Robust Execution of Contingent, Temporally Flexible Plans *

Stephen A. Block, Andreas F. Wehowsky and **Brian C. Williams**
Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, MA, 02139, USA
sblock@mit.edu, andreas@wehowsky.dk, 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 plan extraction phase, by choosing between functionally redundant methods, and during their execution phase, by dispatching temporally flexible plans. Previous contingent 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 the plan extraction component of a robust, *distributed* executive for 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. To execute a TPN, the TPN is first distributed over multiple agents, by creating a hierarchical ad-hoc network and by mapping the TPN onto this hierarchy. Second, candidate plans are extracted from the TPN using a distributed, parallel algorithm that exploits the structure of the TPN. Third, the temporal consistency of each candidate plan is tested using a distributed Bellman-Ford algorithm. Each stage of plan extraction distributes communication to adjacent agents in the TPN, and in so doing eliminates communication bottlenecks. In addition, the distributed algorithm reduces the computational load on each agent. The algorithm is empirically validated on a range of randomly generated contingent plans.

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 requires performing robust execution of contingent, temporally flexible plans in a distributed manner. Methods have been developed for the dynamic execution (Morris & Muscettola 2000) of temporally flexible plans (Dechter, Meiri, & Pearl 1991). These methods adapt to failures that fall within the margins of the plan and hence add robustness to execution uncertainty.

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. To dynamically execute a TPN, Kirk continuously extracts a plan from the TPN that is temporally feasible, given the execution history, and dispatches the plan, using the methods of (Tsamardinos, Muscettola, & Morris 1998). Dynamic execution of contingent plans adds robustness to plan failure.

However, as a centralized approach, Kirk can be brittle to failures caused by communication limitations. This is because the agent performing planning must communicate the plan with all other agents before execution. This leads to a communication bottleneck at the lead agent.

We address this limitation through a distributed version of Kirk, which performs distributed dynamic execution of contingent temporally flexible plans. The distributed approach evens out communication requirements and reduces the brittleness to a single communication bottleneck.

This paper focuses on a method for dynamically selecting a feasible plan from a TPN. A subsequent method for performing distributed execution of the plan is presented in (Block, Wehowsky, & Williams 2006). Our key innovation is a distributed, hierarchical algorithm for searching a TPN for a feasible plan. This algorithm shares the computational load of plan extraction between all agents, thus reducing the computational complexity for each agent, and exploits parallelism through concurrent processing. Our plan selection algorithm, Distributed-Kirk (D-Kirk), is comprised of three stages.

1. Distribute the TPN across the processor network,
2. Generate candidate plans through distributed search on the TPN, and
3. Test the generated plans for temporal consistency.

*This work was made possible by the sponsorship of the DARPA NEST program under contract F33615-01-C-1896

This paper begins with an example TPN and an overview of the way in which D-Kirk operates on it. We provide a formal definition of a TPN and then discuss the three stages of D-Kirk. Finally, we discuss the complexity of the D-Kirk algorithm and present experimental results demonstrating its performance.

Example Scenario

In this section, we discuss at a high level the three step approach taken by D-Kirk to execute a contingent plan. Throughout this paper we use a simple, pedagogical example TPN to aid understanding of the key concepts.

This TPN is represented as a graph in Fig. 1, where nodes represent points in time and arcs represent activities. Each activity is labeled with upper and lower time bounds that represent the temporal constraints on its duration. A pathway through the TPN is a thread of activity. A node representing a non-deterministic choice for the target of its pathway is called a *choice node* and is shown as an inscribed circle.

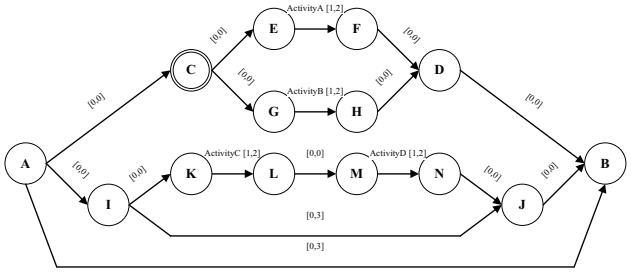

Figure 1: Example TPN

The plan is to be executed by a group of seven processors, $p1, \ldots, p7$. To perform distributed plan selection, the first step is to distribute the TPN itself over the processors. A TPN is built as a hierarchy of nested subnetworks, and we exploit this hierarchy when distributing the plan selection problem.

