Distributed Method Selection and Dispatching of Contingent, Temporally Flexible Plans [*] [†]

Stephen A. Block and **Brian C. Williams**
Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, MA, 02139, USA
sblock@mit.edu, williams@mit.edu

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 uncertainty. Previous work has produced contingent plan execution systems that provide robustness during their execution phase, by dispatching temporally flexible plans, and during their plan extraction phase, by choosing between functionally redundant methods. 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.

This paper introduces a robust, *distributed* executive for temporally flexible plans. To execute a plan, the plan is first distributed over multiple 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 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 the distributed executive, the communication load is spread over all agents, thus eliminating the communication bottleneck. In addition, the distributed algorithms reduce the computational load on each agent and provide opportunities for parallel processing, thus increasing efficiency.

Introduction

The ability to coordinate groups of autonomous agents is key to many real-world tasks, such as a search and rescue mission, or the construction of a Lunar habitat. Achieving this ability dependably requires techniques that are robust to four types of disturbances: temporal uncertainty, execution uncertainty, plan failure, and communication latency. We address the first three types of disturbances by leveraging prior work on robust plan execution.

Temporal Uncertainty. Temporally flexible plans (Dean & McDermott 1987) (Muscettola 1994) allow us to model activities of uncertain duration. Use of these plans allows us to provide robustness to temporal uncertainty.

Execution Uncertainty. To adapt to uncertainty in execution times, while executing the plan in a timely manner, the dispatcher dynamically schedules events. In particular, we use the methods of (Tsamardinos, Muscettola, & Morris 1998), which use a least commitment strategy. Here, scheduling is postponed until execution time, allowing temporal resources to be assigned as-needed. To minimize the computation that must be performed on-line at dispatch time, the plan is first reformulated off-line. This scheme provides robustness to execution uncertainty, including the uncertainty in the execution times of other agents.

Plan Failure. To address plan failure, (Kim, Williams, & Abramson 2001) introduced a system called *Kirk*, 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) (Kim, Williams, & Abramson 2001), 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 above. Use of contingent plans adds robustness to plan failure.

As a centralized approach, however, Kirk can be brittle to failures caused by *communication latency*. Once a plan has been extracted and reformulated it must be executed and in the case of a multi-agent plan, this involves multiple agents. Therefore, we must communicate the primitive activities to all of the agents that will take part in the execution. This creates a communication bottleneck at the master agent.

We address this limitation through a distributed version of Kirk, called D-Kirk, which performs *distributed* execution of contingent temporally flexible plans. In the distributed framework, all agents send and receive messages. This

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

[†]This paper appeared in the Proceedings of the ICAPS Workshop on Planning Under Uncertainty, Cumbria, UK, June 2006 under the title 'Robust Execution of Contingent, Temporally Flexible Plans'.

evens out communication requirements and eliminates the brittleness to communication bottlenecks. In addition, we distribute computation between all agents to reduce computational complexity and take advantage of parallel processing, thus improving performance relative to the centralized architecture

D-Kirk consists of the following four phases.

1. Distribute the TPN across the processor 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.

This paper begins with a summary of dispatchable execution. We then present the steps required to execute a temporally flexible plan in a distributed manner. These are the distribution, reformulation and dispatching algorithms and correspond to steps 1, 3 and 4 above. Our key innovation is a set of distributed algorithm that handle limited inter-agent communication and are robust to all communication delays.

Finally, we extend the work to include contingent plans. We present a summary of TPNs and of previous work by (Wehowsky 2003) for dynamically selecting a feasible plan from a TPN, which corresponds to step 2 above.

Dispatchable Execution

The task of the executive is to robustly execute a temporally flexible plan. Temporally flexible plans make use of simple temporal constraints to describe uncertain durations. A simple temporal constraint $[l, u]$ places a bound $t^+ - t^- \in [l, u]$ on duration between the start time t^- and end time t^+ of the activity or sub-plan to which it is applied.

