Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration

Brian C. Williams, Phil Kim, Michael Hofbaur, Jonathan How,
Jon Kennell, Jason Loy, Robert Ragno, John Stedl and Aisha Walcott
MIT Space Systems and Artifical Intelligence Laboratories
77 Massachusetts Ave., Cambridge, MA 02139 USA
{williams,philkim,hofbaur,jhow,jonk,jloy,rjr,stedl,aisha}@ mit.edu

Keywords: cooperating autonomous agents, planning and execution, embedded programming.

Abstract

In the future webs of unmanned vehicles will act together to robustly achieve elaborate missions within uncertain environments. This web may be a distributed satellite system forming an interferometer, or may be a heterogenous set of rovers and blimps exploring Mars. 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. To support fast mission planning as graph search, the KIRK planner compiles an RMPL program into a *temporal plan network (TPN)*, similar to those used by temporal planners, but extended for symbolic constraints and decisions. To robustly coordinate air vehicle or rover maneuvers we combine the Kirk planning algorithm with randomized algorithms for kinodynamic path planning. Finally, we describe our Mars exploration testbed, including four RWI ATRV vehicles.

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, 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. In addition, the 2000 Mars Program Independent Assessment Team recommended an exploration architecture that adopts a more global view. For example, a heterogenous set of vehicles, such as orbiters, rovers and blimps might work in concert to identify and evaluate sites of greatest scientific interest.

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* that support the ability to specify global strategies for multi-vehicle coordination. First, we argue that the programmer 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[5], 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 [15], shows how RMPL is used to infer hidden state.

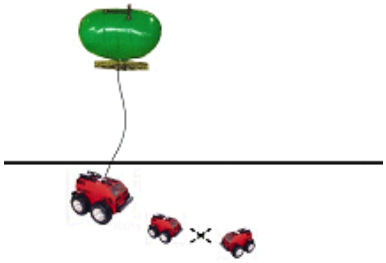
Third, execution of these programs is a form of mission-level planning that searches through the options for the optimal strategy that operates within the time constraints. Vehicle movement is central to these missions, hence to achieve global optimality we unify mission-level planning with global path planning algorithms. In particular we build upon the rapidly exploring random tree (RRT) algorithm of La Valle[8]. In addition, these vehicles may operate in highly dynamic situations or situations where maneuvering is extremely tight.

RMPL offers a middle ground between execution languages, like RAPS [4], and highly flexible, operator-based temporal planners,like HSTS [10]. 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 substantially restrict the space of plans considered to possible threads of execution through the RMPL program. This speeds response and mitigates risk.

To develop the model-based programming paradigm in the context of Mars exploration, we have developed a combination of simulation and ground-based testbeds, including a collection of four RWI ATRV rovers and a set of formation flying spacecraft, called Spheres, to be flown on space station.

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[12] developed for Remote Agent[11]. Fourth, we develop the unification of Kirk with randomized, kino-dynamic path planning algorithms. We present Kirk in the context of a simple Mars exploration mission segment.

2 Example: Cooperative Exploration



Consider a Mars exploration mission, where a team of rovers, blimps and orbiters explore several sites of interest (target areas). Orbiters perform initial surveillance, resulting in a course site map. Agile Scout rovers then refine the site map with high resolution data for local regions. Scouts are provided high resolution instruments for path planning and obstacle avoidance, and low resolution science instruments for initial site evaluation. Finally, laboratory rovers are guided to the most promising sites of interest. These rovers include additional capabilities, such as drills for subsurface exploration, that are used for sample collection.

As an example, a description of the Mars exploration mission using RMPL would include an *Enroute* activity, that specifies the group movement of scout rovers between sites of interest. In this *Enroute*

activity, the group selects one of two possible paths for traveling to the target area, and then traverses together along the path through a series of waypoints to the target position. Upon arrival the group leader transmits a message to the rover operations team at the mission control center (MCC) to indicate their arrival and awaits authorization from MCC before initiating science exploration of the target area.

Enroute includes a series of timing constraints. For example, the two available paths can be traversed only during certain time windows, due to the need for sufficient lighting to recharge the rovers' photovoltaic cells. Enroute might also be bound by mission constraints imposed at a higher level, such as, the requirement that scout exploration completes in 25-30 hours, with 30% of the time allotted to the Enroute activity.