To facilitate TPN distribution, a leader election algorithm is used to arrange the processors into a hierarchy (Fig. 2). The hierarchical structure of the TPN is then used to map subnetworks to processors. For example, the master processor $p1$ handles the composition of two subnetworks of the plan at the start node (node A) and the end node (node B). It passes responsibility for each of the two main subnetworks to the two processors immediately beneath it in the hierarchy. Nodes *C,D,E,F,G,H* are passed to $p2$ and nodes *I,J,K,L,M,N* are passed to $p3$.

In the remaining two steps the processors work together to extract a plan from the TPN that is temporally consistent. Step two generates a candidate plan. This involves selecting a single thread of execution from the plan. This step coordinates in a hierarchical fashion, where each processor sends messages to its neighbors in the hierarchy corresponding to the subnetworks of the TPN. Neighbors are requested to make selections in the subnetworks for which they are responsible and in this way, selections are made concurrently. In the example, only the subnetwork owned by $p2$

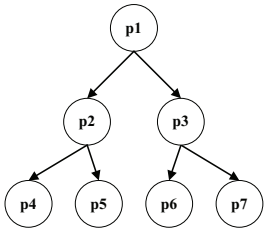

Figure 2: A three-level hierarchy formed by leader election

(nodes C,D,E,F,G,H) contains a choice, that is, $p2$ must decide between `ActivityA` and `ActivityB`, whereas $p3$ has no choice to make.

Having generated a candidate plan, the third and final step of D-Kirk is to test the candidate plan for consistency. Again, this is performed in a hierarchical fashion, where consistency checks are first made at the lowest level and successful sub-plan candidates are then checked at an increasingly higher level. For example, $p2$ and $p3$ simultaneously check that their subnetworks are internally consistent. If so, $p1$ then checks that the parallel execution of the two candidates is consistent.

Throughout the plan generation and consistency checking phases of D-Kirk, processors exchange messages to coordinate their activities. These messages are distributed between all processors, avoiding the communication bottleneck at the leader node which is present in a centralized architecture. In addition, candidate generation (Step 2) and consistency checking (Step 3) are interleaved, such that some of the processors are generating candidates while others are simultaneously checking consistency. This use of parallel processing improves the efficiency of the algorithm.

Temporal Plan Networks

A TPN is used to represent a contingent, temporally flexible plan. The primitive element of a TPN is an `activity`$[l, u]$, which is an executable command with a simple temporal constraint. The simple temporal constraint $[l, u]$ places a bound $t^+ - t^- \in [l, u]$ on the start time t^- and end time t^+ of the network 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), (Kim, Williams, & Abramson 2001). These operators are defined below. 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. 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.

- `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. Each subnetwork is referred to as a *child* subnetwork.
- `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. For a given subnetwork, the subnetwork following it in a `sequence` network is referred to as its *successor*.

Graph representations of the `activity`, `choose`, `parallel` and `sequence` network types are shown in Fig. 3. Nodes represent time events and directed edges represent simple temporal constraints.

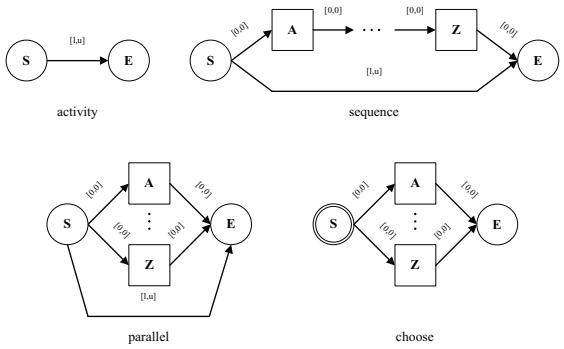

Figure 3: TPN Constructs

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.*

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**.*

TPN Distribution

Recall that the first step of D-Kirk is to distribute the input TPN over the available processors. The TPN may be provided by a high-level generative planner or by a human operator, but plan generation is outside the scope of this paper.

