Executing Reactive, Model-based Programs through Graph-based Temporal Planning

Phil Kim and **Brian C. Williams**
MIT Rm. 37-381
77 Massachusetts Ave.
Cambridge, MA 02139 USA
{kim,williams}@mit.edu

Mark Abramson
Draper Lab
555 Technology Square, MS3F
Cambridge, MA 02139 USA
mabramson@draper.com

Abstract

In the future, webs of unmanned air and space vehicles will act together to robustly perform elaborate missions in uncertain environments. We coordinate these systems by introducing a *reactive model-based programming language (RMPL)* that combines within a single unified representation the flexibility of embedded programming and reactive execution languages, and the deliberative reasoning power of temporal planners. The KIRK planning system takes as input a problem expressed as a RMPL program, and compiles it into a *temporal plan network (TPN)*, similar to those used by temporal planners, but extended for symbolic constraints and decisions. This intermediate representation clarifies the relation between temporal planning and causal-link planning, and permits a single task model to be used for planning and execution. Such a unified model has been described as a holy grail for autonomous agents by the designers of the Remote Agent[Muscettola *et al.*, 1998b].

1 Model-based Programming

The recent spread of advanced processing to embedded systems has created vehicles that execute complex missions with increasing levels of autonomy, in space, on land and in the air. These vehicles must respond to uncertain and often unforgiving environments, both with a fast response time and with a high assurance of first time success. The future looks to the creation of *cooperative robotic networks*. For example, a heterogenous collection of vehicles, such as planes, helicopters and boats, might work in concert to perform a search and rescue during a hurricane or similar natural disaster. In addition, giant space telescopes are being deployed that are composed of satellites carrying the telescope's different optical components. These satellites act in concert to image planets around other stars, or unusual weather events on earth.

The creation of robotic networks cannot be supported by the current programming practice alone. Recent mission failures, such as the Mars Climate Orbiter and Polar Landers, highlight the challenge of creating highly capable vehicles within realistic budget limits. Due to cost constraints, spacecraft flight software teams often do not have time to think through all the plausible situations that might arise, encode the appropriate responses within their software and then validate that software with high assurance. To break through this barrier we need to invent a new programming paradigm.

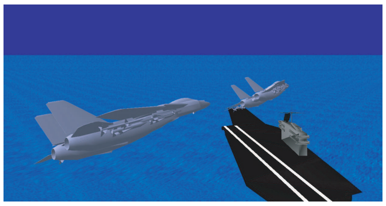
In this paper we advocate the creation of *embedded, model-based programming languages*. First, programmers should retain control for the overall success of a mission, by programming game plans and contingencies that in the programmer's experience will ensure a high degree of success. The programmer should be able to program these game plans using features of the best embedded programming languages available. For example, reactive synchronous languages[Halbwachs, 1993], like Esterel, Lustre and Signal, offer a rich set of constructs for interacting with sensors and actuators, for creating complex behaviors involving concurrency and preemption, and for modularizing these behaviors using all the standard encapsulation mechanisms. Model-based programming extends this style of reactive language with a minimal set of constructs neccessary to perform flexible mission coordination, while hiding its reasoning capabilities under the hood of the language's interpreter or compiler.

Second, we argue that model-based programming languages should focus on elevating the programmer's thinking, by automating the process of reasoning about low-level system interactions. Many recent space mission failures, such as Mars Climate Orbiter and Mars Polar Lander, can be isolated to difficulties in reasoning through low-level system interactions. On the other hand, this limited form of reasoning and book keeping is the hallmark of computational methods. The interpreter or compiler of a model-based program reasons through these interactions using composable models of the system being controlled. We are developing a language, called the *Reactive Model-Based Programming Language (RMPL)*, that supports four types of reasoning about system interactions: reasoning about contingencies, scheduling, inferring a system's hidden state and controlling that state. This paper develops RMPL in the context of contingencies and scheduling, while [Williams *et al.*, 2001], shows how RMPL is used to infer hidden state.

RMPL offers a middle ground between execution languages, like RAPS [Firby, 1995], and highly flexible, operator-based temporal planners,like HSTS [Muscettola *et al.*, 1998a]. RAPS offers the exception handling and concurrency mechanisms of embedded languages, while adding goal monitoring, nondeterministic choice and metric constraints. However, RAPS makes its decisions reactively, without addressing concerns of schedulability and threat resolution, and hence can fall into a failure state. RMPL incorporates the forward looking planning and scheduling abilities of modern temporal planners, but can severely restrict the space of plans considered to possible threads of execution through the RMPL program. This speeds response and mitigates risk.