In all that follows, we discuss temporally flexible plans in terms of a graph representation, where nodes represent time events and directed edges represent simple temporal constraints. An example temporally flexible plan is shown in graph form in Fig. 1.

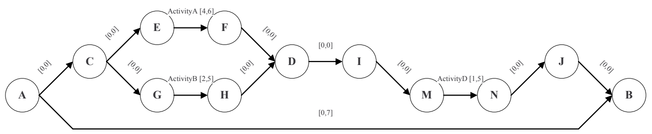

Figure 1: Example temporally flexible plan

Given a temporally flexible plan to be executed, traditional execution schemes fix the execution schedule at planning time, thus removing the temporal flexibility prior to execution. This approach leads to two problems. First, the fixed schedule lacks the flexibility to respond to temporal uncertainty 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.

We overcome this limitation with dispatchable execution (Muscettola, Morris, & Tsamardinos 1998), where scheduling is postponed until execution time. At execution time we have the most information available regarding execution history, so the dispatcher can schedule activities just-in-time. This provides robustness to uncertain durations that lie within the temporally flexible bounds of the plan.

Just-in-time scheduling introduces an added run-time computation that increases latency in the system's ability to respond to disturbances. To minimize the amount of computation that must be conducted in real-time, we use a two-stage execution strategy. Prior to execution, we use reformulation to compile the plan to a form that allows easy dispatching. In particular, we reformulate the plan 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. Dispatching of a plan in MDG form requires only local propagation of timing information.

To form the MDG, reformulation identifies the non-dominated edges in the plan. These are the edges along which execution information must be propagated at run time. The most simple reformulation algorithm begins by forming the All Pairs Shortest Path (APSP) graph. It then traverses the APSP graph and tests every edge to determine whether it is non-dominated. However, constructing the entire APSP graph is inefficient, requiring $O(N^2)$ time complexity and $O(N^2)$ space complexity, where N is the number of nodes in the plan. Furthermore, searching the APSP graph for non-dominated edges has $O(N^3)$ time complexity, giving an overall complexity for the simple reformulation algorithm of $O(N^3)$ in time and $O(N^2)$ in space.

We overcome this problem with fast reformulation (Tsamardinos, Muscettola, & Morris 1998). Fast reformulation extracts non-dominated edges by a series of traversals of the graph, without forming the complete APSP graph. In the worst case, we conduct a traversal from every node in the plan graph, to test whether the implicit edge from that node to the node being investigated is non-dominated. In order for the dominance tests to be conducted efficiently, we must conduct each traversal in Reverse Post Order (RPO) for the predecessor graph rooted at the relevant start node.

We can further reduce complexity by exploiting Rigid Components (RCs). A rigid component is a set of nodes whose execution times are fixed relative to each other. During reformulation we can represent each RC by a single node, thus reducing the effective value of N. The other nodes in each RC are then added back into the plan once reformulation is complete. Note that the treatment of RCs in this way is also required for the dominance tests to function correctly.

The node representing a given RC is known as the RC leader, and is the node with minimum Single Source Shortest Path (SSSP) distance from the start node. In order for it to represent the RC in the dominance tests, the edges to all other RC members are re-routed to the leader node. The member nodes themselves are connected with a doubly linked chain of edges, in increasing SSSP distance, as shown in Fig. 2. All other edges are deleted. We connect RC member nodes in this way because the doubly linked chain edges are guaranteed to be part of the MDG, so do not need to be processed in the dominance tests.

