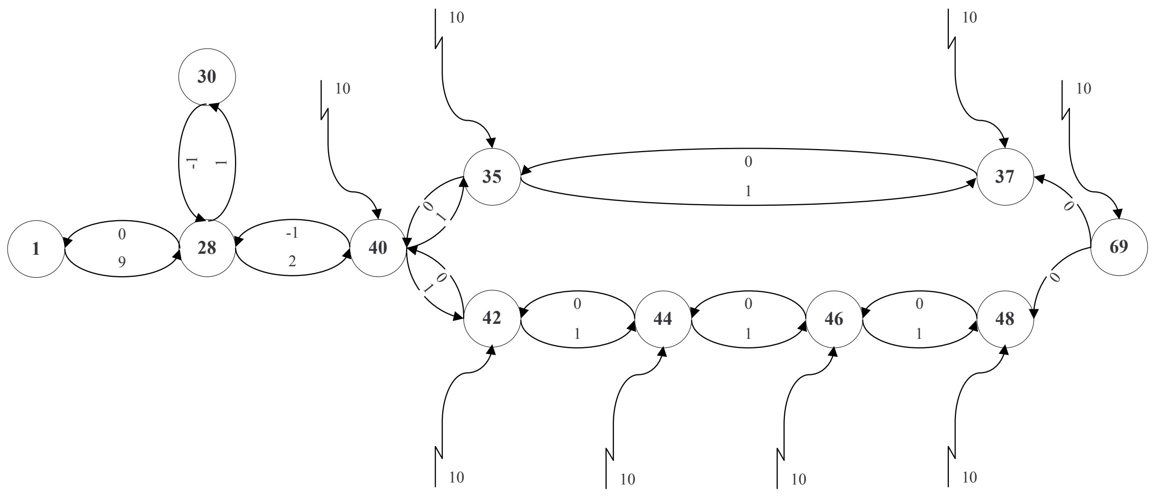


Figure 4-33: This figure shows the complete MDG for the example plan. Edges from node 1 are shown as stepped lines for clarity.

### 4.3.4 Complexity

Using the complexities of each stage described above, we obtain a worst case overall computational complexity of $O(N^2e + Ne)$ per node for the distributed reformulation algorithm, where $N$ is the number of nodes in the plan and $e$ is the number of edges per node. This is due to the Bellman-Ford SSSP calculations, each of which has complexity $O(Ne)$. One calculation is used to obtain the predecessor graph for RC processing and a further $N$ to form the predecessor graphs for the dominance test traversals. Note that the computation to be performed on receipt of each message is simple, hence the computational complexity is of the same order as the message complexity.

In the centralized case, the centralized Bellman-Ford algorithm has complexity $O(NE)$, where $E$ is the total number of edges in the plan, giving an overall computational complexity of $O(N^2E)$. For typical plans generated from the Reactive Model-Based Programming Language (RMPL), $E = O(Ne)$. Therefore the distributed processing provides a typical improvement of a factor of $N$ per node or $A$ per agent, where there are $A$ agents involved in executing the plan and each is responsible for $\frac{N}{A}$ nodes.

## 4.4 Conclusion

This chapter presented the details of the distributed fast reformulation algorithm used by D-Kirk. Reformulation takes a temporally flexible plan and produces a Minimal Dispatchable Graph (MDG). A dispatchable graph is one in which temporal constraints are made explicit, so only local propagation of timing information is required at dispatch time, thus reducing the computational load on the dispatcher. A minimal dispatchable graph has the minimum number of edges from the set of all possible dispatchable graphs. The fast reformulation reduces computational and space requirements by calculating the MDG edges without forming the entire ASPS graph. Our distributed version of this algorithm distributes data and processing between all of the agents involved in the plan and thereby reduces the computational complexity for a given agent.

# Chapter 5

# Plan Dispatching

## 5.1 Introduction

The D-Kirk executive uses the methods of dispatchable execution to allow the executive to respond to uncertain activity durations at execution time. Dispatchable execution maintains temporal flexibility in the plan at planning time so that it can be exploited at execution time. However, exploiting this flexibility requires additional computation at execution time. Dispatchable execution therefore uses a two-stage execution strategy to minimize the computation that must be performed in real-time. First, reformulation transforms the plan into a form which minimizes the computation required during dispatching. Second, dispatching executes the activities in the plan. This chapter describes the distributed dispatching algorithm used by D-Kirk.

## 5.2 Distributed Dispatching

D-Kirk represents a temporally flexible plan as a Simple Temporal Network (STN). Reformulation takes the STN and produces a Minimal Dispatchable Graph (MDG). A dispatchable graph makes explicit the temporal relationships in the plan, such that only local propagation of timing information is required at dispatch time. A minimal dispatchable graph has the minimum number of edges from the set of dispatchable graphs.

Recall that in the STN representation of a plan, nodes represent events and edges represent temporal constraints. Activities are represented as labels on the edges in the STN. However, during plan distribution we modify the representation of an activity, such that it is instead described in terms of a pair of events. The activity start event occurs at the node at the start of the edge initially representing the activity, and the activity end event occurs at the node at the end of the edge.

The task of the dispatcher is to execute the activities in the plan. Executing an STN consists of assigning execution times to each node. The dispatcher therefore executes the plan by assigning an execution time to each node. When a node is executed, its activity start and end events are executed on the agent responsible for the activity in question.

As discussed in Chapter 4, the distributed reformulation algorithm does not guarantee correct execution of a plan containing uncontrollable activities. However, the dispatching algorithm by Muscettola et. al. [20], on which the distributed dispatching algorithm in D-Kirk is based, does allow such activities to be executed. Therefore, only the reformulation algorithm must be modified to guarantee correct execution of uncontrollable activities, as discussed in Section 7.5.

### 5.2.1 Algorithm Structure

We base the D-Kirk distributed dispatching algorithm on the dispatching algorithm by Muscettola et. al. [20]. We distribute the dispatching algorithm over all nodes and nodes communicate using a message passing scheme. Each node runs the dispatching algorithm independently and concurrently and is is responsible for dispatching itself.

As is the case with the distributed reformulation algorithm described in Chapter 4, the distributed dispatching algorithm is event driven, where an event is the receipt of a message. We therefore use a state machine approach to structure the algorithm, which allows each node to track its progress and provides robustness to delays in message delivery, as described previously. This allows the algorithm to function correctly in terms of state transitions without requiring any synchronization between processors.

Synchronization *is* required, however, to ensure that tasks are executed at precisely the correct times. For this we assume a synchronous execution model. In particular, we assume that each processor has a synchronized clock. The task of achieving this synchronization is not trivial, but is beyond the scope of this work [15]. Note, however, that approximate synchronization, to within the delivery time of a single message, is sufficient for many practical applications.

During dispatching, each node maintains its *execution window*. A node's execution window is the time window in which it could be executed, given the temporal constraints in the plan and the execution times of the nodes that have already been executed. A node is *alive* when the current time falls within its execution window. When the dispatching algorithm begins, all execution windows are initialized to $[0, +INF]$, which places no constraint on the node's execution time.

When a node is executed, it informs its neighbor nodes of the fact by sending an EXECUTED message. A node receiving such a messages uses the information to update its execution window and to determine when it is *enabled*. A node is enabled when all nodes that must execute before it have been executed. Each node also keeps track of the status of any uncontrollable activities which for which it owns end events.

A node must be enabled, alive and all of its uncontrollable activity end events must have completed before it can execute. We wait for enablement before we check for whether a node is alive because enablement always increases the lower bound on a node's execution window. As execution of the plan progresses, a node tracks its progress with regard to these three requirements and uses its clock to determine when it can execute.

The state transition diagram for the distributed dispatching algorithm is shown in Fig. 5-1 and the pseudo-code is in Fig. 5-2. There are four main phases to the algorithm; waiting to start, waiting to become enabled, waiting to become alive and executing. These phases are explained in the following subsections.

### 5.2.2 Waiting to Start

The dispatching process is triggered by a DISPATCH message sent to the start node of the plan from an external source. The start node then informs all of the other nodes

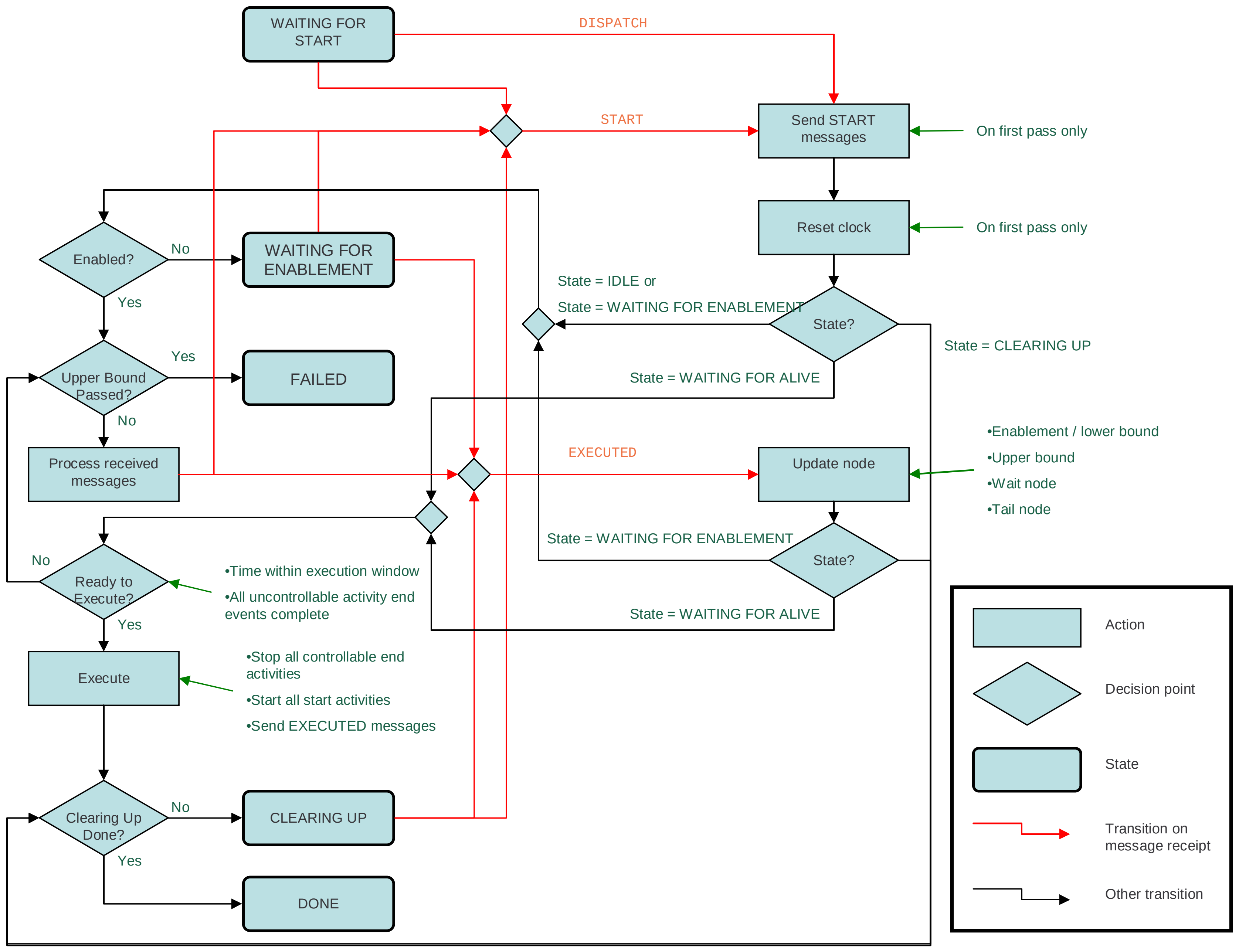


Figure 5-1: This figure shows a state transition diagram for the distributed dispatching algorithm.

```
1: Wait for START or DISPATCH message
2: Reset clock
3: Send START message to neighboring nodes
4: while All nodes on outgoing non-positive edges have not executed do
5:    Process received EXECUTED messages
6: end while
7: while No error do
8:    if Current time has exceeded upper time bound then
9:       Execution failure
10:   end if
11:   Process received EXECUTED messages
12:   if current time is within execution window AND All uncontrollable end activ-
      ities are complete then
13:      Break
14:   end if
15: end while
16: Stop all controllable end activities
17: Start all start activities
18: Inform neighbor nodes that node has executed
```

Figure 5-2: This figure shows the pseudo-code for the distributed dispatching algorithm.

in the plan that the dispatching process has begun, so that they can synchronize their clocks. It achieves this by sending START messages to all of its neighbor nodes. On receipt of a START message, a node resets its clock (line 2) and forwards the message to all of its neighbors (line 3).