The paper begins by introducing a subset of RMPL that includes constructs from traditional reactive programming plus constructs for specifying contingencies and scheduling constraints. Second, we describe how *Kirk*, an RMPL-based planner/executive, compiles RMPL programs into *temporal plan networks (TPN)*, which compactly represent all possible threads of execution of an RMPL program, and all resource constraints and conflicts between concurrent activities. Third, we present Kirk's online planning algorithm for RMPL that "looks" by using network search algorithms to find threads of execution through the TPN that are temporally consistent. The result is a partially ordered temporal plan. Kirk then "leaps" by executing the plan using plan execution methods[Tsamardinos *et al.*, 1998] developed for Remote Agent[Muscettola *et al.*, 1998b]. Finally, we discuss Kirk's application to a simulated search and rescue mission.

2 Example: Cooperative Search and Rescue



As part of a search and rescue mission, consider an activity called Enroute, in which a group of vehicles fly together from a rendezvous point to the target search area. In this activity, the group selects one of two paths for traveling to the target area, flies together along the path through a series of waypoints to the target position, and then transmits a message to the forward air controller to indicate their arrival, while waiting until the group receives authorization to engage the target search area.

The two paths available for travel to the target area are each only available for a predetermined window of time, which is important to consider when selecting one of these paths. In addition, the timing of the Enroute activity is bound by externally imposed requirements, for example, the search and rescue mission must complete in 25-30 minutes, with 20% to 30% of the time allotted to the Enroute activity.

Codifying the Enroute activity requires most standard features of embedded languages. There are both sequential and concurrent threads of activities, such as going to a series of way points, and sending a message to the forward air controller (FAC), while concurrently awaiting authorization. There are maintenance conditions and synchronizations. For example, the air corridor needs to be maintained safe during flight, and synchronization occurs with the FAC.

In addition to constructs found in traditional embedded languages, we need constructs for expressing timing requirements and alternative choices or contingencies, in this example to use one of two corridors. These constructs are common to robotic execution languages[Firby, 1995]. However, they are only used reactively. Kirk must reason forward through the RMPL program's execution, identifying a course of action that is consistent.

3 RMPL Constructs

To summarize, RMPL needs to include constructs for expressing concurrency, maintaining conditions, synchronization, metric constraints and contingencies. The relevant RMPL constructs are as follows. We use lower case letters, like c, to denote activities or conditions, and upper case letters, like A and B, to denote well-formed RMPL expressions:

a. Invokes primitive activity a, starting at the current time. This is the basic construct for initiating activities.

c. Asserts that condition c is true at the current time, where c is a literal. This is the basic construct for asserting conditions.

if c **thennext** A. Starts executing A if condition c is currently satisfied, where c is a literal. This is the basic construct for expressing conditional branches and asserting preconditions.

do A **maintaining** c. Executes A, and ensures throughout A that c occurs. This is the basic construct for introducing maintenance conditions and protections.

A, B. Concurrently executes A and B. It is the basic construct for forking processes.

$A; B$. Consecutively executes A and then B. It is the basic construct for sequential processes.

$A[l, u]$. Constrains the duration of program A to be at least l and at most u. This is the basic construct for expressing timing requirements.

choose $\{A, B\}$. Reduces non-deterministically to program A or B. This is the basic construct for expressing multiple strategies and contingencies.

Note that together, c and **if** c **thennext** A provide the basic constructs for synchronization, by specifying required and asserted conditions. A, B and $A; B$ provide the neccessary constructs for building complex concurrent threads.

The "do maintaining" construct offers a building block for creating complex preemption and exception handling mechanisms. Note that to fully exploit these mechanisms Kirk would need to perform conditional planning. The algorithms presented in this paper only address unconditional planning. With this restriction "do maintaining" acts as a maintenance condition that Kirk must prove holds at planning time.