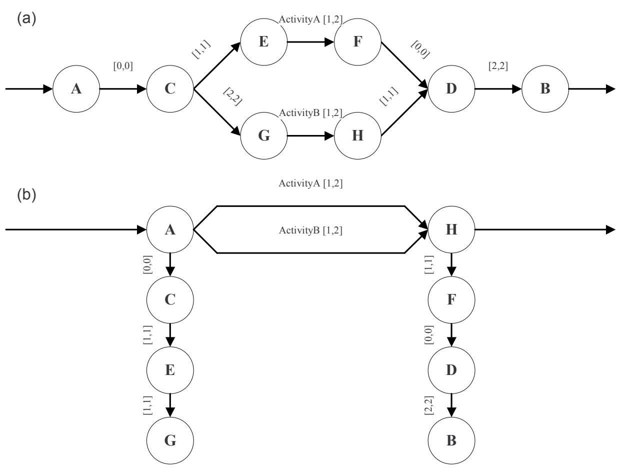

Figure 2: Rigid Components; (a) Before processing; (b) After processing

Plan Distribution

The first step of D-Kirk is to distribute the input plan over the available processors. A plan is distributed by assigning to each processor the responsibility for zero or more nodes of the plan graph, such that each node is assigned to exactly one processor. During the subsequent steps of D-Kirk, each processor maintains all data relevant to its assigned nodes.

D-Kirk begins by establishing a hierarchical, ad-hoc communication network, using the leader election algorithm in (Nagpal & Coore 1998). Given a set of processors with fixed but unknown communication availabilities, this algorithm forms a hierarchy of processors where communication is guaranteed between a processor and its *leader*, *neighbor leaders* and *followers*, as shown in Fig. 3.

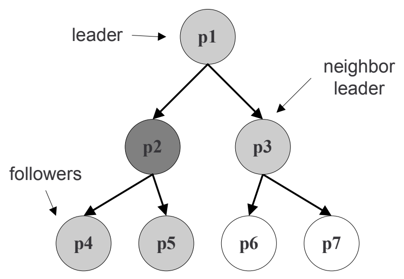

Figure 3: Example processor hierarchy, showing communication availability for processor p2

We then use a distribution algorithm (Wehowsky 2003) to assign nodes to processors in the hierarchy. The temporal consistency checking performed in the planning phase is centered around the temporal constraints between nodes of the plan. Therefore, this algorithm distributes the plan such that each pair of processors responsible for nodes linked by a temporal constraint are able to communicate.

The algorithm allows the plan to be distributed down to a level at which a processor handles only a single node; this permits D-Kirk to operate on heterogeneous systems that include computationally impoverished processors.

Reformulation

In D-Kirk we distribute the fast reformulation algorithm between all nodes, using a message passing scheme. We use a state machine approach to structure the reformulation algorithm, so each node can track its progress through the computation. This means that we do not require any existing synchronization between nodes: the state machine provides synchronization based solely on the messages received. Also, this 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 that are received early.

As a result, 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. This allows reliable operation when guarantees can not be placed upon the speed of communication channels. Also, the algorithm provides significant error checking with regard to the type of message that can be received at a given time.

The algorithm operates on the distance graph corresponding to the input plan. It requires only the communication channels guaranteed during distribution of the plan. Messages that must be sent between nodes that are not connected by known channels are redirected to follow the channels, with the re-routing handled internally by the algorithm.

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

We also present the computational complexity of each phase of the algorithm, for comparison with the centralized case. Note that the algorithm is event driven, where an event is the receipt of a message. The computation to be performed on receipt of each message is simple, so the computational complexity is of the same order as the message complexity.

We illustrate the operation of the algorithm on the example temporally flexible plan shown in Fig. 1. Line numbers refer to the pseudo-code for the distributed reformulation algorithm shown in Fig. 4.

Rigid Component Processing (lines 1-16)

As mentioned above, for correct operation of the dominance tests, we must represent each RC by a single node. RC processing is therefore the first part of the distributed reformulation algorithm and proceeds as follows.

Form predecessor graph. The predecessor graph is the graph of shortest paths from the start node to all other nodes. To form the predecessor graph we use the distributed Bellman-Ford algorithm (Lynch 1997), algorithm to find the SSSP distance to each node (line 2). The distributed Bellman-Ford algorithm requires only local knowledge of the graph at each node, hence allowing the SSSP calculation to be performed locally. To ensure that the algorithm converges in time linear in the number of nodes, the Bellman-Ford algorithm is run synchronously. This algorithm is fully parallel. Once the SSSP distances have been computed, we use them to form the predecessor graph (line 3).