### 5.2.3 Waiting to Become Enabled

In the second stage of the distributed dispatching algorithm, a node waits to become *enabled*. A node is enabled when all nodes that must execute before it have been executed. In terms of the MDG, nodes that must execute before a given node are those to which the node is connected by outgoing non-positive edges (line 4).

The node forms a list of the nodes to which it is connected by outgoing non-positive edges in the MDG. The node waits for and processes EXECUTED messages (line 5) until it has received messages from *all* nodes on the list. For example, Fig. 5-3 shows the MDG for the example plan at time 5 during dispatching. Executed nodes are shaded dark gray and enabled nodes are shaded light gray. Each node is labeled with its execution window in the case of a node which has yet to be executed, or its execution time in the case of an executed node. Node 69 in the example plan is connected to two nodes by outgoing non-positive edges in the MDG; nodes 37 and 48. Both of these nodes must be executed in order for node 69 to become enabled. Node 69 therefore waits to receive EXECUTED messages from both of these nodes.

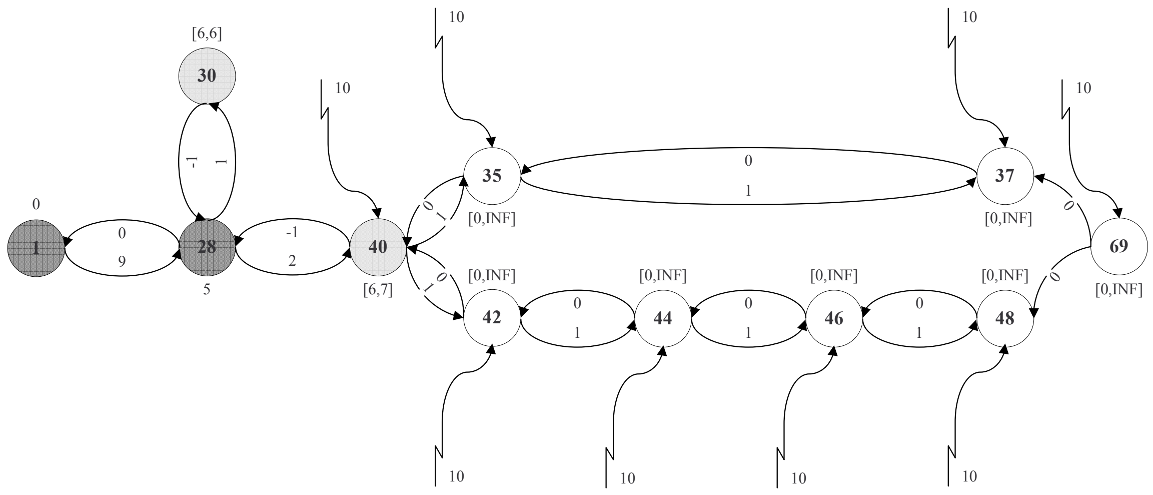


Figure 5-3: This figure shows the MDG for the example plan at time 5 during dispatching. Executed nodes are shaded dark gray and enabled nodes are shaded light gray. Each node is labeled with its execution window in the case of a node which has yet to be executed, or its execution time in the case of an executed node. Some edges from node 1 are shown as stepped lines for clarity.

The received EXECUTED messages are also used to update the node's execution window. For this reason a node processes all received EXECUTED messages while waiting to become enabled. Each EXECUTED message carries the execution time of the sending node as a parameter. A node receiving such a message uses the execution time, together with the temporal constraint between it and the sender of the message, to update its execution window.

In particular, when a node receives an EXECUTED message from a node to which it has an outgoing non-positive edge, it uses the execution time parameter to update the lower bound on its execution window. The updated lower bound is given by $t - d$, where $t$ is the received execution time and $d$ is the length of the connecting edge.

In the example in Fig. 5-3, node 28 has just executed at time 5 and has sent EXECUTED messages to nodes 30 and 40 with parameter value 5. Node 30 has used its outgoing edge of length -1 to node 28 to update its lower execution bound from zero to 6 and node 40 has done likewise.

When a node receives an EXECUTED message from a node to which it has an incoming non-negative edge, it uses the execution time parameter to update the upper bound on its execution window. The updated upper bound is given by $t + d$.

In the example plan in Fig. 5-3, node 28 has just sent EXECUTED messages to nodes 30 and 40 with parameter value 5. Node 30 has used its incoming edge of length 1 from node 28 to update its upper execution bound from $+INF$ to 6. Node 40 has used its incoming edge of length 2 from node 28 to update its upper execution bound from $+INF$ to 7.

### 5.2.4 Waiting to Become Alive

Once enabled, a node must wait to become *alive* before it can execute. A node is alive when the current time is such that it could execute now without violating any temporal constraints. This occurs when the current time is within its execution window (line 12). Note that a node's execution window is initialized to $[0, +INF]$ but may have been updated before this stage of the algorithm is reached. For example, at time 5, node 30 in the example plan is enabled, but not alive, since its execution window is $[6, 6]$, as shown in Fig. 5-3.

A *contingent* node, which owns end events for uncontrollable activities, must also wait for all such uncontrollable activities to complete (line 12). By definition, the executive has no control over the duration of uncontrollable activities, rather their duration dictates the execution time of their end node. Therefore the dispatcher must execute the node at exactly the moment at which the uncontrollable activities complete. If the node is not enabled and alive at this time, execution of the plan fails.

While waiting to become alive and for uncontrollable activities to complete, the node checks that the current time does not exceed the upper bound of the execution window (line 8), else the plan execution fails (line 9). Also, the node continues to respond to received EXECUTED messages (line 11), which may further reduce its execution window.

### 5.2.5 Executing

Once a node is enabled and alive, and in the case of a contingent node, all of the uncontrollable activities for which it owns the end event have completed, it can execute. A contingent node must execute immediately, as described above. A *controllable* node, which does not own end events for any uncontrollable activities, can execute at any time between the time at which the above become true and the upper bound of its execution window. In D-Kirk we use a minimum-time dispatching policy, where nodes are executed as soon as possible. Therefore we do not need to distinguish between contingent and controllable nodes in this regard.

When executing a node, the dispatcher first executes the end events for controllable activities (line 16). It then executes all of the node's activity start events (line 17). Finally, the node sends EXECUTED messages to inform neighbor nodes that it has executed (line 18). EXECUTED messages are sent to nodes to which the node is connected by outgoing non-negative edges (for execution window upper bound updates) and by incoming non-positive edges (for execution window lower bound updates and enablement updates).

### 5.2.6 Complexity

The number of messages sent by each node during the distributed reformulation algorithm is determined by the number of edges connected to it in the MDG. The message complexity is therefore $O(e')$, where $e'$ is the average number of MDG edges connected to each node. If $A$ agents are involved in the plan, and each agent owns $\frac{N}{A}$ nodes on average, then the message complexity per agent is $O\left(\frac{Ne'}{A}\right)$. In the centralized case, the lead node must send messages to every other node in the plan to instruct them to execute their activities, giving a peak message complexity of $O(N)$, where $N$ is the number of nodes in the plan. Therefore, compared to the centralized case, D-Kirk reduces the number of messages at dispatch time, when we must operate in real time and are most susceptible to communication delays, by a factor of $\frac{A}{e'}$. $e'$ is typically a small constant determined by the branching factor of the plan. Furthermore, since the computational complexity is directly proportional to the number of messages received, D-Kirk improves this too.

## 5.3 Conclusion

This chapter presented the distributed dispatching algorithm used by D-Kirk to execute a temporally flexible plan. The dispatcher takes a plan in the form of a Minimal Dispatchable Graph, which is produced by the reformulator described in Chapter 4. The dispatcher executes the activities in the plan, using the temporal flexibility in the plan to respond to uncertain activity durations.

The distributed nature of the algorithm reduces the peak message complexity per node and avoids the communication bottleneck present at the master node of a centralized system. This provides the executive with robustness to communication latency.

# Chapter 6

# Plan Extraction

## 6.1 Introduction

Chapters 4 and 5 described the methods used by D-Kirk to robustly execute a temporally flexible plan. This chapter extends D-Kirk to handle contingent plans. Contingent plans provide robustness to plan failure by allowing us to specify redundant methods. We achieve this by adding a plan extraction phase to D-Kirk.

## 6.2 Distributed Plan Extraction

Plan extraction takes place after plan distribution and before plan dispatching. The input to D-kirk is a temporally flexible plan with contingencies in the form of a Temporal Plan Network (TPN). Once the TPN has been distributed between the agents involved with the plan, the plan extraction phase selects a single method for each contingency such that the time constraints of the entire plan are satisfied. The selected temporally consistent plan is then reformulated and dispatched as described in Chapters 4 and 5.

For example, Fig. 6-1 shows the TPN for the manipulator tool delivery scenario introduced in Chapter 1. This plan involves a single choice node, node 2, which introduces two functionally redundant methods. The first method is represented by the upper pathway through the TPN, where WAM0 picks up the tool from pick-up

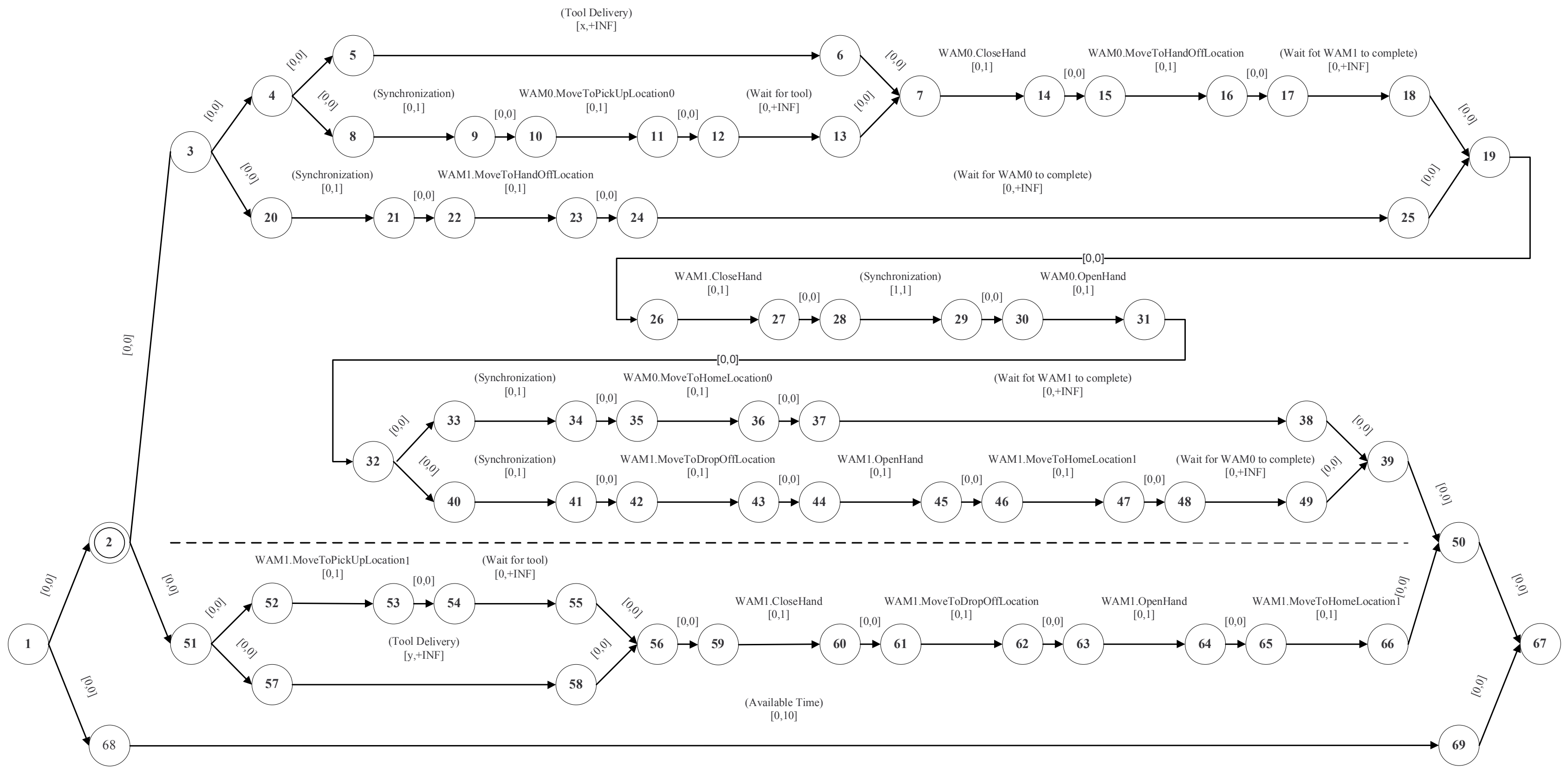


Figure 6-1: This figure shows the TPN for the tool delivery scenario introduced in Chapter 1 and was first shown in Fig. 1-2.