Using these constructs we express the Enroute activity as follows:

```
Group-Enroute()[l,u] = {
   choose {
      do {
         Group-Fly-Path(PATH1_1,PATH1_2,
            PATH1_3,TAI_POS)[l*90%,u*90%];

      } maintaining PATH1_OK,
      do {
         Group-Fly-Path(PATH2_1,PATH2_2,
            PATH2_3,TAI_POS)[l*90%,u*90%];
      } maintaining PATH2_OK
```

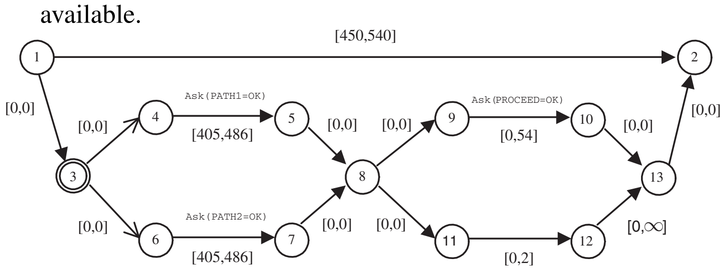
```
   };
   {
      Group-Transmit(FAC,ARRIVED_TAI)[0,2],
      do {
         Group-Wait(TAI_HOLD1,TAI_HOLD2)
            [0,u*10%]
      } watching PROCEED_OK
   }
}
```

The *choose* expression models the two options for flight paths. 90% of the total time of the overall maneuver is allocated to this group flight. Each flight has a maintenance condition that the flight path is okay. Arrival is transmitted to the forward air controller, and receipt of a message to proceed is concurrently monitored.

4 Temporal Plan Networks

Executing an RMPL program involves choosing a set of threads of execution (*Plans*), checking to ensure that the execution is consistent and schedulable, and then scheduling events on the fly. It is essential that we generate these plans quickly. This suggests compiling RMPL programs to a plan graph, along the lines of Graphplan or Satplan [Weld, 1999], and then searching the precompiled graph. However, it is also important for the plan to have the temporal flexibility offered by a partially ordered, temporal plan. Least committment leaves slack to adapt to execution uncertainties and to recover from faults. This partial committment is expressed in temporal planning through a *Simple Temporal Network (STN)*[Dechter *et al.*, 1991]. Hence, a key observation of our approach is that to build in temporal flexibility we should build our graph-based plan representation, called a *Temporal Plan Network (TPN)*, as a generalization of an STN.

The TPN corresponding to the above Enroute program is shown below. Activity name labels are omitted to keep the figure clear, but the node pairs 4,5 and 6,7 represent the two Group-Fly-Path activities, and node pairs 9,10 and 11,12 correspond to the Group-Wait and Group-Transmit activities, respectively. Node 3 is a decision node that represents a choice between two methods for flying to the search area. The TPN represents the consequences of the constraint that the mission last between 25 and 30 minutes. It also models the decision between the two paths to the target area, and it models the restrictions that each of the paths can only be used if they are available.



A TPN encodes all feasible executions of an activity. It does this by augmenting an STN with two types of constraints: temporal constraints restrict the behavior of an activity by bounding the duration of an activity, time between activities, or more generally the temporal distance between two events. Symbolic constraints restrict the behavior of an activity by expressing the assertion or requirement of certain conditions by activities that all valid executions must satisfy.

For example, consider some of the possible executions of the Enroute activity. One possible execution is that the group flies along path one (pair 4,5) to the target area in 420 time units (seconds in this case), transmits an arrival message to the forward air controller (11,12) for one second, and concurrently waits (9,10) for another 40 seconds to receive authorization to proceed. Another possible execution is that the group selects the second path, flies to the target area in 500 seconds, takes 2 seconds to transmit the arrival message, and is authorized to proceed immediately. If it were the case that path one was available from the time at which the Enroute activity started to at least the time that the group arrived at the target area, then the first execution is valid. This is because it satisfies both the temporal constraints on the Enroute activity, and the requirement that path one is available for the duration of the flight along it. The planning algorithm presented in the next section performs the identification of consistent activity executions.