Note that for the purpose of extracting RCs, we perform the SSSP calculation from a *phantom node*, a virtual node which has edges of zero length to every other node in the plan. The distance graph corresponding to the example plan in Fig. 1, with phantom node added, is shown in Fig. 5(a). The SSSP distances and predecessor graph for the example plan are shown in Fig. 5(b).

The complexity of this stage is dominated by the SSSP calculation, which has computational complexity $O(Ne)$, where e is the number of edges at each node.

Extract reverse post order. The post order is the order in which nodes are removed from the search queue during search. We extract the Reverse Post Order (RPO) from a Depth First Search (DFS) on the predecessor graph (line 4). Since the RPO is inherently ordered, this part of the algorithm can not be conducted in parallel and is entirely serial. The computational complexity for each node is $O(e)$.

Note that in the presence of the phantom node, this search is more complicated than a simple DFS. We simulate a DFS search from the phantom node by starting a DFS from every node that has a zero SSSP distance. The RPO for the example plan is shown in Fig. 5(c).

A given node must complete its part in forming the predecessor graph before it can take part in RPO extraction, but

```
 1: Process Rigid Components
 2: Compute SSSP distances from phantom node using synchronous distributed Bellman-Ford
 3: Form predecessor graph using SSSP distances
 4: Perform DFS on predecessor graph and record RPO
 5: for Each node in the graph, taken in RPO order do
 6:     Perform DFS on transposed predecessor graph to extract members of this RC and their edges
 7:     Determine member node with minimum SSSP distance and set as RC leader
 8:     for Each member node do
 9:         Form edges for RC doubly linked chain and record as members of MDG
10:         Delete all other edges
11:     end for
12:     for Each edge do
13:         Relocate to RC leader
14:         Inform remote node of the relocation
15:     end for
16: end for
17: Perform Dominance Tests
18: for Each RC leader do
19:     Compute SSSP distances from this node for the graph of RC leaders using synchronous distributed Bellman-Ford
20:     Form predecessor graph using SSSP distances
21:     Perform DFS on predecessor graph and record RPO
22:     Begin traversal in RPO
23:     for Each node traversed do
24:         Use minimum and non-positive data to apply dominance tests
25:         if Implicit edge is non-dominated then
26:             Record implicit edge as member of MDG
27:         end if
28:         Update minimum and non-positive data
29:         Propagate minimum and non-positive data to successors
30:     end for
31:     Record non-dominated edges as members of MDG
32: end for
33: Initialize Execution Windows
34: Compute SSSP distances from start node using outgoing non-negative MDG edges using synchronous distributed Bellman-Ford
35: Record SSSP distances as upper bound of execution windows
36: Compute SSSP distances from start node using incoming non-positive MDG edges using synchronous distributed Bellman-Ford
37: Record SSSP distances as lower bound of execution windows
```

Figure 4: Distributed Reformulation Algorithm

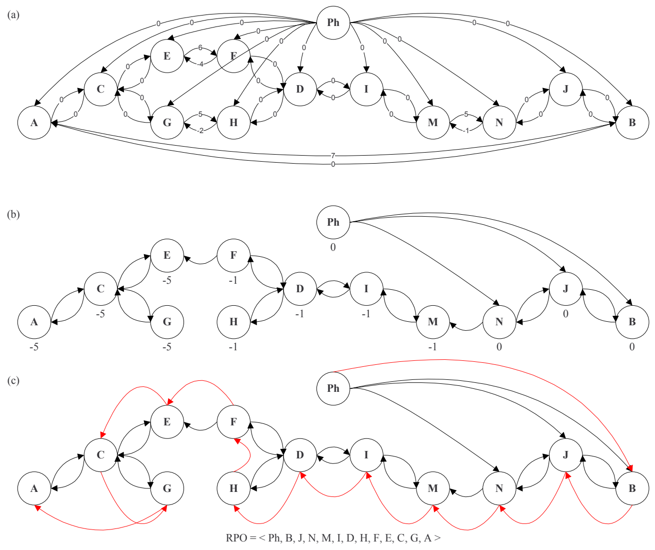

Figure 5: Reformulation example: (a) Distance graph with phantom node; (b) SSSP distances and predecessor graph; (c) RPO

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.