location 0 and passes it to WAM0 at the hand-off location. WAM1 then delivers the tool to the drop-off location and both manipulators return to their home locations. In the second method, represented by the lower pathway though the TPN, WAM1 picks up the tool from pick-up location 1 and delivers it to the hand-off location without assistance from WAM0.

Immediately prior to execution, the system obtains values for $x$ and $y$, the unknown minimum wait times for the appearance of a tool at pick-up locations 0 and 1 respectively. For this example, assume that the values are $x = 1$ and $y = 20$. This means that the lower bound on the execution time of the upper contingency, involving both manipulators, is 2, whereas that of the lower contingency, involving only WAM1, is 20. The simple temporal constraint between nodes 68 and 69 places an upper time bound of 10 on the entire plan. Therefore, the lower contingency does not satisfy all of the temporal constraints in the plan and D-Kirk selects the upper contingency, which is then reformulated and dispatched.

The objective of plan selection is to choose between functionally redundant methods to select a plan in which all time constraints are satisfied. We therefore seek a temporally consistent plan, as defined in Section 2.2. To select a plan, we make assignments to the choice variables in the choice nodes of the TPN. These variables encode which of the methods is currently selected at that choice node, as described in Section 2.5. Plan selection is therefore a restricted conditional Constraint Satisfaction Problem (CSP) [16], where the constraints are the timing constraints in the plan and the conditional element is the assignment of values to choice variables.

### 6.2.1 Algorithm Structure

D-Kirk makes use of the distributed plan extraction algorithm by Wehowksy [31] [1]. This algorithm consists of two interleaved processes: generation of candidate plans and testing of candidates for temporal consistency. The candidate plans correspond to different assignments to the choice variable at each choice node. The D-Kirk planning algorithm uses parallel, recursive, depth first search to make these assignments. This use of parallel processing is one key advantage of D-Kirk over a traditional, centralized approach. Consistency checking is implemented using the distributed Bellman-Ford Single Source Shortest Path (SSSP) algorithm, which is run on the distance graph corresponding to the portion of the TPN that represents the current candidate. Details of the distributed Bellman-Ford algorithm are given in Section 4.2. Temporal inconsistency is detected as a negative weight cycle [5], as described in Section 2.5.

The plan extraction algorithm distributes processing between all of the agents involved in the plan. Each agent is responsible for the nodes relating to the activities it will carry out. Each node operates independently and maintains the information corresponding to the temporal constraints to which it is connected. In particular, we use the distributed data structures described in Chapter 3. Nodes communicate by message passing.

The behavior of a given node depends upon its type with regard to the four RMPL operators used to form a TPN, introduced in sections 2.2 and 2.4. In particular, we classify a node as the start or end node of either an `activity`, a `parallel` subnetwork or a `choose` subnetwork. We use the start and end nodes of these three constructs because an `activity` is the fundamental unit of the TPN and the `parallel` and `choose` networks introduce additional start and end nodes, as shown in figures 2-1 and 2-11. The `sequence` operator, however, does not introduce additional start and end nodes. In this way, we handle activities, `parallel` networks and `choose` networks explicitly.

To handle `sequence` networks, the start node of each network maintains a pointer to the start node of the network which follows it in a sequence, if present. This following network is known as a successor network. In this way, each network coordinates with its successor network, such that sequences of networks are handled implicitly. We therefore refer to the three network types considered by the plan extraction algorithm as `activity-sequence`, `parallel-sequence` and `choose-sequence`. These types are shown in graph form in Fig. 6-2.

The nested RMPL operators used to construct the TPN form a hierarchy of network types. We use the hierarchy formed by the plan extraction algorithm constructs (`activity-sequence`, `parallel-sequence` and `choose-sequence`) to structure the distributed plan extraction algorithm. For example, consider the example TPN

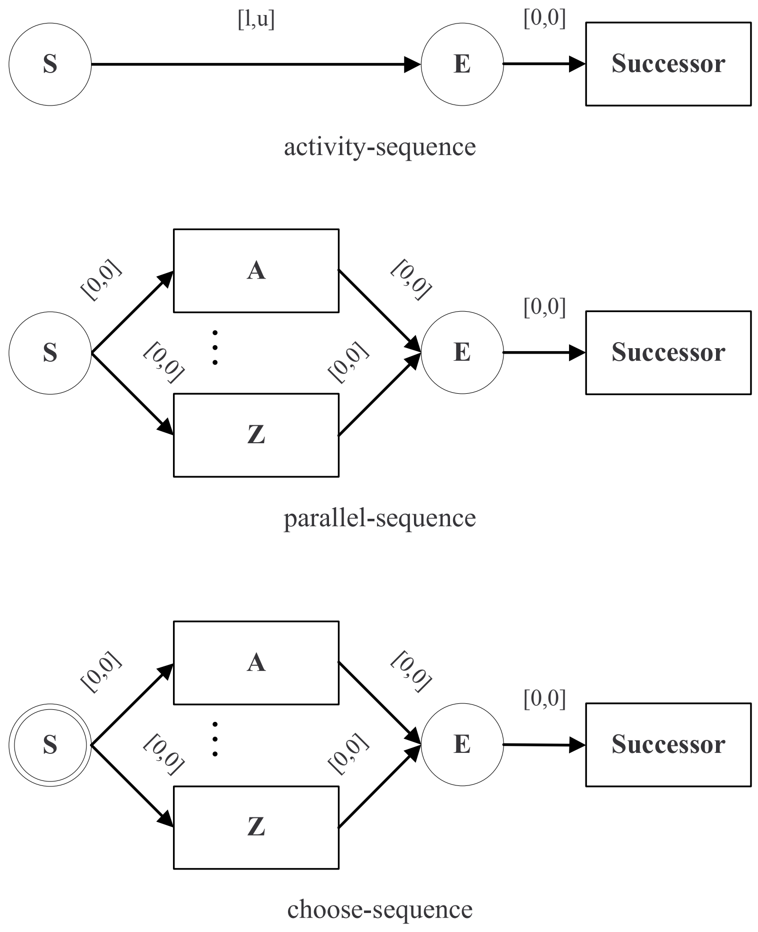


Figure 6-2: The `activity-sequence`, `parallel-sequence` and `choose-sequence` constructs used by the distributed plan extraction algorithm.

in Fig. 6-1. At the highest level, level 1, this is a `parallel-sequence` network with 2 subnetworks at level 2. The level 2 subnetwork at the bottom of the figure is the `activity-sequence` representing the available time. The level 2 subnetwork at the top of the figure is a `choose-sequence` network with 2 contingencies at level 3. The upper contingency is a sequence of subnetworks; a `parallel-sequence` subnetwork, 3 `activity-sequence` objects and a second `parallel-sequence` subnetwork, all of which are at level 3. The lower contingency is also a sequence of subnetworks; a `parallel-sequence` subnetwork and 4 `activity-sequence` objects, all of which are also at level 3.

The planning algorithm exploits the hierarchical structure of the TPN to allow parallel processing. Also, consistency checking is interleaved with candidate generation, such that D-Kirk simultaneously runs multiple instances of the distributed Bellman-Ford algorithm on isolated subsets of the TPN.

Note that while a simple temporal constraint $[l, u]$ is locally inconsistent if $l > u$, we assume that the TPN is checked prior to running D-Kirk, to ensure that all temporal constraints are locally consistent. This assumption means that only `parallel-sequence` networks can introduce temporal inconsistencies.

### 6.2.2 Candidate Plan Generation

The first phase of the distributed plan extraction algorithm is candidate plan generation. Candidate plans are generated by a parallel distributed Depth First Search (DFS) on the TPN in which choice variables are assigned systematically. D-Kirk uses the following messages for candidate plan generation.

- *FINDFIRST* instructs a network to make the initial search for a consistent set of choice variable assignments.

- *FINDNEXT* is used when a network is consistent internally, but is inconsistent with other networks. In this case, D-Kirk uses FINDNEXT messages to conduct a systematic search for a new consistent assignment, in order to achieve global consistency. FINDNEXT systematically moves through the subnetworks and returns when the first new consistent assignment is found. Therefore, a successful FINDNEXT message will cause a change to the value assigned to a single choice variable, which may in turn cause other choice variables to become active or inactive. Conversely, FINDFIRST attempts to make the first consistent assignment to the choice variable in every choice node in the network.

- *FAIL* indicates that no consistent set of assignments was found and hence the current set of assignments within the network is inconsistent.

- *ACK*, short for acknowledge, indicates that a consistent set of choice variable assignments has been found.

Whenever a node initiates search in its subnetworks, using FINDFIRST or FINDNEXT messages, the relevant processors search the subnetworks simultaneously. This is the origin of the parallelism in the algorithm.

The following three subsections describe the actions carried out by the start node of each network type on receipt of a FINDFIRST or FINDNEXT message. A more detailed description of the algorithm is given by Wehowsky [31].

### Activity-Sequence Network

During search, an `activity-sequence` start node propagates request messages forward and response messages backward.

### Parallel-Sequence Network

On receipt of a FINDFIRST message, the start node $v$ of a `parallel-sequence` network initiates a search of its subnetworks and of any successor network, in order to find a temporally consistent plan. The pseudo-code is in Fig. 6-3.

First, the start node sends FINDFIRST messages to the start node of each child subnetwork of the `parallel-sequence` structure (lines 2-4) and to the start node of the successor network, if present (lines 5-7). These searches are conducted in parallel and the start node waits for responses from each. If any of the child subnetworks or

```
1: $parent \leftarrow$ sender of $msg$
2: for each child $w$ do
3:    Send FINDFIRST to $w$
4: end for
5: if successor $B$ exists then
6:    Send FINDFIRST to $B$
7: end if
8: Wait for all responses from children
9: if successor $B$ exists then
10:    Wait for response from $B$
11: end if
12: if any of the responses is FAIL then
13:    Return FAIL to parent
14: else
15:    if check-consistency( $v$ ) then
16:       Return ACK to parent
17:    else
18:       if search-permutations($v$) then
19:          Return ACK to parent
20:       else
21:          Return FAIL to parent
22:       end if
23:    end if
24: end if
```

Figure 6-3: This figure shows the pseudo-code for the response of the start node $v$ of a `parallel-sequence` network to a FINDFIRST message.

the successor network fails to find a locally consistent assignment (line 12), then no consistent assignment to the choice variables exists and the start node returns FAIL (line 13).