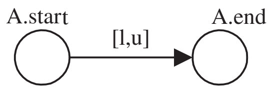
A Temporal Planning Network is a Simple Temporal Network, augmented with *symbolic constraints* and *decision nodes*. These additions are sufficent to capture all RMPL constructs given earlier. Like a simple temporal network, the nodes of a TPN represent temporal events, and the arcs represent temporal relations that constrain the temporal distance between events. An arc of a TPN may be labeled with a symbolic constraint Tell(c) or Ask(c), as well as a duration. A Tell(c) label on an arc (i,j) asserts that the condition represented by c is true over the interval between the temporal events modeled by the nodes i and j. Similarly, an Ask(c) label on an arc (i,j) requires that the condition represented by c is true over the interval represented by this arc. For example, in the Enroute TPN, the Ask(PATH1=OK) label on the arc (4,5) represents the requirement for path one to be available for the interval of time corresponding to the interval of time between the temporal event modeled by node 4 and node 5. These Ask-type symbolic constraints allow for the encoding of conditions in the network.

Decision nodes are used to explicitly introduce choices in activity execution that the planner must make. For example, in the Enroute activity there are two choices of paths for the group to use for flying to the target area, path one and path two. The activity model captures the two choices as out-arcs of node 3 of the enroute TPN. This decision node is designated by a double outline and dashed out-arcs. All other nodes in the Enroute TPN are non-decision nodes.

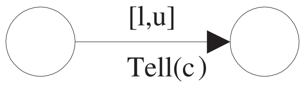
5 Compiling RMPL to TPN

Given a well formed RMPL expression, we compile it to a TPN by mapping each RMPL primitive to a TPN as defined below. RMPL sub-expressions, denoted by upper case letters, are recursively mapped to equivalent TPN:

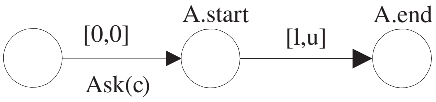
$A[l, u]$. Invoke activity A between l and u time units.



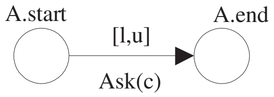
$c[l, u]$. Assert that condition c is true now until $[l, u]$.



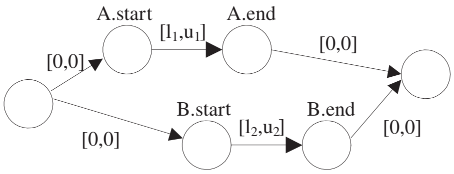
if c **thennext** $A[l, u]$. Execute A for $[l, u]$, if condition c is currently satisfied.



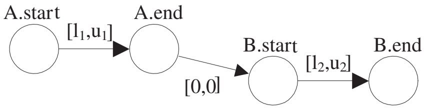
do $A[l, u]$ **maintaining** c. Execute A for $[l, u]$, and ensure throughout A that c occurs.



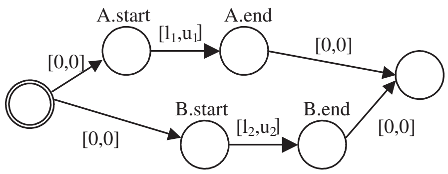
$A[l_1, u_1], B[l_2, u_2]$. Concurrently execute A for $[l_1, u_1]$ and B for $[l_2, u_2]$.



$A[l_1, u_1]; B[l_2, u_2]$. Execute A for $[l_1, u_1]$, then B for $[l_2, u_2]$.



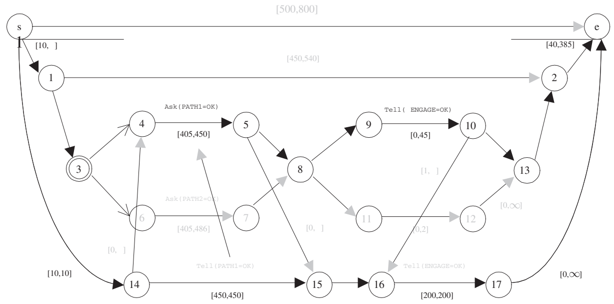
choose $\{A[l_1, u_1], B[l_2, u_2]\}$. Reduces to $A[l_1, u_1]$ or $B[l_2, u_2]$, non-deterministically.



6 Planning using TPNs

After compiling an RMPL program into a TPN, Kirk's planner uses the TPN to search for an execution that is both complete and consistent. The execution corresponds to an unconditional, temporal plan. A plan is complete if choices have been made for each relevant decision point, it contains only primitive-level activities, and all activities labeled Ask(c) have been linked to a Tell(c). A plan is consistent if it does not violate any of its temporal constraints or symbolic constraints. The resulting plan is then executed using the plan runner described in [Tsamardinos *et al.*, 1998].