To support this and the subsequent two steps, the processors must first be able to communicate. To accomplish this, D-Kirk begins by establishing a hierarchical, ad-hoc communication network. In addition, an overall leader is selected, in order to initiate plan extraction and to communicate with the outside world.

Ad-Hoc Processor Network Formation

We use the leader election algorithm in (Nagpal & Coore 1998) to arrange the processors into a hierarchical network, an example of which is shown in Fig. 2. For each processing node, the node immediately above it in the hierarchy is its *leader*, those at the same level within that branch of the hierarchy are its *neighboring leaders* and those directly below it in the hierarchy are its *followers*. The leader election algorithm forms a hierarchy using a message passing scheme that ensures that every node can communicate with its leader, as well as with all neighboring leaders and followers. Note that the hierarchical communication structure lends itself well to distributing the TPN, which is also hierarchical. It may be possible to improve the performance of the plan selection algorithm by using the TPN structure to guide the formation of the processor hierarchy. This is an interesting direction for future work.

TPN Distribution over the Processor Network

A TPN is distributed by assigning to each processor the responsibility for zero or more nodes of a TPN graph, such that each node is assigned to exactly one processor. During the subsequent two steps of D-Kirk, each processor maintains all data relevant to its assigned nodes.

This distribution scheme requires that two processors be able to communicate whenever they have been assigned two TPN nodes that are linked by a temporal constraint. This requirement is satisfied by D-Kirk's TPN distribution algorithm, shown in Fig. 4. See (Wehowsky 2003) for further explanation. Note that the algorithm allows the TPN 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.

We now demonstrate this distribution algorithm on the TPN of Fig. 1 and the processor hierarchy of Fig. 2. The TPN is supplied from an external source, which establishes a connection with the top-level leader, *p1*. The TPN is a `parallel` network at the highest level, hence processor *p1* assigns the start and end nodes (nodes *A,B*) to itself (line 7). There are two subnetworks, which *p1* assigns to its two followers, *p2* and *p3* (lines 15-18). *p1* passes the `choose` subnetwork (nodes *C,D,E,F,G,H*) to *p2* and the `sequence` subnetwork (nodes *I,J,K,L,M,N*) to *p3*. *p2* and *p3* then process their subnetworks in parallel. *p2* assigns the start and end nodes (nodes *C,D*) to itself (line 7). This network has two subnetworks, which *p2* assigns to two of its followers, *p4* and *p5* (lines 15-18). *p2* passes `ActivityA` (nodes *E,F*) to *p4* and `ActivityB` (nodes *G,H*) to *p5*. Since activities can not be decomposed, *p4* and *p5* assign nodes *E,F* and *G,H*, respectively, to themselves (lines 3-4). Meanwhile, *p3* receives the `sequence` subnetwork and assigns the start and end nodes (nodes *I,J*) to itself (line 7). This network has two subnetworks, which *p3* assigns to two of its followers, *p6* and *p7* (lines 15-18). *p3* passes `ActivityC` (nodes *K,L*) to *p6* and `ActivityD` (nodes *M,N*) to *p7*. *p6* and *p7* then assign nodes *K,L* and nodes *M,N*, respectively, to themselves (lines 3-4).

```
1: wait for TPN
2: n ← number of followers of p
3: if TPN is of type activity then
4:    assign start and end nodes of TPN to p
5: else
6:    k ← number of subnetworks
7:    assign start and end nodes to p
8:    if n = 0 then
9:       if p has a neighbor leader v then
10:         send k/2 subnetworks of TPN to v
11:         assign k/2 subnetworks of TPN to p
12:      else
13:         assign TPN to p
14:      end if
15:   else if n ≥ k then
16:      for each of k subnetworks of TPN do
17:         assign subnetwork of TPN to a follower of p
18:      end for
19:   else if n < k then
20:      for each of n subnetworks of TPN do
21:         assign subnetwork to a follower of p
22:      end for
23:      assign remaining (k − n) subnetworks of TPN to p
24:   end if
25: end if
```

Figure 4: TPN Distribution Algorithm for node p

Candidate Plan Generation

Having distributed the TPN across the available processors, D-Kirk's second step generates candidate plans. These candidates correspond to different assignments to the choice variables at each choice node, and can be viewed as a restricted form of a conditional CSP (Mittal & Falkenhainer 1990). D-Kirk uses parallel, recursive, depth first search to perform these assignments. This use of parallel processing is one key advantage of D-Kirk over a traditional, centralized approach.