Conversely, suppose that all child subnetworks and the successor network find variable assignments such that each is internally temporally consistent. The start node must then check for consistency of the entire `parallel-sequence` network (line 15). This is performed by a distributed Bellman-Ford consistency checking algorithm, which is explained in the next section. If the consistency check is successful, the start node returns an ACK message to its parent (line 16) and the search of the `parallel-sequence` network is complete.

If, however, the consistency check is not successful, the start node must continue searching through all permutations of assignments to the child subnetworks for a globally consistent solution (line 18). The pseudo-code for this search is in Fig. 6-4. The start node sends FINDNEXT messages to each subnetwork (lines 1-2). If a subnetwork fails to find a new consistent solution then the start node sends a FINDFIRST message to that subnetwork to reconfigure it to its original, consistent solution (lines 11-12) and we move on to the next subnetwork. If at any point, a subnetwork successfully finds a new consistent solution, the start node tests for global consistency and returns `true` if successful (lines 4-6). If the consistency check is unsuccessful, we try a different permutation of variable assignments (line 8) and continue searching. If all permutations are tested without success, then no globally consistent assignment exists (line 15).

```
1: for  $w$ = child-0 to child-n  do
2:    Send FINDNEXT to w
3:    Wait for response
4:    if  $response$ = ACK  then
5:       if  check-consistency( $v$ )  then
6:          Return true
7:       else
8:          $w \leftarrow$ child-0
9:       end if
10:   else
11:      Send FINDFIRST to $w$
12:      Wait for response
13:   end if
14: end for
15: Return false
```

Figure 6-4: This figure shows the `search-permutations(node` $v$`)` function.

When the start node $v$ of a `parallel-sequence` network receives a FINDNEXT message, it systematically searches all consistent assignments to its subnetworks, in order to find a new globally consistent assignment, just as described above. The pseudo-code is in Fig. 6-5.

If the search for a new globally consistent assignment (line 1) is successful, the start node sends ACK to its parent (line 2). If it fails, however, the start node attempts to find a new assignment to the successor network. If a successor network

```
1: if search-permutations() then
2:     Send ACK to parent
3: else if successor $B$ exists then
4:     Send FINDNEXT to $B$
5:     Wait for response
6:     Return response to parent
7: else
8:     Return FAIL to parent
9: end if
```

Figure 6-5: This figure shows the pseudo-code for the response of the start node $v$ of a `parallel-sequence` network to a FINDNEXT message.

is present, the start node sends a FINDNEXT message and returns the response to its parent (lines 3-6). If no successor network is present, then no globally consistent assignment exists and the node returns FAIL (line 8).

### Choose-Sequence Network

When the start node of a `choose-sequence` network receives a FINDFIRST message, it searches for a consistent plan by making an appropriate assignment to its choice variable and by initiating search in its successor network, if present. The pseudo-code is in Fig. 6-6.

The start node begins by sending a FINDFIRST message to any successor network (lines 2-4). It then systematically assigns each possible value to the network's choice variable and, in each case, sends a FINDFIRST message to the enabled subnetwork (lines 5-7). If a subnetwork returns FAIL, indicating that no consistent assignment exists, the current value of the choice variable is trimmed from its domain to avoid futile repeated searches (line 17), and the next value is assigned. As soon as a subnetwork returns ACK, indicating that a consistent assignment to the subnetwork was found, the start node checks the response from the successor network, if present. If no successor network is present (line 14), or if it is present and has returned ACK (line 14), the network is consistent and the start node returns ACK to its parent. If the successor network is present but has returned FAIL, the network is inconsistent and the start node returns FAIL to its parent (line 12). If all assignments to the

```
 1: parent ← sender of msg
 2: if successor B exists then
 3:    Send FINDFIRST to B
 4: end if
 5: for w = child-0 to child-n do
 6:    choiceVariable ← w
 7:    Send FINDFIRST to w
 8:    Wait for response from child w
 9:    if response = ACK then
10:       if successor B exists then
11:          Wait for response from successor B
12:          Return response to parent
13:       else
14:          Return ACK to parent
15:       end if
16:    else
17:       Remove w from child list
18:    end if
19: end for
20: Return FAIL to parent
```

Figure 6-6: This figure shows the pseudo-code for the response of the start node $v$ of a `choose-sequence` network to a FINDFIRST message.

network's choice variable are tried without receipt of an ACK message from a child subnetwork, the start node returns FAIL to its parent, indicating that no consistent assignment exists (line 20).

When the start node of a `choose-sequence` network receives a FINDNEXT message, it first attempts to find a new consistent assignment for the network while maintaining the current value of the choice variable. The pseudo-code is in Fig. 6-7.

The start node does this by sending FINDNEXT to the currently selected subnetwork (lines 1-2). If the response is ACK, a new consistent assignment has been found, so the start node returns ACK to its parent and the search is over (lines 4-6). If this fails, however, the start node searches unexplored assignments to the network's choice variable, in much the same way as it does on receipt of a FINDFIRST message (lines 7-16). If this strategy also fails, the start node attempts to find a new consistent assignment in any successor network. If a successor network does not exist, the node return FAIL to its parent (line 30). If a successor network does exist,

```
 1: $w \leftarrow$ current assignment
 2: Send FINDNEXT to $w$
 3: Wait for response
 4: if response = ACK then
 5:   Return ACK to parent
 6: end if
 7: while $w <$ child-n do
 8:   $w \leftarrow$ next child
 9:   Send FINDFIRST to $w$
10:   Wait for response
11:   if response = ACK then
12:     Return ACK to parent
13:   else
14:     Remove $w$ from child list
15:   end if
16: end while
17: if successor $B$ exists then
18:   Send FINDNEXT to $B$
19:   for $w$ = child0 to child-n do
20:     $choiceVariable \leftarrow w$
21:     Send FINDFIRST to $w$
22:     Wait for response from child $w$
23:     if response = ACK then
24:       Break
25:     end if
26:   end for
27:   Wait for response from $B$
28:   Return response to parent
29: else
30:   Return FAIL to parent
31: end if
```

Figure 6-7: This figure shows the pseudo-code for the response of the start node $v$ of a `choose-sequence` network to a FINDNEXT message.

the start node sends a FINDNEXT message (line 18). In this case, the start node must also reconfigure the `choose-sequence` network to its previous consistent state, by cycling through assignments to the choice variable until a consistent assignment is found (lines 19-26). Finally, the start node forwards the response from the successor network to its parent (lines 27-28).

We demonstrate the operation of the candidate generation phase of the distributed plan extraction algorithm on the example TPN shown in Fig. 6-1. The algorithm is initiated by node 1, the start node of the plan, which is the start node of a `parallel-sequence` network. Node 1 begins by sending a FINDFIRST message to the start nodes of its two subnetworks, nodes 2 and 68, which concurrently initiate search in their subnetworks. Node 1 has no successor network, so it waits for a reply from both of these nodes. Search of the `activity-sequence` subnetwork at node 68 is straightforward: it is temporally consistent by definition and it has no successor network, so it replies with an ACK message to node 1. Node 2 is the start node of a `choose-sequence` network, so on receipt of the FINDFIRST message from node 1 it first selects one of its subnetworks. Assume that the lower subnetwork, consisting of nodes 51 through 66, is selected, so node 2 sends a FINDFIRST message to node 51. Node 2 has no successor network, so it waits for a reply from node 51. Node 51 is the start node of a `parallel-sequence` network, so on receipt of the FINDFIRST message from node 2 it sends FINDFIRST messages to nodes 52 and 57, the start nodes of its two subnetworks. It also has a successor network, the `activity-sequence` network comprised of nodes 59 and 50, so it also sends a FINDFIRST message to node 59. These three nodes initiate search in their networks in parallel and node 51 waits for a reply from all three.

Node 52, the start node of the upper subnetwork of the `parallel-sequence` subnetwork, is the start node of an `activity-sequence` network. Since the `activity-sequence` network is temporally consistent, it forwards the FINDFIRST message to node 54, the start node of its successor network and waits for a reply. Node 54 is also the start node of an `activity-sequence` network, but it has no successor network, so it replies to node 52 with an ACK message. This ACK message is received by node 52 and forwarded to node 51, the start node of the `parallel-sequence` subnetwork. Node 57, the start node of the lower subnetwork of the `parallel-sequence` subnetwork, is the start node of an `activity-sequence` network with no successor, so it replies to node 51 with an ACK message. Node 51 has now received ACK messages from both of its successor subnetworks, so checks the temporal consistency of the combined `parallel-sequence` subnetwork. In this case the subnetwork is consistent, but the node must still wait for a reply from node 59, representing its successor network.

Node 59 is the start node of an `activity-sequence` network, but has a successor network, so forwards the FINDFIRST message from node 51 to node 61. The same is true of node 61, which forwards the message to node 63, which forwards it again to node 65. This node is the start node of an `activity-sequence` network without a successor. It therefore replies to node 63 with an ACK message, which is forwarded to node 61 and then on to node 59 and finally node 51. Node 51 can now knows that both the `parallel-sequence` subnetwork and its successor network are temporally consistent, so it replies to node 2 with an ACK message. Node 2 forwards the ACK message from node 51 to node 1.

Node 1 has now received ACK messages from both of its subnetworks, so checks the temporal consistency of its `parallel-sequence` network. In this case, the network is not consistent, because the lower bound of the upper subnetwork (20) exceeds the upper bound of the lower subnetwork (10). The details of how this consistency check is performed are given in the following section. Fig. 6-8 shows the TPN for the manipulator tool delivery scenario at this point during plan extraction. The deselected portion of the plan is shown in light gray. FINDFIRST messages are shown as blue arrows and ACK messages as green arrows.

Having determined that the currently selected plan is inconsistent, node 1 attempts to find a new, consistent assignment by sending a FINDNEXT message to each of its subnetworks in turn. It first sends a FINDNEXT message to node 2 and waits for a reply. Node 2, the start node of the `choose-sequence` subnetwork attempts to find a new consistent assignment to its network in the hope that this might be globally consistent. To achieve this, it first instructs its currently selected subnetwork to make a new assignment. It therefore sends a FINDNEXT message to node 51. Similarly, node 51 attempts to find a consistent assignment by forwarding the FINDNEXT message to each of its subnetworks in turn. The first subnetwork is an `activity-sequence` network with node 52 as its start node. This has no subnetworks to reconfigure, so forwards the FINDNEXT message to node 54, the start node of its successor network. Node 54 is the start node of an `activity-sequence network`, so has no subnetworks to reconfigure. It also has no successor network, so it replies to