The input to Kirk's planner is a TPN describing an activity scenario. A scenario consists of the TPN for the top-level activity invoked and any constraints on its invocation. The following TPN invokes Enroute (nodes 1-13). In a parallel thread it constrains the time ranges over which path one is available (nodes 14-15) and over which the vehicles may perform search (nodes 16-17).



The output of the planner consists of a set of paths through the input network from the start-node to the end-node of the top-level activity. In the example the paths s-1-3-4-5-8-9-10-13-2-e and s-14-15-16-17-e define a consistent execution. The first path defines the execution of the group of vehicles, and the second path defines the "execution" of the rest of the world in terms of the assertion or requirement of relevant conditions over the duration of the scenario. The portion of the TPN not selected for execution is shown in gray.

Planning involves two interleaved phases. The first phase resembles a network search that discovers the sub-network ,that constitute a feasible plan, while incrementally checking for temporal consistency. The second phase is analogous to the repair step of a causal link planner, in which threats are detected and resolved, and open conditions are closed[Weld, 1994].

6.1 Phase One: Select Plan Execution

The first phase selects a set of paths from the start-node to the end-node of the top-level activity. The planner handles this execution selection problem as a variant of a network search[Ahuja *et al.*, 1993] rooted at the start-node of the TPN encoding of the top-level activity.

Searching the Network

Recall that each node of a TPN is either a decision node or a non-decision node. If a plan includes a non-decision node with multiple out-arcs, then all of these arcs and their tail nodes must be included in the plan. If a plan includes a decision node with multiple out-arcs, then the arcs represent alternate choices, and the planning algorithm selects exactly one to be included in the plan.

Network search completes only when all paths reach the end-node of the top-level activity, and the subnetwork of the TPN, defined by these paths, is temporally consistent. This corresponds to testing consistency of an STN[Dechter *et al.*, 1991], as discussed in the next section.

The first phase of planning is summarized by the *Modified Network Search algorithm,* shown below. The set A, is the set of active nodes, which are those nodes whose paths have not yet been fully extended. The sets SN and SA are the sets of selected nodes and selected arcs, respectively:

```
Modified-Network-Search( N )
   A = { start-node of N };
   SN = { start-node of N };
   SA = { };
   While ( A is not empty )
```

```
6       Node = Select and remove a member of A;
7       If ( Node is a decision-node )
8          Arc = Select any unmarked out-arc of Node and
9          Mark Arc and
10          Add Arc to SA;
11          If ( tail of Arc is not in SN  )
12             Add tail of Arc to A and SN;
13          End-If
14       Else
15          For each Arc that is an out-arc of Node
16             Add Arc to SA;
17             If ( tail of Arc is not in SN  )
18                Add tail of Arc to A and SN;
19             End-If
20          End-For
21       End-If
22
23       If ( Cycle-Induced(SN, SA) )
24          If ( Not(Temporally-Consistent(SN, SA)) )
25             Backtrack(SN, SA, A);
26          End-If
27       End-If
28    End-While
29 End-Function
```

The algorithm extends an active node at each iteration. Decision nodes are treated by extending the path along one out arc (lines 8-13), while non-decision nodes are treated by branching the path and extending along all out arcs (lines 15-20). At the end of each iteration of the main While-loop, the modified network search tests for temporal consistency (lines 24-26). If the test fails, then the search calls Backtrack(..) in line 25, which reverts SN, SA, and A to their states before the most recent decision that has unmarked choices remaining, and selects a different out-arc. While for simplicity this explanation uses chronological backtracking, a wealth of more efficient search algorithms can be applied.

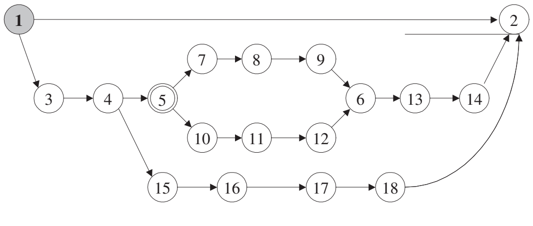
Note that it is not necessary to check temporal consistency after every iteration of the While-loop, since as long as no cycles are induced in the network, there is no way for a temporal inconsistency to be induced. Determining whether a cycle has been created can be done for each arc that is selected by checking whether the arc's tail node has already been selected. Since this can be done in constant time, it is significantly more efficient in practice than testing temporal consistency after every iteration, although it doesn't impact worst case complexity.