Process rigid components. We determine the members of each rigid component with a series of DFSs on the transposed predecessor graph. The transposed predecessor graph for the example plan is shown in Fig. 6(a). We start a DFS from every node in the graph, with the order determined by the predecessor graph RPO calculated above (line 5). The nodes visited in each DFS belong to a single RC.

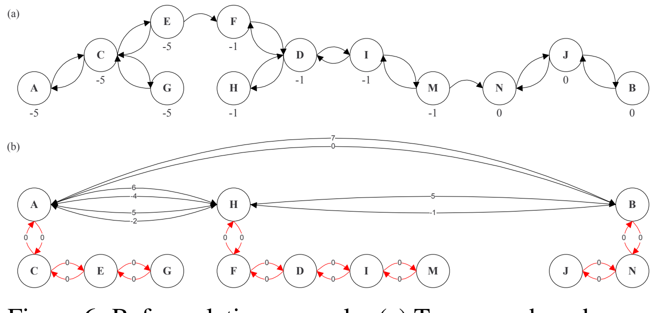

Figure 6: Reformulation example: (a) Transposed predecessor graph; (b) Plan with RC processing complete

During each DFS, we gather the list of member nodes and the list of edges belonging to them (line 6). After each DFS is complete, the relevant start node assigns as RC leader the member node with the minimum SSSP distance (line 7). It also sends messages to all members of the RC to instruct them to form the doubly linked chain of edges that connects the member nodes (line 9) and to delete all other edges (line 10). If the receiving node is the RC leader, it also adds the rearranged edges from the other members of the RC (line 13). Finally, the start node sends messages to inform the nodes at the far end of the RC members' edges that the edges have been moved to the RC leader (line 14). The example plan with RC processing complete is shown in Fig. 6(b).

Within each DFS, processing is performed in parallel: multiple branches of the tree are searched simultaneously. However, the DFSs must be strictly ordered relative to each other for the RC extraction to be successful, so this is done sequentially.

The complexity of this phase is dominated by the extraction of the members of the RC and by rearranging edges to the leader, giving a computational complexity of $O(e)$.

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.

Dominance Test Traversals (lines 17-32)

The dominance tests determine whether an edge is a member of the MDG and are applied in a series of graph traversals. The traversals are conducted on the subset of the plan graph comprised of RC leaders (line 18). In a traversal started at node A, the dominance test applied as we traverse node B determines whether or not the implicit edge AB is a member of the MDG.

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.

Each traversal must follow the RPO for the predecessor graph rooted at the start node of the traversal. The procedure for each traversal is as follows.

Form predecessor graph. We form the predecessor graph as described above, but we do not use a phantom node and we ignore any edges to RC non-leaders (lines 19-20). The computational complexity per traversal remains $O(Ne)$.

Extract reverse post order. We extract the RPO as described above, again not using a phantom node and ignoring any edges to RC non-leaders (line 21). The computational complexity per traversal remains $O(e)$.

Traverse and apply dominance tests. We conduct the traversal in the order given by the RPO (line 22). The traversal begins at the first node in the RPO and by definition, this is the last node to complete the RPO extraction phase, so the traversal does not begin until RPO extraction is complete.

At each node in the traversal, we send messages to all predecessor nodes with the values of the following two pieces of data (lines 28-29).

- **minimum** : the minimum SSSP distance encountered on this traversal.
- **non-positive** : whether a non-positive SSSP distance has been encountered on this traversal.

When the traversal reaches a node, it uses these pieces of data to determine whether or not the implicit edge is dominated (line 25). If the edge is not dominated, it is recorded locally (line 26). The edge is also recorded in a list which is passed back to the start node of the traversal for recording there (line 31).