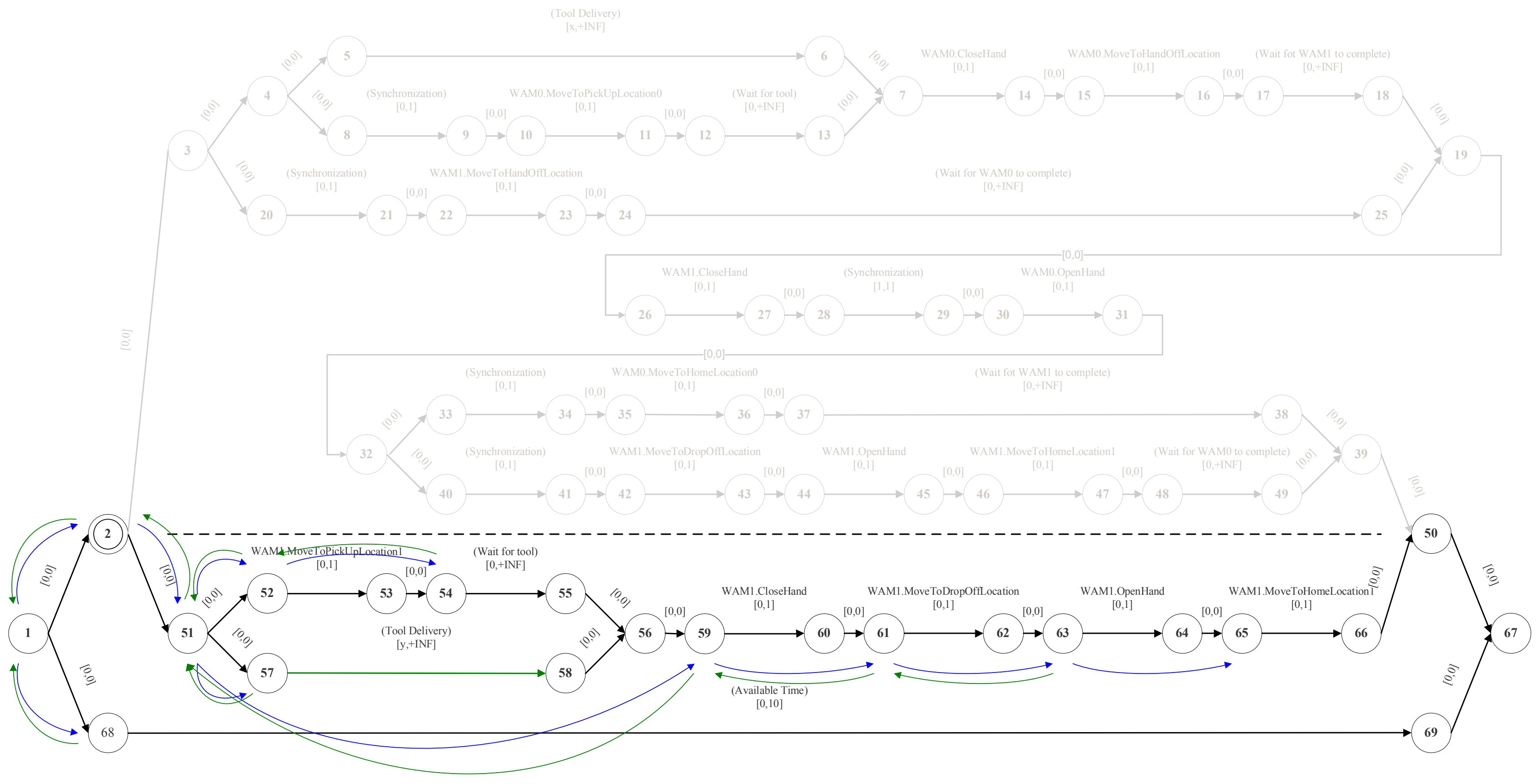


Figure 6-8: This figure shows the TPN for the manipulator tool delivery scenario during plan extraction when the temporal inconsistency is discovered at node 1. The deselected portion of the plan is shown in light gray. FINDFIRST messages are shown as blue arrows and ACK messages as green arrows.

node 52 with a FAIL message.

Node 51 now knows that the upper of its two subnetworks can not be reconfigured to find a new consistent solution. It therefore sends a FINDFIRST message to this subnetwork to configure it to its original consistent solution. This causes exactly the sequence of events in nodes 52 and 54 as was described earlier. Node 51 then attempts to reconfigure its lower subnetwork. However, this too is an `activity-sequence` subnetwork so no reconfiguration is possible and the reply is a FAIL message from node 57. Again, node 51 sends a FINDFIRST message to this subnetwork to configure it to its original consistent solution and the sequence of events in node 57 is exactly as was described earlier. Node 51's final attempt to find a new consistent solution to is to send a FINDNEXT message to the start node of its successor network, node 59. However, this network is an `activity-sequence` network and so are the following 3 successor networks, so no reconfiguration is possible. The FINDNEXT message is therefore forwarded through nodes 61 and 63 to node 65, but the eventual reply to node 51 is FAIL. Node 51 sends a FINDFIRST message to its successor subnetwork to configure it to its original consistent solution and the sequence of events in nodes 59, 61, 63 and 65 is exactly as was described earlier.

Node 51 has now exhausted all attempts to find a new consistent solution in its subnetworks and its successor networks. It therefore replies to node 2 with a FAIL message. Fig. 6-9 shows the TPN for the manipulator tool delivery scenario at this point during plan extraction. The deselected portion of the plan is shown in light gray. FINDNEXT messages are shown as black arrows and FAIL messages as red arrows.

Node 2 has failed to reconfigure the currently selected subnetwork to provide global temporal consistency of the plan. It therefore selects a new subnetwork, in the hope that it will provide global temporal consistency. In this case, it selects the only other subnetwork, the upper subnetwork, and initiates search by sending a FINDFIRST message to node 3. It then waits for a reply.

Node 3 is the start node of a `parallel-sequence` network, so forwards the FINDFIRST message to nodes 4 and 21, the start nodes of its two subnetworks. It also

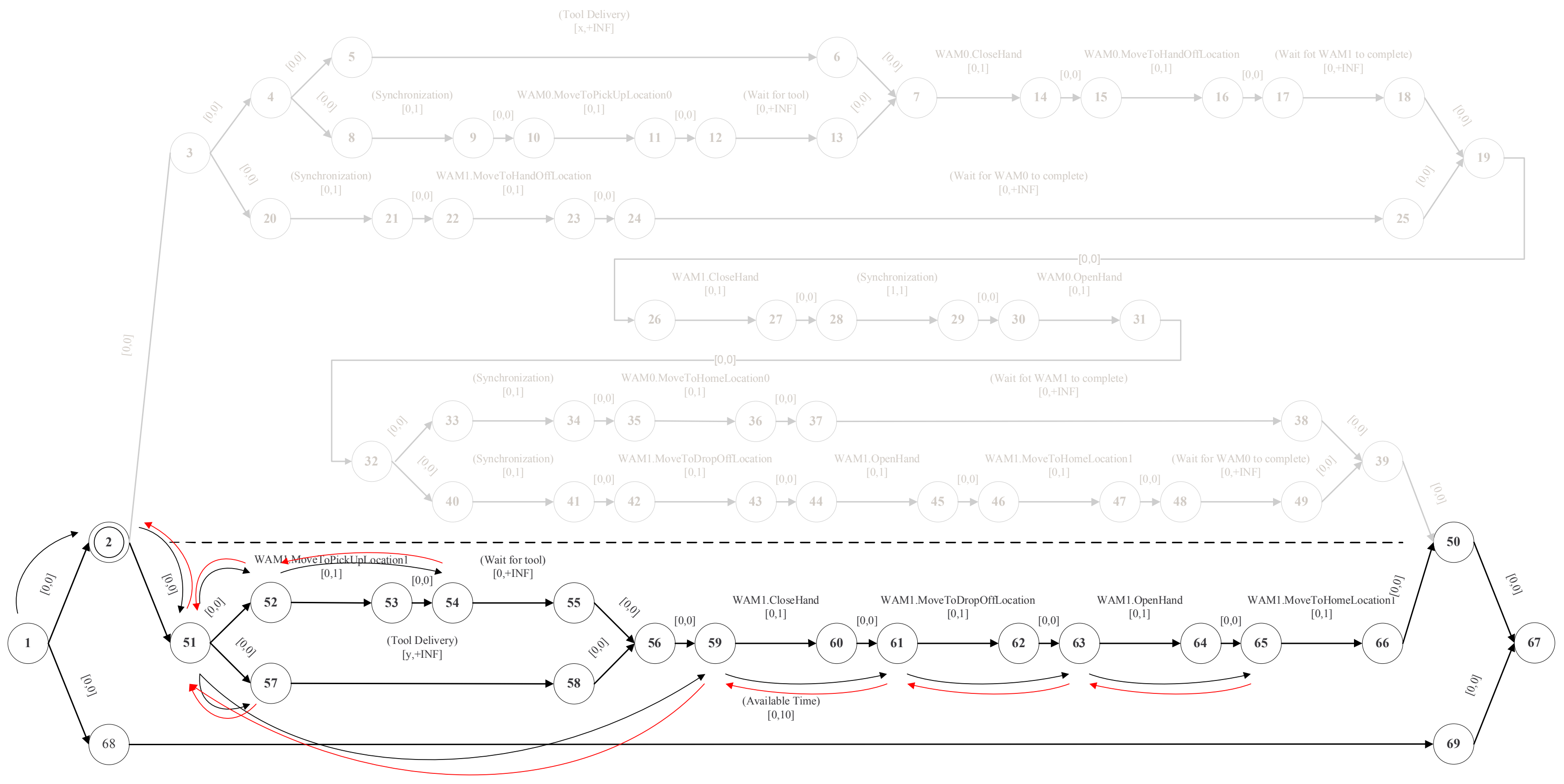


Figure 6-9: This figure shows the TPN for the manipulator tool delivery scenario during plan extraction after node 2 has attempted to find a new consistent assignment in its currently selected subnetwork (nodes 51 through 66). The deselected portion of the plan is shown in light gray. FINDNEXT messages are shown as black arrows and FAIL messages as red arrows.

forwards the message to node 26, the start node of its successor network, and waits for replies. The FINDFIRST message received by node 26 is forwarded along the chain of successor networks, visiting nodes 28, 30 and 32. Each subnetwork searches for a consistent assignment, but there are no `choose-sequence` networks in this branch of the plan, so there are no choices to be made. The last network searched therefore replies with a ACK message, which is forwarded back along the chain of nodes. The branch does however, include 3 `parallel-sequence` networks, at nodes 3, 4 and 32. When the ACK messages reaches these nodes, therefore, a consistency check is made, but in this example these are all successful, so the ACK message eventually reaches node 3.

Once node 3 has received the ACK messages from both of its subnetworks and its successor network, it replies to node 2 with an ACK message. Node 2 has now found a new locally consistent assignment, as it was instructed to by the first FINDNEXT message from node 1. It therefore replies to node 1 with an ACK message.

Node 1 now knows that the subnetwork rooted at node 2 is locally consistent. The subnetwork rooted at node 68 is also consistent, as shown by the first round of FINDFIRST messages. Node 1 must now check the temporal consistency of the selected plan as a whole, using the distributed Bellman-Ford algorithm, as described in the next section. In this case, the selected plan is indeed temporally consistent because the lower bound of the upper subnetwork (2) is less than the upper bound of the lower subnetwork (10). Plan extraction is therefore complete.

Fig. 6-10 shows the TPN for the manipulator tool delivery scenario at the end of plan extraction. The deselected portion of the plan is shown in light gray. FINDFIRST messages are shown as blue arrows, FINDNEXT messages as black arrows, ACK messages as green arrows and FAIL messages as red arrows.

### 6.2.3 Temporal Consistency Checking

The second phase of the distributed plan extraction algorithm is the testing of candidate plans for temporal consistency. This is interleaved with candidate generation. If the candidate plan is inconsistent, we must return to the candidate generation phase