Also note that the algorithm stops extending a path when it encounters a node that is already in SN. The fact that this node is already in SN implies that two concurrent threads of execution have merged.

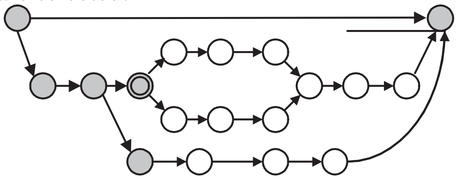
Finally, after the modified network search completes, the selected nodes and arcs define a set of paths from the start-node to the end-node of the top activity.

Example:Searching the Enroute Network

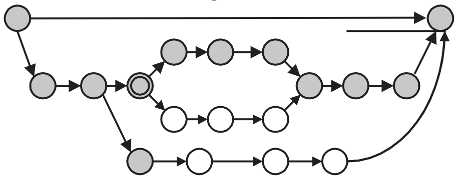
To illustrate the modified network search, we return to the Enroute input network, where node 1 is the start-node and node 2 is the end-node:



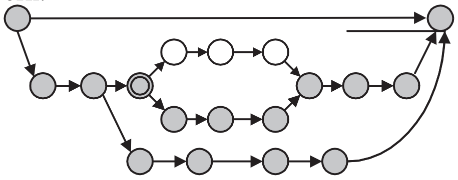
Initially, node 1 is selected, which is indicated by its darker shade, and it is active. In the first iteration, Kirk chooses node 1 from the set of active nodes, and since node 1 is not a decision node, it selects all out-arcs and adds their tails to the selected and active set. This continues until both node 5 and node 15 are selected:



At this point, the modified network search chooses node 5 from the active set. Since node 5 is a decision node, the algorithm must choose either arc (5,7) or arc (5,10). It selects arc (5,7) and continues extending until it reaches the following:



Note that arc (14,2) is selected, forming the cycle, 1-3-4-5-7-8-9-6-13-14-2-1, so the algorithm checks for temporal consistency. In this example, this selected sub-network is temporally inconsistent, so the algorithm backtracks to the most recent decision with open options, which is Node 5. Out-arc (5,10) has not yet been tried, so it is selected and the path extend to the end-node. Finally a path through arc (15,16) is found to the end-node, resulting in the temporally consistent sub-network:



Checking Temporal Consistency

To check temporal consistency we note that any subnet of a Plan Network, minus its symbolic constraint labels, forms a Simple Temporal Network. Hence temporal consistency can be checked using standard methods for Simple Temporal Networks [Dechter *et al.*, 1991]. Recall that an STN is consistent if and only if its encoding as a distance graph contains no negative cycles [Dechter *et al.*, 1991]. There exist several well known algorithms for detecting negative cycles in polynomial time. The Bellman-Ford algorithm [Cormen *et al.*, 1990] can be used to check for negative cycle in $O(nm)$ time, where m and n are the number of arcs and nodes in the distance graph, respectively. This algorithm only needs to maintain one distance label at each node, which takes only $O(n)$ space. A variant of this algorithm is used by HSTS [Muscettola *et al.*, 1998a] for fast inconsistency detection.

The algorithm we use in the Kirk planner is a variant of the generic label-correcting single-source shortest-path algorithm [Ahuja *et al.*, 1993], which takes $O(nm)$ worst-case asymptotic running time, but performs faster in many situations. This algorithm also requires only $O(n)$ space. Space precludes a more detailed development.

6.2 Phase Two: Threats and Open Conditions

Symbolic constraints– Ask(c) and Tell(c) – are handled analogous to threats and open conditions in causal link planning[Weld, 1994]. Two symbolic constraints conflict if one is either asserting (by using Tell) or requesting (by using Ask) that a condition is true, and the second is asserting or requesting that the same condition is false. For example, Tell(Not(c)) and Ask(c) conflict. An open condition in a TPN appears as Ask constraints, which represent the need for some condition to be true over the interval of time represented by the arc labeled with the Ask constraint.