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 $O(e)$.

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. The SSSP distances, predecessor graph, RPO and non-dominated edges for the example plan are shown in Figs. 7(a), (b) and (c) for the traversals from nodes B, H and A respectively.

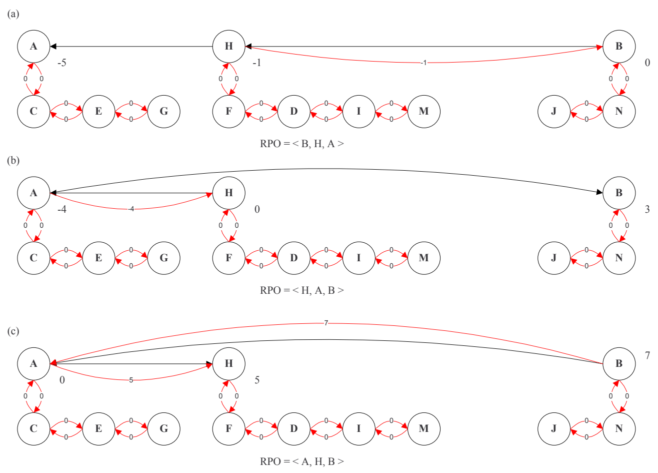

Figure 7: Reformulation example: SSSP distances, predecessor graph and non-dominated edges for MDG traversals; (a) From node B; (b) From node H; (c) From node A

Initialize Execution Windows (lines 33-37)

Once the dominance test traversals are complete, we must initialize the execution window for each node before dispatching. A node can not determine when it has completed the dominance test phase because we can not easily calculate how many traversals it will be involved in. This is because, in general, a node is only reachable from a subset of the nodes in the plan for the purposes of SSSP distance and RPO calculations. Therefore, when the traversal from a given node is complete, this node sends a message to inform the overall start node that this is the case. Once all such messages have been received, we know that the dominance test phase is complete and the execution windows can be calculated.

Calculate upper bounds. We determine the upper execution bound on each node with a SSSP calculation from the start node. We use the distributed Bellman-Ford algorithm described above, but we do not use the phantom node and we consider only outgoing non-negative MDG edges (lines 34-35). In the worst case, the number of MDG edges per node is $O(e)$, so the computational complexity per traversal is $O(Ne)$.

Calculate lower bounds. We determine the lower execution bound on each node with a SSSP calculation from the start node. We use the distributed Bellman-Ford algorithm described above, but again we do not use the phantom node and we consider only incoming non-positive MDG edges (lines 36-37). Again, the number of MDG edges per node is $O(e)$, so the computational complexity per traversal is $O(Ne)$. Since we can not conduct multiple Bellman-Ford calculations simultaneously, we complete the upper bound calculations before we begin those for the lower bounds. The complete MDG and initial execution windows for the example plan are shown in Fig. 8.

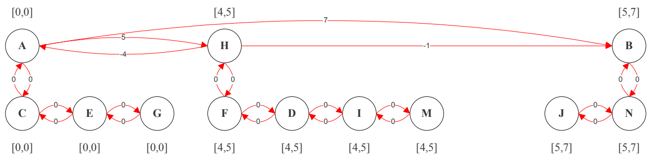

Figure 8: Reformulation example: Complete MDG and initial execution windows

Complexity

Using the complexities of each stage described above, we obtain a worst case overall computational complexity of $O(N^2 e + Ne)$ for the distributed reformulation algorithm. Both of these terms are due to the Bellman-Ford SSSP calculations: approximately N calculations of complexity $O(Ne)$ used to obtain the predecessor graphs for RC processing and dominance test traversals, and 2 calculations of complexity $O(Ne)$ for the execution windows.

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 $O(N^2 E)$. The improvement due to distributed processing, therefore, is approximately a factor of $\frac{E}{e} \approx N$.

Dispatching

Once the plan has been reformulated to an MDG, the dispatcher executes the activities in real time. We base our distributed algorithm on the dispatching algorithm of (Muscettola, Morris, & Tsamardinos 1998). We distribute the dispatching algorithm over all nodes, using a message passing scheme.