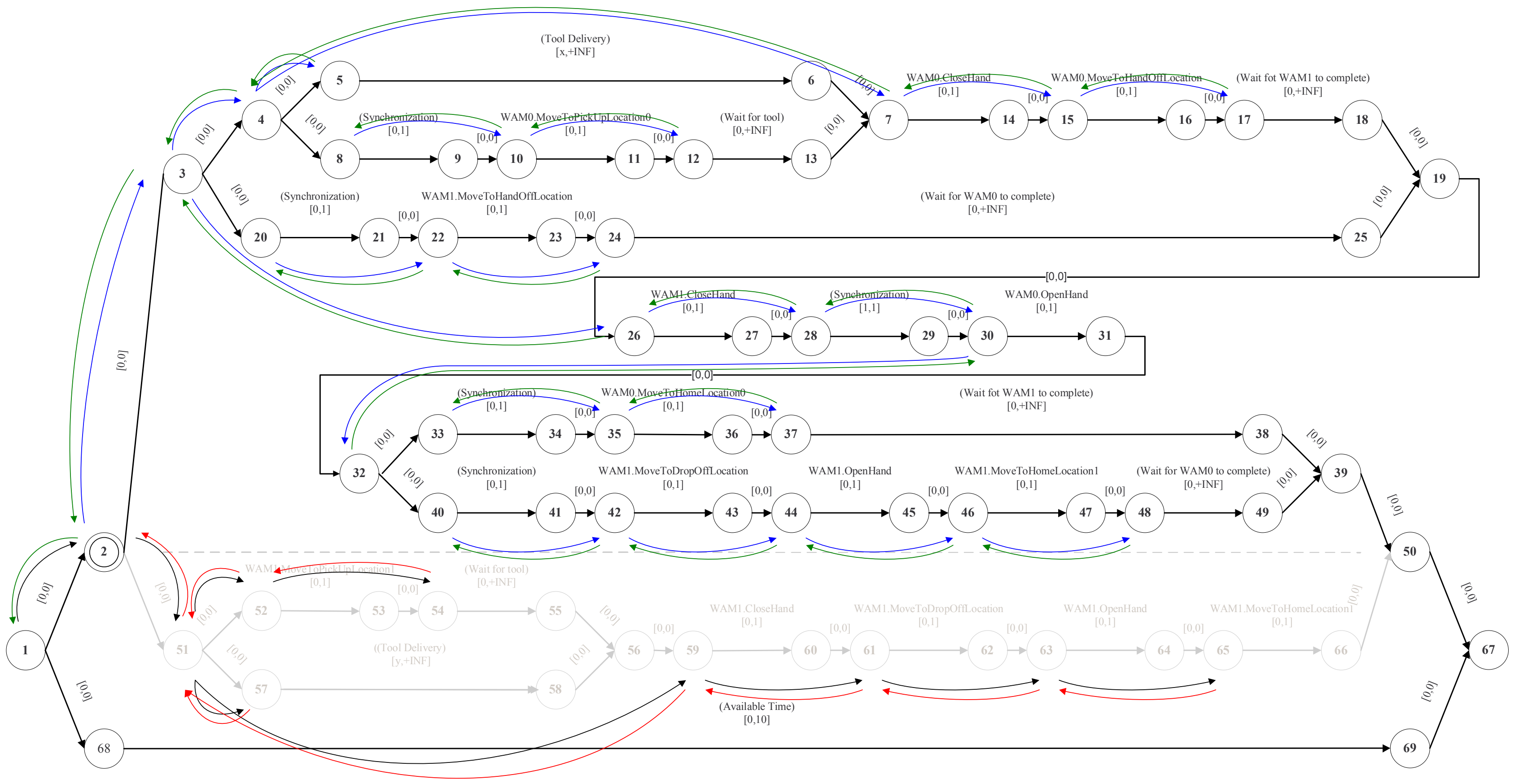


Figure 6-10: This figure shows the TPN for the manipulator tool delivery scenario at the end of plan extraction. The deselected portion of the plan is shown in light gray. FINDFIRST messages are shown as blue arrows, FINDNEXT messages as black arrows, ACK messages as green arrows and FAIL messages as red arrows.

in search of a new, temporally consistent plan.

As was discussed in Section 2.5, a temporal inconsistency appears as a negative cycle in the distance graph corresponding to the plan. We check for the presence of a negative cycle by performing a Single Source Shortest Path (SSSP) calculation on the distance graph corresponding to the portion of the TPN that represents the current candidate, from the start node of the plan. The plan extraction algorithm in D-Kirk uses the distributed Bellman-Ford algorithm, which was described in Section 4.2, to calculate the SSSP distances.

Use of the distributed Bellman-Ford algorithm has two key advantages. First, it requires only local knowledge of the TPN at every processor. Second, when run synchronously, it runs in time linear in the number of processors. Consistency checking is interleaved with candidate generation, such that D-Kirk simultaneously runs multiple instances of the distributed Bellman-Ford algorithm on isolated subsets of the TPN.

Recall that the distributed Bellman-Ford algorithm uses an iterative scheme to update each node's SSSP distance estimate. In the absence of negative cycles in the distance graph, the distance estimates will converge to the true SSSP values. In the presence of negative cycles, the concept of a SSSP distance is invalid, so the algorithm will not converge. If run synchronously, the algorithm converges in $N-1$ rounds, where $N$ is the number of nodes in the plan, provided that negative cycles are not present. We can therefore test for negative cycles by running $N$ rounds of the synchronous algorithm. If the distance estimates remain unchanged between the last two rounds, then they were converged after $N-1$ rounds and no negative cycles exist.

For example, consider the distance graph shown in Fig. 6-11. This graph is temporally inconsistent due to the negative cycle ABDCA. Fig. 6-11(a) shows the initial state for a distributed Bellman-Ford SSSP calculation using node A as the start node. The initial SSSP distance estimate is zero for node A and $+INF$ for all other nodes. After the first round, the distance estimate of nodes B and C have been reduced to 8 and 10, respectively, due to the edges AB and AC, as shown in Fig. 6-11(b). In the second round, node D reduces its distance estimate to 9, due to the edge BD, as shown in Fig. 6-11(c). By the end of the third round, node C has reduced its distance estimate to 5, due to the edge DC, as shown in Fig. 6-11(d). In the absence of negative cycles, the distance estimates would now have converged to the true SSSP values. However, the fourth round, shown in Fig. 6-11(e), causes node A to reduce its distance estimate to -1, due to the edge CA. This means that the distance estimates had not converged in the previous round and a negative cycle exists. In fact, the estimates will continue to change due to the negative cycle, as shown by Fig. 6-11(f), which shows the fifth round.

### 6.2.4 Complexity

The planning phase of D-Kirk offers an improvement in computational complexity compared to a centralized architecture. The distributed Bellman-Ford algorithm has time complexity $O(Ne)$, compared to $O(NE)$ for the centralized version of the algorithm, where $N$ is the number of nodes in the plan, $e$ is the number of edges per node and $E$ is the total number of edges. note, however, that the overall worst-case computational complexity of the planning algorithm remains exponential, due to the candidate generation phase.

## 6.3 Conclusion

To summarize, this chapter introduced the plan extraction component of D-Kirk. The plan extraction algorithm employs a series of distributed algorithms that exploit the hierarchical structure of a TPN to search the TPN for candidate plans and finally, check for candidate temporal consistency. This distributed approach reduces the computational load on each processor and allows concurrent processing.

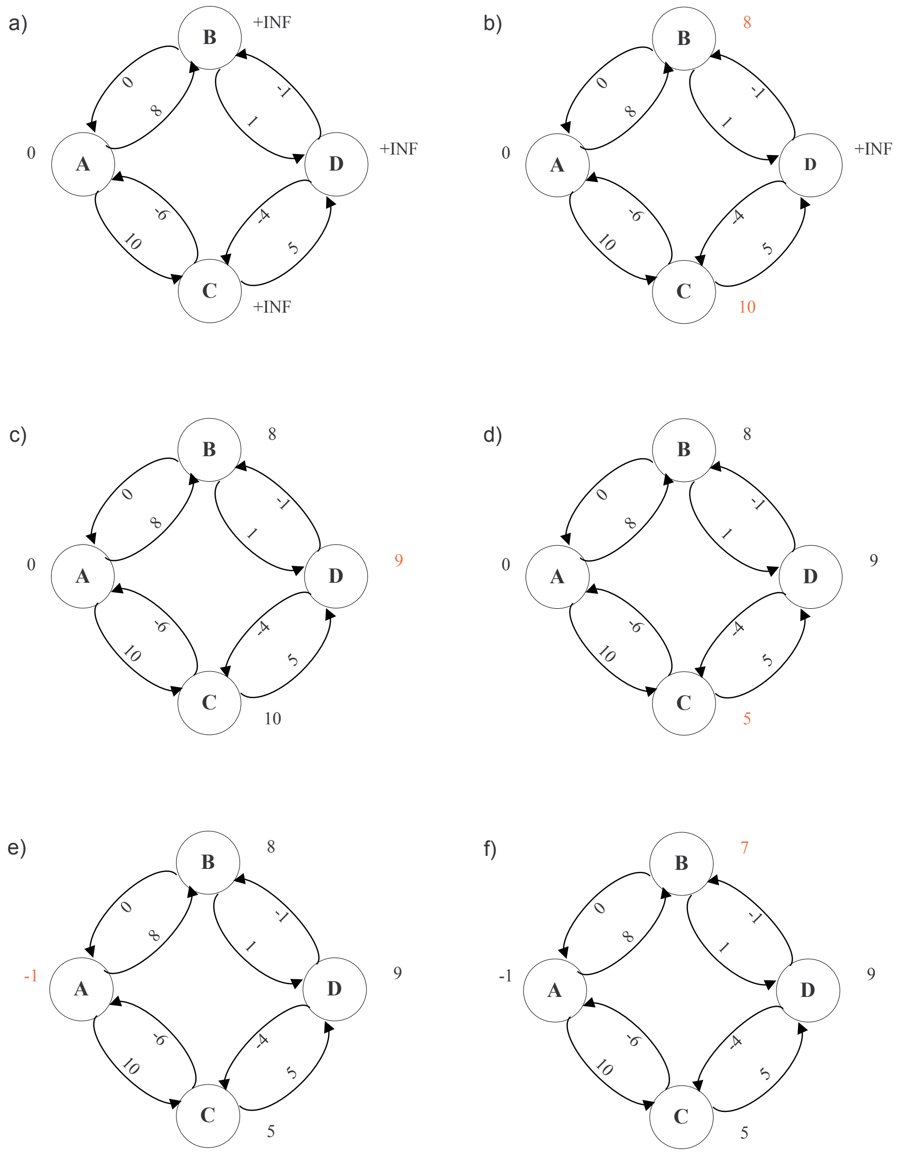


Figure 6-11: This figure shows how the synchronous distributed Bellman-Ford algorithm detects a temporal inconsistency as a negative cycle in the distance graph after $N$ rounds. (a) Initial state for synchronous distributed Bellman-Ford single source shortest path calculation from node A. (b) Single source shortest path distance estimates after round 1. (c) Single source shortest path distance estimates after round 2. (d) Single source shortest path distance estimates after round 3. (e) Single source shortest path distance estimates after round 4 show failure to converge as a result of the negative cycle. (f) Single source shortest path distance estimates continue to vary after round 5.

# Chapter 7

# Results

## 7.1 Introduction

This chapter presents experimental results to confirm the performance of D-Kirk and to demonstrate its use in a realistic scenario. It also gives a conclusion for the work presented in this thesis and discusses possible directions for future work.

## 7.2 Performance Tests

This section presents experimental results to verify the following key advantages of the dispatching and reformulation phases of D-Kirk over a centralized executive.[1] The advantage offered by the dispatching phase of D-Kirk over a centralized executive is the avoidance of the communication bottleneck. This is measured in terms of the maximum number of messages sent by a single node during dispatching. The advantage offered by the reformulation phase of D-Kirk over a centralized executive is the reduction in computational complexity experienced by each node. This is measured in terms of the average CPU time required by a node during reformulation. The tests were conducted using a C++ implementation of D-Kirk with a simulated plant.

[1]The data from all of the tests described here can be found on-line through the MERS website at http://mers.csail.mit.edu.

The performance tests make use of a set of generic input plans of varying complexity. We present experimental data for three types of plan, as described below. The sequence and and parallel types are included to represent two extremes of plan and the RMPL type represents a more typical plan. For each type of plan we vary $N$, the number of nodes in the plan.

- **Sequence** plans consist of all nodes arranged in sequence, with adjacent nodes connected by a simple temporal constraint. Here the branching factor is at a minimum. It is equal to 2, since each node is connected to only the nodes immediately before and after it in the sequence.

- **Parallel** plans have a start and end node, with $N-2$ nodes arranged in parallel between them. Here the branching factor is at a maximum and is approximately equal to the number of nodes in the plan.

- **RMPL** plans represent a typical plan generated from an RMPL program for practical use. They contain `parallel` and `sequence` constructs, so the branching factor has an intermediate value.

### 7.2.1 Dispatching

The key advantage over a centralized executive offered by the dispatching phase of D-Kirk systems is its robustness to communication latency. This is achieved by avoiding the communication bottleneck at dispatch time present in centralized systems by reducing the peak communication complexity. Here we provide experimental data to validate this claim. The metric we use is the maximum number of messages sent by a single node during dispatching.

We conduct tests using the three types of plan introduced above. In each case, we show how the maximum number of messages sent by a single node during the dispatching phase of D-Kirk varies with the number of nodes in the plan. We also compare the results to the centralized case.