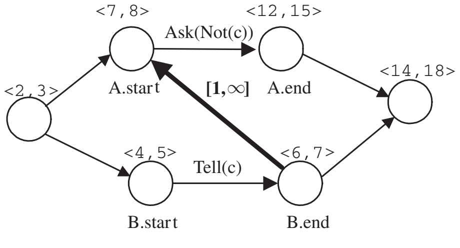
Resolving Threats

To detect threats the planner computes the feasible time bounds for each temporal event (node) in the network, and then uses these bounds to identify potentially overlapping intervals that are labeled with inconsistent constraints. These bounds can be computed by solving an all-pairs shortest-path problem over the distance graph of the partially completed plan. Kirk uses the Floyd-Warshall algorithm for computing all-pairs shortest paths. We are currently evaluating Johnson's algorithm which runs in $O(n^2 \log(n)+mn)$, or $O(n^2 \log(n))$ if $m = O(n)$.

Once these feasible time ranges are determined, the planner detects which arcs may overlap in time. If there are two arcs that may overlap and that are labeled with conflicting symbolic constraints, then they are resolved by ordering the intervals, if possible.

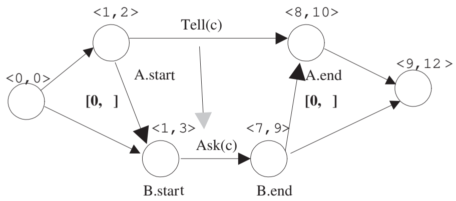
These interval pairs need to be identified efficiently. Kirk maintains an interval set data structure for each proposition p that keeps track of all intervals that assert or require p or its negation. In order to identify threats, the planner need only check each interval set for threats. This takes $O(si^2)$ asymptotic running time, where i is the maximum cardinality over all interval sets, and performs much better in practice because the interval sets typically have few elements. More sophisticated indexing schemes may improve performance, such as interval tree structures [Cormen *et al.*, 1990].

A threat is resolved by introducing temporal constraints. Each threat consists of two arcs that represent intervals of time that may overlap. To resolve threats we introduce a constraint that forces an ordering between the two activites, similar to promotion and demotion in classical planning[Weld, 1994]:



Closing Open Conditions

An open condition is represented by an arc labeled with an Ask constraint, which represents the request for a condition to be satisfied over the interval of time represented by the arc. If this interval of time is contained by another interval over which the condition is asserted by a Tell constraint, then the open condition is satisfied (i.e., closed), and a causal link is drawn from the Tell to the Ask. Open conditions are detected simply by scanning through all activites and checking any Ask constraints. Finding potentially overlapping intervals is performed using the same method described above for detecting threats. Once a Tell is found that can satisfy an open condition, temporal constraints are added so that the duration of the open condition is contained within the Tell. This method of closing open asks is also closely related to the way that the HSTS planner satisfies compatibilities [Muscettola *et al.*, 1998a]:



7 Implementation and Discussion

Kirk's compiler generates TPN specification files, and is written in Lisp. Kirk's planner, written in C++, generates a plan from the TPN and checks consistency. Kirk's executive, based on the remote agent plan runner [Tsamardinos *et al.*, 1998], takes the resulting partially ordered temporal plan and executes it on the multi-air vehicle simulator. The following table summarizes Kirk's performance on nominal plans for several activities within the search and rescue scenario. The fully expanded TPN generated from the Group-Search-and-Rescue activity included 273 nodes. The testing platform was an IBM Aptiva E6U with an Intel 400Mhz Pentium II processor and 128MB of RAM, running Redhat Linux version 6.1:

Top Activity	Nodes	Activities	Plan Time
Follow(..)	4	1	4 ms
Group-Rescue(..)	27	8	235 ms
Group-Enroute()	112	19	16 s
Group-SR-Mission()	273	47	404 s

"Top Activity" refers to the top-level activity that was being planned. "Nodes" is the size of the expanded TPN after planning. Usually, about half of these were included in the final plan, with the rest corresponding to unselected executions. "Activities" indicates the number of primitive activities included in the final plan. Finally, the "Plan Time" gives the time that it took for Kirk to generate a plan corresponding to each of these activities.

Kirk offers two sources for efficiency. First, typically an RMPL program significantly constrains the space of possible plans considered, in the spirit of hierarchical task network planners [Erol *et al.*, 1994]. Second, the use of TPNs reduces online planning to graph search. In the example Kirk does well with no search guidance up to about 100 nodes. At this point the time becomes dominated by the time required to compute feasible time bounds for events. This is due to the use of Bellman-Ford and chronological search. We are exploring a reimplementation based on Johnson's algorithm and a more sophisticated search strategy.