D-Kirk operates on three network types formed from the four types fundamental to a TPN. These are `activity`, `parallel-sequence` and `choose-sequence`, as shown in Fig. 5, where the subnetworks $A_i, \ldots, Z_i$ are of any of these three types. We handle the simple temporal constraint present on a `sequence` network by considering a `sequence` network as a special case of a `parallel-sequence` network, in which only one subnetwork exists.

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.

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.

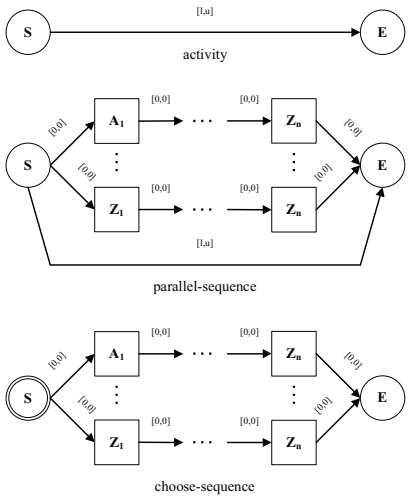

Figure 5: Constructs for D-Kirk

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 sections describe the actions carried out by the start node of each network type on receipt of a `findfirst` or `findnext` message. Pseudo-code for each section of the algorithm is shown in Figs. 7 through 11 in the appendix. A more detailed description of the algorithm, including a walk-through of the pseudo-code for an example TPN, can be found in (Wehowsky 2003).

Activity

During search, an `activity` 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. 7.

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 thus conducted in parallel. If any of the child subnetworks or 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. 8. 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).

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. 9.

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 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. 10.

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 18), 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 forwards the response from the successor network to its parent and the search terminates (line 12). If no successor network is present, the network is consistent and the start node returns `ack` to its parent (line 14). If all assignments to the 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 21).

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. 11.

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 8-18). Finally, if this strategy also fails, the start node attempts to find a new consistent assignment in any successor network, by sending a `findnext` message (lines 19-20).

Temporal Consistency Checking

In the third and final step, each plan generated by candidate generation must be tested for temporal consistency. Consistency checking is implemented using a distributed Bellman-Ford Single Source Shortest Path (SSSP) algorithm (Lynch 1997), 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). 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.

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. This completes the development of the three stages of D-Kirk involved in plan extraction.

Discussion

We conclude by discussing the performance of D-Kirk. In the centralized case, the time complexity of TPN plan extraction is determined by the complexity of the SSSP algorithm, used to detect temporal inconsistencies and the number of consistency checks that must be conducted. In the worst case, N^e checks must be made, where N is the number of nodes and e is the size of the domain of the choice variables. A centralized Bellman-Ford SSSP algorithm has time complexity NE, where E is the number of edges, giving an overall worst case time complexity of $N^{e+1}E$.

D-Kirk uses the distributed Bellman-Ford algorithm, whose computational complexity is linear in the number of nodes, since each node communicates only with its local neighbors. D-Kirk begins consistency checking at the deepest level in the TPN hierarchy, progressing to higher levels only if consistency is achieved. Therefore, in the worst case, N consistency checks are required for each permutation of choice variable assignments; one initiated at each node. However, the number of variables involved in each consistency check is reduced as the depth of the check is increased. If the graph hierarchy is assumed to be of depth m, with branching factor b, then a consistency check initiated at a node at depth x involves $O\left(b^{m-x}\right)$ nodes. Each node communicates with its $b+1$ neighbors, hence the time complexity is $O\left(\left(b+1\right)b^{m-x}\right)$. The number of searches at depth x is b^x, thus the total complexity is $O\left(\left(b+1\right)b^m\right)$. Averaging over all nodes, the computational complexity per node is $O\left(\left(b+1\right)b^{m(e+1)}\right)$.

Compared to the computational complexity of centralized plan extraction, $N^{e+1}E = b^{m(e+1)}b^{m-1}$, we see that D-Kirk is more efficient provided that $b+1 < b^{m-1}$; this is true for typical plans.

Furthermore, the time complexity of D-Kirk is likely to be far better than the worst-case estimates, since the fast consistency checks performed on smaller plans at deeper levels may prevent unnecessary, more expensive consistency checks at higher levels.