In the centralized case, only the master node sends messages. In the worst case, for a plan with $N$ nodes, the master node must send $N-1$ messages; one to instruct each of the other nodes in the plan to execute. This number is independent of the structure of the plan. In the distributed case, we count the number of EXECUTED messages sent by each node and take the maximum.

Fig. 7-1 shows the variation in the maximum number of messages sent as a function of the number of nodes in the plan for each of the three types of plan. The first trend to note is that for all plans, the maximum number of messages sent by a single node in the distributed dispatcher is less than that of the centralized dispatcher.

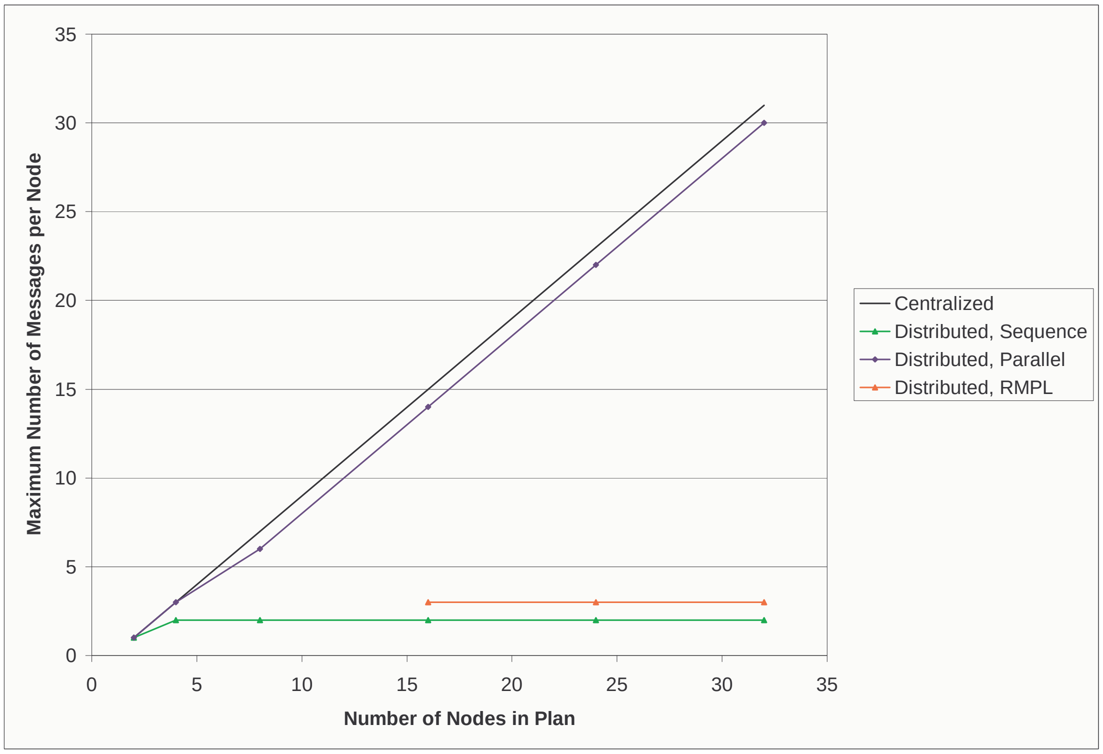


Figure 7-1: This figure shows the results of the tests used to verify the performance of D-Kirk with regard to communication complexity. The plot shows the peak number of messages sent by a node during dispatching as a function of the number of nodes in the plan.

The parallel plan type represents the worst case scenario for the distributed dispatcher because the branching factor is at its maximum. In the input TPN the start node has an edge to every other node in the plan, other than the end node. This is also true of the MDG after reformulation, so the start node must send $N-2$ EXECUTED messages. For this reason, the distributed dispatcher performs only slightly better than the centralized dispatcher in terms of the maximum number of messages sent by a single node.

The sequential plans lie at the other end of the spectrum. Even after reformulation, each node has edges only to its 2 immediate neighbors. This means that in the distributed dispatching algorithm each node must send EXECUTED messages to at most 2 nodes. For this reason, the distributed dispatcher performs significantly better than the centralized dispatcher in terms of the maximum number of messages sent by a single node.

The RMPL plans use a combination of `parallel` and `sequence` constructs. Therefore, each node has MDG edges to number of nodes intermediate between that for the sequence and parallel plans discussed above. In the examples presented here, the maximum number of parallel branches introduced at a given node is 2, so the maximum number of EXECUTED messages sent by a single node is 3. This means that the distributed dispatcher performs significantly better than the centralized dispatcher in terms of the maximum number of messages sent by a single node.

To summarize, these results show that the distributed dispatcher reduces the maximum number of messages sent by any node in the plan, relative to the centralized dispatcher, in all cases. In realistic scenarios, where an agent owns on average $\frac{N}{A}$ nodes, the number of messages would increase by this factor but still remains less than that of the centralized dispatcher. This helps eliminate the communication bottleneck present in a centralized architecture and provides robustness to communication latency. The improvement is most significant when the plan is simply a sequence of nodes, but the improvement for more realistic plans is also significant.

### 7.2.2 Reformulation

The key advantage provided by the reformulation phase of D-Kirk over a centralized executive is the reduction in computational complexity of the processing that must be carried out by each node. Here we provide experimental data to validate this claim. The metric we use is the average CPU time required by a node to complete the reformulation process.

We conduct tests using the three types of plan introduced above. In each case, we show how the average CPU time required by a node during the reformulation phase of D-Kirk varies with the number of nodes in the plan. We then compare these results to the centralized case.

In the centralized executive, only the master node performs the computation required by reformulation. In the distributed case, all nodes perform a part of the computation. The CPU time required by each node is similar, so we take an average value. In all cases, the value plotted is an average over 5 trials, to account for random variation.

Note that for the purpose of these tests we do not wish to include message delivery time in the measurement of each node's processing time. This is because the time delay involved in sending a message is heavily dependent upon the type of communication used. For example, communication between two nodes owned by the same agent will likely be much faster than communication between nodes owned by different agents, which may be implemented over TCP/IP or a serial link. We therefore assume a uniform communication cost.

Recall that the reformulation algorithm begins by rearranging rigid components. A leader is assigned to each rigid component and all edges to non-member nodes are relocated from all other member nodes to the leader node. Once this is complete, only the leader node of each RC takes part in the MDG formation phase of reformulation. This means that for MDG formation, the number of nodes $N$ is effectively reduced to the number of rigid components. MDG formation accounts for the majority of the total computational complexity of reformulation (a fraction of approximately $\frac{N}{N+1}$). This means that a reduction in the effective value of $N$ due to the presence of rigid components can reduce the average CPU time significantly. This is confirmed by the experimental results discussed below.

Fig. 7-2 shows the variation in processor time as a function of the number of nodes in the plan for the D-Kirk reformulation algorithm for each of three types of plan. The first feature to note is that for all three types of plan, the relationship between the processor time and the number of nodes in the plan is polynomial. The complexity analysis in Section 4.3.4 predicts that the worst-case computational complexity is $O\left(N^2 e\right)$, where $N$ is the number of nodes in the plan and $e$ is the number of immediate neighbors to each node. Analysis of the experimental data shows that the processor time varies as $N^{1.8}$ for sequence type plans, $N^{1.9}$ for parallel type plans and $N^{0.5}$ for RMPL type plans. The relationships for the sequence and parallel type plans are as expected, as the predicted quadratic complexity is a worst-case estimate. The apparently low complexity for the RMPL type plans is due to the presence of rigid components in the plan. RMPL plans connect activities using a simple temporal constraint with zero lower and upper bounds. This forms the nodes it connects into a rigid component. The number of rigid components, therefore, is approximately equal to half the number of nodes in the plan.

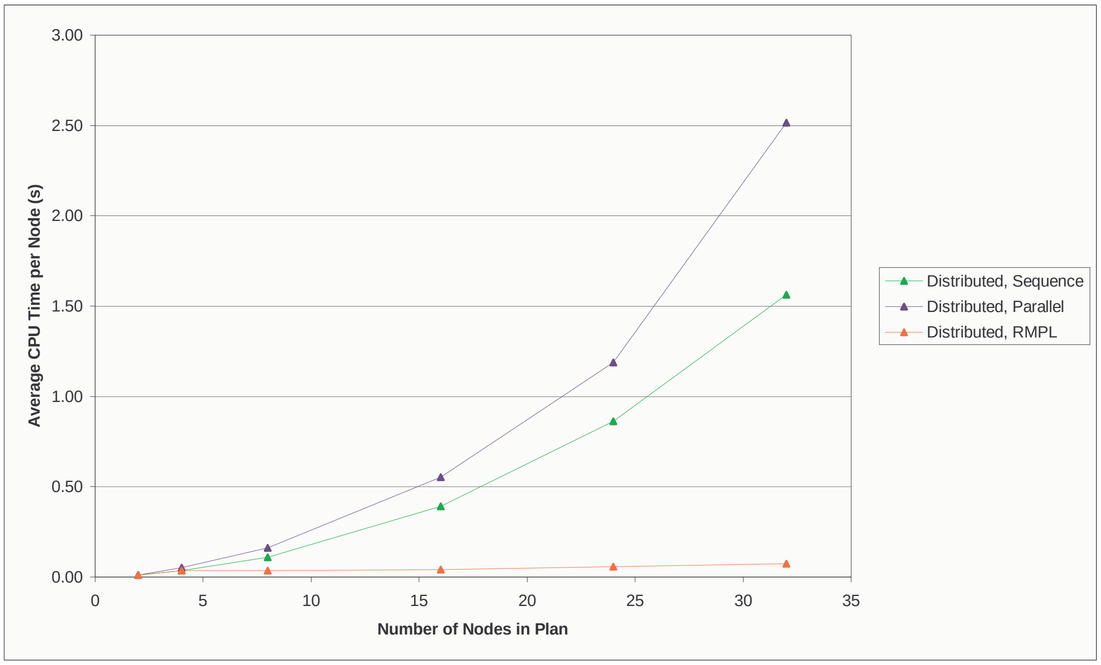


Figure 7-2: This figure shows the results of the tests used to verify the performance of D-Kirk with regard to computational complexity. The plot shows the average run time required by a node during reformulation.

Fig. 7-3 compares the computation time required by each node in D-Kirk to that of the lead node in the centralized case. The results are plotted on a log-log scale to allow better visualization of the large range of values present. It is immediately clear that for the relatively small plans presented, the centralized reformulation algorithm runs significantly faster than the distributed algorithm for a given value of $N$ by as much as two orders of magnitude. This is to be expected, as the distributed algorithm is significantly more complicated than the centralized version, and processor time must be spent sending and receiving messages.

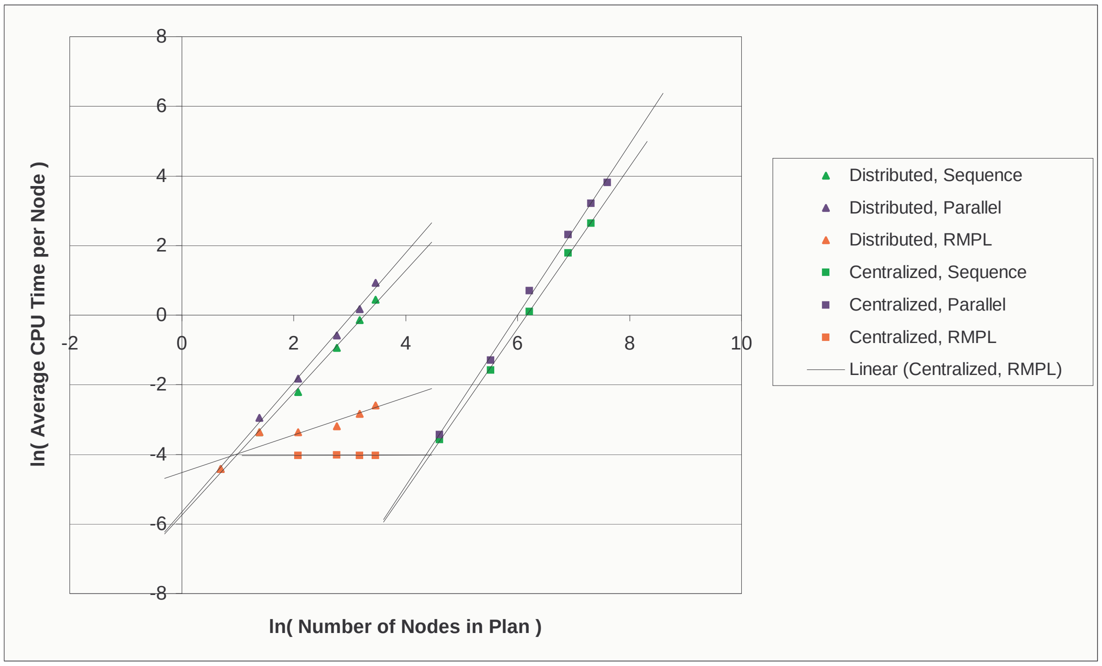


Figure 7-3: This figure shows the results of the tests used to verify the performance of D-Kirk with regard to computational complexity. The plot compares the average run time required by a node during reformulation for the centralized and distributed algorithms.