The primary contribution of this paper is the Reactive Model-based Programming Language and the Temporal Plan Network representation. The algorithms presented here only begin to explore RMPL/TPN-based planning. The following are some example directions for further research.

This paper focuses on the use of TPNs as a synthesis of causal link planning[Weld, 1994], temporal planning [Muscettola, 1994] and hierarchical task network planning[Erol *et al.*, 1994]. Can methods from graph-based planning[Blum and Furst, 1997; Weld, 1999; Smith and Weld, 1999], particularly mutual exclusion relationships, be effectively employed within a TPN? An important element of practical temporal planners in the space domain, such as HSTS[Muscettola, 1994] and IxTeT[Laborie and Ghallab, 1995], is the ability to plan with depletable resources. Can RMPL and TPNs be similarly extended? How can RMPL and TPNs be extended to support decision theoretic planning and agile maneuver planning, common to robotic vehicles?

RMPL offers an expressive embedded programming language, by inheriting most of its primitive combinators from the Timed Concurrent Constraint Language (TCC) [Saraswat *et al.*, 1996]. For example, as with TCC, these primitives allow a rich set of operators to be derived for preemption and exception handling, similar to those found in embedded languages like Esterel[Berry and Gonthier, 1992]. However, the algorithm presented here performs unconditional planning, and hence only considers the case where exceptions can be prevented. RMPL's ability to express exception handling mechanisms can best be exploited through the development of conditional planning algorithms.

Finally, RMPL allows the programmer to constrain the family of possible behaviors that the planner considers when controlling an embedded system. It is important that this family of behaviors be safe. Embedded languages like Esterel[Berry and Gonthier, 1992], Lustre[Halbwachs *et al.*,] and Signal[Guernic *et al.*,] offer a clean semantics, and offer support for direct machine verification of safety and liveness properties. The verification of RMPL programs would be similar, but requires methods, such as timed automata verification, that support metric constraints and non-determinism.

Acknowledgments

We would like to thank Michael Hofbaur, Tony Abad and the anonymous reviewers for their invaluable insights. This research is supported in part by the Office of Naval Research under contract N00014-99-1-1080 and by the DARPA MOBIES program under contract F33615-00-C-1702.

References

[Ahuja *et al.*, 1993] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[Berry and Gonthier, 1992] G. Berry and G. Gonthier. The *esterel* programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, November 1992.

[Blum and Furst, 1997] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[Cormen *et al.*, 1990] T. Cormen, C Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Camb., MA, 1990.

[Dechter *et al.*, 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *AIJ*, 49:61–95, 1991.

[Erol *et al.*, 1994] K. Erol, J. Hendler, and D. Nau. Htn planning: Complexity and expressivity. In *Proceedings of AAAI-94*, pages 1123–1128, 1994.

[Firby, 1995] R. James Firby. The RAP language manual. Technical Report AAP-6, Univ. Chicago, March 1995.

[Guernic *et al.*,] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with *signal*. pages 1321–1336.

[Halbwachs *et al.*,] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language *lustre*. pages 1305–1320.

[Halbwachs, 1993] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic, 1993.

[Laborie and Ghallab, 1995] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of IJCAI-95*, 1995.

[Muscettola *et al.*, 1998a] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in temporal reasoning for autonomous control systems. In *Autonomous Agents*, 1998.

[Muscettola *et al.*, 1998b] N. Muscettola, P. Nayak, B. Pell, and B. C. Williams. The new millennium remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

[Muscettola, 1994] N. Muscettola. HSTS: Integrating planning and scheduling. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[Saraswat *et al.*, 1996] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *J Symb Comp*, 22(5-6):475–520, 1996.

[Smith and Weld, 1999] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proceedings of IJCAI-99*, 1999.

[Tsamardinos *et al.*, 1998] I. Tsamardinos, N. Muscettola, and P. Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of AAAI-98*, 1998.

[Weld, 1994] D. Weld. An introduction to least commitment planning. In *AI Magazine*, 1994.

[Weld, 1999] D. Weld. Recent advances in ai planning. In *AI Magazine*, 1999.

[Williams *et al.*, 2001] B. C. Williams, S. Chung, and V. Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *Proceedings of IJCAI-01*, 2001.