The run time performance of D-Kirk was tested empirically on randomly generated TPNs, using a C++ implementation that simulates execution on an array of processors. Exactly one TPN node was assigned to each processor and the number of nodes in each randomly generated TPN was varied between 1 and 100. For each TPN instance, the number of TPN constructs (`parallel`, `sequence` or `choose`) was varied between 3 and 30 and the maximum recursive depth was varied between 4 and 10. Run time was measured by the number of listen-act-respond cycles completed by the processor network.

Fig. 6 shows a plot of the average number of cycles against the average number of nodes. The results show that the variation in the number of cycles is roughly linear with the number of nodes. This suggests that in practice, the run time will be dominated by the distributed Bellman-Ford consistency checking algorithm, which is linear in the number of nodes, as opposed to backtrack search, which is exponential. The reason for this is that the worst-case exponential time complexity of D-Kirk occurs only when the TPN is composed entirely of `choose` subnetworks.

To summarize, this paper introduced the Distributed Temporal Planner (D-Kirk), the plan extraction component of a distributed executive that operates on contingent, temporally flexible plans. D-Kirk operates on a Temporal Plan Network (TPN), by distributing both data and algorithms across available processors. D-Kirk employs a series of distributed algorithms that exploit the hierarchical structure of a TPN to, first, form a processor hierarchy and assign TPN subnetworks to each processor; second, search the TPN for candidate plans, and, finally, check for candidate temporal consistency. This distributed approach spreads communication more evenly across the processors, helping to alleviate the communication bottleneck present in a centralized architecture. Furthermore, D-Kirk reduces the computational load on each processor and allows concurrent processing.

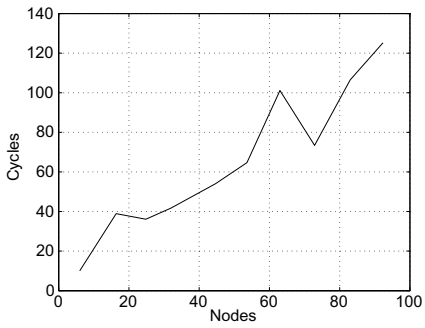

Figure 6: Number of cycles vs. number of nodes

References

Block, S. A.; Wehowsky, A.; and Williams, B. C. 2006. Robust execution of contingent, temporally flexible plans. In *ICAPS 2006 Workshop on Planning Under Uncertainty and Execution Control for Autonomous Systems*.

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)*.

Morris, P., and Muscettola, N. 2000. Execution of temporal plans with uncertainty. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-2000)*, 491–496.

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*.

Appendix : D-Kirk Pseudo-Code

```
1: parent ← sender of msg
2: for each child 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:   send fail to parent
14: else
15:   if check-consistency( v ) then
16:      send ack to parent
17:   else
18:      if search-permutations(v) then
19:         send ack to parent
20:      else
21:         send fail to parent
22:      end if
23:   end if
24: end if
```

Figure 7: `parallel-findfirst(node v)`

```
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 ← 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 8: `search-permutations(node v)` function

```
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:    send response to parent
7: else
8:    send fail to parent
9: end if
```

Figure 9: `parallel-findnext(node v)` function

```
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:         send response to parent
13:      else
14:         send ack to parent
15:      end if
16:      return
17:   else
18:      remove w from child list
19:   end if
20: end for
21: send fail to parent
```

Figure 10: `choose-findfirst()` function

```
1: w ← current assignment
2: send findnext to w
3: wait for response
4: if response = ack then
5:    send ack to parent
6:    return
7: end if
8: while w < child-n do
9:    w ← next child
10:   send findfirst to w
11:   wait for response
12:   if response = ack then
13:      send ack to parent
14:      return
15:   else
16:      remove w from child list
17:   end if
18: end while
19: if successor B exists then
20:   send findnext to B
21:   for w = child0 to child-n do
22:      choice variable ← w
23:      send findfirst to w
24:      wait for response from child w
25:      if response = ack then
26:         break
27:      end if
28:   end for
29:   wait for response from B
30:   send response to parent
31: else
32:   send fail to parent
33: end if
```

Figure 11: `choose-findnext()` function