However, the complexity of the distributed algorithm is lower than that of the centralized algorithm. As mentioned above, the analysis in Section 4.3.4 predicts a worst-case computational complexity of $O\left(N^2 e\right)$ per node and this is confirmed by the experimental data. For the centralized reformulator, we expect a worst-case computational complexity of $O\left(N^2 E\right)$, where $E$ is the total number of edges in the plan. For the sequence and parallel plan types presented here, the total number of edges $E$ is at least $2N$, so we can expect a cubic relationship between the computational complexity and the number of nodes. The experimental data shows that the processor time varies as $N^{2.3}$ for sequence type plans and $N^{2.5}$ for parallel type plans. Again, this is as expected, since the predicted cubic complexity is a worst-case estimate.

The reduced complexity of the distributed reformulation algorithm means that for large plans, it will run more quickly than the centralized algorithm. For small plans, this advantage is outweighed by the overhead incurred due to the more complicated algorithm, and this explains the trend we see in Fig. 7-3.

We therefore conclude that the most significant advantage of D-Kirk is in the reduced number of messages required at dispatch time. To best exploit this advantage, however, we should use the distributed approach only for coordinating nodes owned by separate agents, where inter-processor communication is costly. In this case, the reduction in the number of messages will be of practical benefit in providing robustness to communication latency. For nodes owned by the same agent, however, the cost of intra-processor communication is very low and there is no advantage to be gained from using the distributed approach. Instead, we should use a centralized executive to coordinate these nodes and use the distributed approach for coordination between the groups of nodes owned by each agent. This will reduce the CPU time required by each node because the simpler centralized algorithm will be used where possible. This idea is discussed further in Section 7.5, on future work.

## 7.3 Real World Scenario

To demonstrate the effectiveness of D-Kirk in a realistic scenario we used the C++ implementation mentioned earlier to execute plans on a pair of four-degree-of-freedom robotic manipulators. The manipulators used were Barret Technology's Whole Arm Manipulators (WAMs). Each manipulator is controlled by an individual PC, so each manipulator was considered an individual agent. To provide inter-agent communication, a TCPIP/IP message interface was written for the D-Kirk implementation.

The plan executed is the manipulator tool delivery scenario introduced in Chapter 1. The graph representation of this plan is shown in Fig. 1-2. D-Kirk successfully executed the plan, providing tight coordination of the two manipulators in real time. The distributed executive significantly lowered the peak communication complexity at dispatch time, reducing the maximum number of dispatch messages sent by any given node from 68, in the centralized case, to 3 in the distributed case.

## 7.4 Conclusion

To summarize, this thesis introduced D-Kirk, a distributed executive that performs robust execution of contingent, temporally flexible plans. In particular, D-Kirk operates on Temporal Plan Networks (TPNs) and distributes both data and processing across all available processors. D-Kirk employs a series of distributed algorithms that first, form a processor hierarchy and assign TPN subnetworks to each processor; second, search the TPN for a temporally consistent plan; third, reformulate the selected plan to a form amenable to execution and; finally, dispatch the plan.

The novel contributions made by this thesis include a distributed version of the fast reformulation algorithm. This algorithm uses parallel processing wherever possible to maximize efficiency and uses a state machine based approach to provide robustness to variation in message delivery times. We also present a novel distributed dispatching algorithm that provides a significant reduction in the number of messages required at dispatch time. This algorithm too is robust to variation in message delivery times.

The distributed approach spreads communication evenly across the processors, thus eliminating the bottleneck in communication at dispatch time that is present in a centralized architecture. Furthermore, the distributed algorithms reduce the computational load on each processor at all four stages of execution and allow concurrent processing for increased performance.

Experimental results confirmed the significant reduction in the number of messages required by the distributed executive at dispatch time, compared to the centralized executive. The magnitude of the reduction is $O\left(\frac{A}{e'}\right)$, where $e'$ is the number of edges connected to each node in the MDG and $A$ is the number of agents involved in executing the plan. Therefore this factor depends heavily upon the structure of the plan being executed. The value of $e'$ is determined by the branching factor of the plan and is typically a small constant less then 5. Experimental results also showed the reduction in computational complexity during reformulation provided by the distributed executive. However, the overhead of the added complexity of the distributed framework is significant and suggests that best results may be obtained from a hybrid of the centralized and distributed approaches.

## 7.5 Future Work

The most obvious area in which improvements could be made to D-Kirk is in reducing the CPU time required during reformulation. The distributed reformulation algorithm reduces the computational complexity per node from $O(N^3)$ in the centralized case to $O(N^2)$ in the distributed case. However, the distributed algorithm introduces significant processing overhead so is significantly slower than the centralized version for many plans of practical size.

To overcome this poor performance in computational speed, without sacrificing the advantages to be gained in the reduced number of messages at dispatch time, a hybrid of the distributed and centralized algorithms may be the best solution. In this case, nodes owned by a single agent, where intra-agent communication is of very low cost, would be coordinated using a centralized algorithm to reduce computational requirements. However, coordination of the nodes owned by different agents would be done using a distributed approach, thereby minimizing message complexity where it matters.

The task of forming this hybrid executive would not be trivial because some nodes would need to communicate with neighbor nodes owned by both their local agent and by remote agents. The task could be simplified by partially decoupling the plan into groups of nodes owned by a single agent. Recent work in this area has been done by Stedl [25].

Another area for obvious improvement is in extending D-Kirk to correctly handle plans with uncontrollable activities. The fast reformulation algorithm [27] used in D-Kirk produces an MDG for which correct execution is not guaranteed if the input plan contains uncontrollable activities. To ensure correct execution of uncontrollable activities we must ensure that the MDG is dynamically controllable. To implement this change in D-Kirk we would have to produce a distributed version of either the dynamic controllability algorithm by Morris et. al. [18] or the fast dynamic controllability algorithm by Stedl [25]. Note that the distributed dispatching algorithm is capable of correctly dispatching both controllable and uncontrollable activities.

Another area for possible improvement is in the plan distribution phase. Currently, the algorithm makes no attempt to produce a distribution which minimizes the peak computational load on a single processor. Currently, a single node processor could be assigned a large number of nodes while all other processors are responsible for only a single node. If the algorithm were to consider the plan structure, a distribution scheme could be found which is more efficient with regard to computational complexity.

Another potential improvement is to provide robustness to failure of a processor. This is a significant challenge, and would involve autonomous detection of the failure followed by redistribution of the plan and re-planning, before the execution was resumed.

# Bibliography

[1] Stephen A. Block and Brian C. Williams. Robust execution of contingent, temporally flexible plans. In *AAAI 2006 Workshop on Cognitive Robotics*, Boston, MA, July 2006.

[2] Daniel Coore, Radhika Nagpal, and Ron Weiss. Paradigms for structure in an amorphous computer. Technical Report AIM-1614, Massachusetts Institute of Technology, 6, 1997.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1995.

[4] Thomas L. Dean and Drew V. McDermott. Temporal database management. *Artificial Intelligence*, 32:1–55, 1987.

[5] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[6] Marie E. DesJardins, Edmund H. Durfee, Charles L. Ortiz, and Michael J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 1999.

[7] Robert Effinger. Optimal temporal planning at reactive timescales via dynamic backtracking branch and bound. Master's thesis, MIT, Cambridge, MA, August 2006.

[8] Tara Estlin, Gregg Rabideau, Darren Mutz, and Steve Chien. Using continuous planning techniques to coordinate multiple rovers. In *Proceedings of the IJCAI Workshop on Scheduling and Planning*, 1999.

[9] Tara A. Estlin, Alexander Gray, Tobias Mann, Gregg Rabideau, Rebecca Castano, Steve Chien, and Eric Mjolsness. An integrated system for multi-rover scientific exploration. In *AAAI/IAAI*, pages 613–620, 1999.

[10] Andreas Hofmann and Brian C. Williams. Safe execution of temporally flexible plans for bipedal walking devices. In *Proceedings of the International Conference of Automation, Planning and Scheduling (ICAPS-2005)*, 2005.

[11] Lina Khatib, Paul Morris, Robert A. Morris, and Francesca Rossi. Temporal constraint reasoning with preferences. In *IJCAI*, pages 322–327, 2001.

[12] Phil Kim, Brian C. Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle, WA, 2001.

[13] Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takhashi, Atsuhi Shinjou, and Susumu Shimada. Robocup-rescue: Search and rescue for large scale disasters as a domain for multi-agent research. In *Proceedings of the IEEE Conference on Systems, Men, and Cybernetics*, 1999.

[14] Thomas Léauté and Brian C. Williams. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-2005)*, 2005.

[15] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[16] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-1990)*, 1990.

[17] Paul Morris and Nicola Muscettola. Execution of temporal plans with uncertainty. In *AAAI/IAAI*, pages 491–496, 2000.

[18] Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *IJCAI*, pages 494–502, 2001.

[19] Nicola Muscettola. *HSTS: Integrating Planning and Scheduling.* Morgan Kaufmann, 1994.

[20] Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Principles of Knowledge Representation and Reasoning*, pages 444–452, 1998.

[21] Paolo Pirjanian, Terrance L. Huntsberger, Ashitey Trebi-Ollennu, Hrand Aghazarian, Hari Das, Sanjay S. Joshi, , and Paul S. Schenker. Campout: a control architecture for multirobot planetary outposts. In *Proceedings of SPIE*, pages 221–230, 2000.

[22] Paolo Pirjanian and Maja Matarić. Multi-robot target acquisition using multiple objective behavior coordination. In *Proceedings of the IEEE Conference on Robotics and Automation*, 2000.

[23] Patrick Riley and Manuela Veloso. Planning for distributed execution through use of probabilistic opponent models. In *Proceedings of IJCAI-2001 Workshop PRO-2: Planning under Uncertainty and Incomplete Information*, 2001.

[24] Peter J. Schwartz and Martha E. Pollack. Planning with disjunctive temporal constraints. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-2004)*, 2004.

[25] John L. Stedl. Managing temporal uncertainty under limited communication: A formal model of tight and loose team coordination. Master's thesis, MIT, Cambridge, MA, September 2004.

[26] Ioannis Tsamardinos. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Proceedings of the Second Hellenic Conference on AI (SETN 2002)*, 2002.

[27] Ioannis Tsamardinos, Nicola Muscettola, and Paul Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-1998)*, pages 254–261, 1998.

[28] Thierry Vidal and Hélène Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *JETAI*, 11(1):23–45, 1999.

[29] Thierry Vidal and Malik Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proceedings of the 12th European Conference on Artifical Intelligence (ECAI 96)*, 1996.

[30] Aisha Walcott. Unifying model-based programming and random path planning through optimal search. Master's thesis, MIT, Cambridge, MA, May 2004.

[31] Andreas F. Wehowsky. Safe distributed coordination of heterogeneous robots through dynamic simple temporal networks. Master's thesis, MIT, Cambridge, MA, September 2003.

[32] Brian C. Williams, Michel Ingham, Seung Chung, and Paul H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, 2003.