As with the reformulation algorithm, we use a state machine approach to provide robustness to delays in message delivery. This allows the algorithm to function correctly without requiring any synchronization between processors.

However, synchronization is required for the precisely timed execution of tasks. For this we assume a synchronous execution model. In particular, we require that each processor has a synchronized clock. The task of achieving this synchronization is not trivial, but is beyond the scope of this work. Note however, that approximate synchronization, to within the delivery time of a single message, is sufficient for many practical applications.

During dispatching, the processors operate independently, monitoring incoming messages and their clock to determine when their nodes' activities can be executed. The only messages used are those sent to inform neighbor nodes that a node has executed. The information carried in these messages is used to update the lower and upper bounds on a node's execution window and to propagate enablement conditions. Line numbers in the following description of the algorithm refer to the pseudo-code shown in Fig. 9.

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

Figure 9: Distributed Dispatching Algorithm

First, a node must wait to be *enabled* (lines 1-4). A node is enabled when all nodes that must execute before it have been executed. These nodes are identified as those which are found at the end of outgoing non-positive MDG edges (line 2). While waiting, the node responds to incoming messages (line 3).

Once enabled, a node must wait for the current time to enter its execution window and for all uncontrollable activities that end at this node to complete (lines 6-14). While waiting, the node checks that the current time does not exceed the upper bound of the execution window (line 7), else the plan execution fails (line 8). Also, the node continues to respond to incoming messages (line 10).

Once the current time is in the execution window and all uncontrollable activities have completed, we can execute the node (lines 15-18). First, we stop any controllable activities that end at this node (line 16) and then start any activities which begin at this node (line 17). Finally, we send messages to inform neighbor nodes that the node has executed (line 19).

The number of messages sent by each node is determined by the number of edges connected to it in the MDG. In the worst case, this number is $O(e)$.

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 $O(E)$. 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.

Furthermore, since the computational complexity is directly proportional to the number of messages received, D-Kirk improves this too.

Temporal Plan Networks

In order to encode contingencies, a Temporal Plan Network (TPN) augments the temporally flexible plan representation with a `choose` operator. 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 primitive element of a TPN is an `activity`$[l, u]$, which is 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 duration between the start time t^- and end time t^+ of the activity or contingent sub-plan to which it is applied. A TPN is built from a set of primitive activities and is defined recursively using the `choose`, `parallel` and `sequence` operators, taken from the Reactive Model-based Programming Language (RMPL) (Williams *et al.* 2003). A TPN encodes all executions of a non-deterministic concurrent, timed program, comprised of these operators.

- `choose`$(TPN_1, \ldots, TPN_N)$ introduces multiple subnetworks of which only one is to be chosen. A choice variable is used at the start node to encode the currently selected subnetwork. A choice variable is *active* if it falls within the currently selected portion of the TPN.
- `parallel`$(TPN_1, \ldots, TPN_N)\,[l, u]$ introduces multiple subnetworks to be executed concurrently. A simple temporal constraint is applied to the entire network.
- `sequence`$(TPN_1, \ldots, TPN_N)\,[l, u]$ introduces multiple subnetworks which are to be executed sequentially. A simple temporal constraint is applied to the entire network.

Graph representations of the `activity`, `choose`, `parallel` and `sequence` network types are shown in Fig. 10. Nodes represent time events and directed edges represent simple temporal constraints. A choice node is shown as an inscribed circle.

A *temporally consistent* plan is obtained from the TPN if and only if a *feasible choice assignment* is found. See (Wehowsky 2003) for a more precise definition.

Definition 1 *A temporally flexible plan is* ***temporally consistent*** *if there exists an assignment of times to each event such that all temporal constraints are satisfied.*

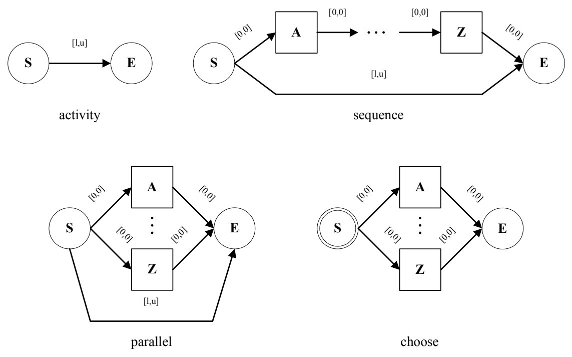

Figure 10: TPN Constructs

Definition 2 *A* ***feasible 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, and 3) the temporally flexible plan (program execution) corresponding to this assignment is* ***temporally consistent***.

Plan Extraction

The plan extraction phase of D-Kirk consists of two interleaved processes: generation of candidate plans and testing them for temporal consistency.

The candidate plans correspond to different assignments to the choice variables at each choice node and are obtained by solving a conditional CSP (Mittal & Falkenhainer 1990). The D-Kirk planning algorithm uses parallel, recursive, depth first search to make these assignments.

Consistency checking is implemented using the distributed Bellman-Ford SSSP algorithm and is run on the distance graph corresponding to the portion of the TPN that represents the current candidate. Temporal inconsistency is detected as a negative weight cycle (Dechter, Meiri, & Pearl 1991).

The planning algorithm exploits the hierarchical structure of the TPN to allow parallel processing. 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. D-Kirk uses a distributed message-passing architecture that employs the following messages for candidate plan generation.

- `findfirst` instructs a network to make the initial search for a consistent set of choice variable assignments. If a node at level n in the hierarchy receives a `findfirst` message, it propagates it to all of its subnetworks at level $n + 1$ simultaneously. If each subnetwork finds a consistent assignment, we then check for consistency at level n.
- `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. To achieve this, a node at level n in the hierarchy receiving a `findnext` message

forwards the message to each subnetwork at level $n+1$ in turn. When a new consistent assignment to a subnetwork is found, we check for consistency at level n. 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.

- `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 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. Overall, the worst-case computational complexity of the planning algorithm remains exponential, due to the candidate generation phase.

An example TPN containing a single choice node I is shown in Fig. 11. The only feasible choice assignment is the pathway through nodes M and N, and this gives rise to the temporally consistent plan shown in Fig. 1.

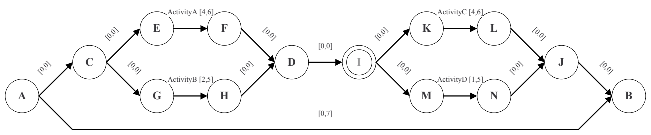

Figure 11: Reformulation example: Input TPN

Conclusion

To summarize, this paper 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 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. This 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.

References

Dean, T., and McDermott, D. 1987. Temporal database management. *Artificial Intelligence* 32:1–55.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Kim, P.; Williams, B.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001).*

Lynch, N. 1997. *Distributed Algorithms*. Morgan Kaufmann.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-1990).*

Muscettola, N.; Morris, P.; and Tsamardinos, I. 1998. Reformulating temporal plans for efficient execution. In *Principles of Knowledge Representation and Reasoning*, 444–452.

Muscettola, N. 1994. *HSTS: Integrating Planning and Scheduling*. Morgan Kaufmann.

Nagpal, R., and Coore, D. 1998. An algorithm for group formation in an amorphous computer. In *Proceedings of the Tenth International Conference on Parallel and Distributed Systems (PDCS-1988).*

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast transformation of temporal plans for efficient execution. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-1998)*, 254–261.

Wehowsky, A. F. 2003. Safe distributed coordination of heterogeneous robots through dynamic simple temporal networks. Master's thesis, MIT, Cambridge, MA.

Williams, B. C.; Ingham, M.; Chung, S.; and Elliott, P. 2003. Model-based programming of intelligent embedded systems and robotic explorers. In *IEEE Proceedings, Special Issue on Embedded Software*.