Codifying the Enroute activity requires most standard features of embedded languages. Enroute includes both sequential and concurrent threads of activities, such as going to a series of way points, and sending a message to the mission control center (MCC), while concurrently awaiting authorization. There are maintenance conditions and synchronizations. For example, the selected path needs to be maintained safe during traverse, and synchronization occurs with the MCC.

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 paths. These constructs are common to robotic execution languages[4]. 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.

do A **watching** c. Executes A until condition c becomes true.

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.

Together, c and **if** c **thennext** A provide basic constructs for synchronization, by specifying required and asserted conditions. A, B and $A; B$ provide constructs for building complex concurrent threads and **do**...**maintaining** acts as a maintenance condition that Kirk must prove at planning time.

Using these constructs we express the Enroute activity as follows:

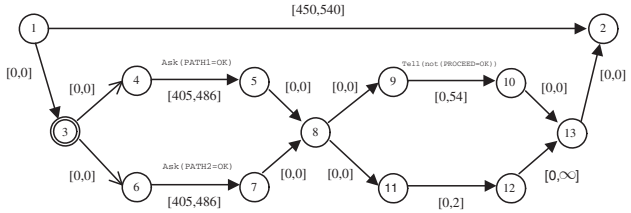
```
Group-Enroute()[l,u] = {
   choose {
      do {Group-Traverse-Path(PATH1_1,PATH1_2,
            PATH1_3,TAI_POS)[l*90%,u*90%];
      } maintaining PATH1_OK,
      do {Group-Traverse-Path(PATH2_1,PATH2_2,
            PATH2_3,TAI_POS)[l*90%,u*90%];
      } maintaining PATH2_OK
   };
   {Group-Transmit(MCC,ARRIVED_TAI)[0,2],
    do {Group-Wait(TAI_HOLD1,TAI_HOLD2)
            [0,u*10%]
    } watching PROCEED_OK
   }
 }
```

The *choose* expression models the two options for path traversals. 90% of the total time of the overall maneuver is allocated to this group traverse. Each traverse has a maintenance condition that the path

is okay. Arrival is transmitted to the mission control center, 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 [14], 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)*[3]. 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 above. Activity name labels are omitted to keep the figure clear, but the node pairs 4,5 and 6,7 represent the two Group-Traverse-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 traversing to the search area. The TPN represents the consequences of the constraint that the mission last between 25 and 30 hours. 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 traverses along path one (pair 4,5) to the target area in 420 time units (minutes in this case), transmits an arrival message to the mission control center (11,12) for one minute, and concurrently waits (9,10) for another 40 minutes to receive authorization to proceed. Another possible execution is that the group selects the second path, traverses to the target area in 500 minutes, takes 2 minutes 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 traverse along it. The planning algorithm presented in the next section performs the identification of consistent activity executions.

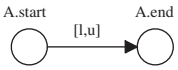
A Temporal Planning Network is a Simple Temporal Network, augmented with *symbolic constraints* and *decision nodes*. These additions are sufficent to capture all RMPL constructs given earlier. Like a simple temporal network, the nodes of a TPN represent temporal events, and the arcs represent temporal 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 traversing 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. All other nodes in the Enroute TPN are non-decision nodes.

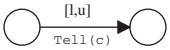
5 Compiling RMPL to TPN

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:

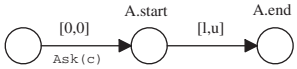
$A[l,u]$. Invoke activity A between l and u time units.



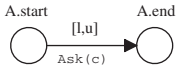
$c[l,u]$. Assert that condition c is true now until $[l,u]$.



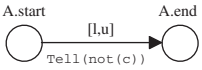
if c **thennext** $A[l,u]$. Execute A for $[l,u]$, if condition c is currently satisfied.



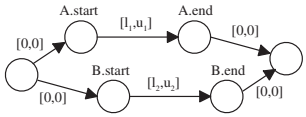
do $A[l,u]$ **maintaining** c. Execute A for $[l,u]$, and ensure throughout A that c occurs.



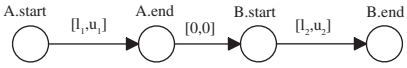
do $A[l,u]$ **watching** c. This program executes A for $[l,u]$, and preempts the execution as soon as condition c becomes true.



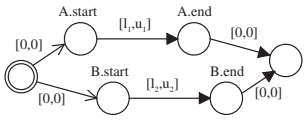
$A[l_1,u_1], B[l_2,u_2]$. Concurrently execute A for $[l_1,u_1]$ and B for $[l_2,u_2]$.



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



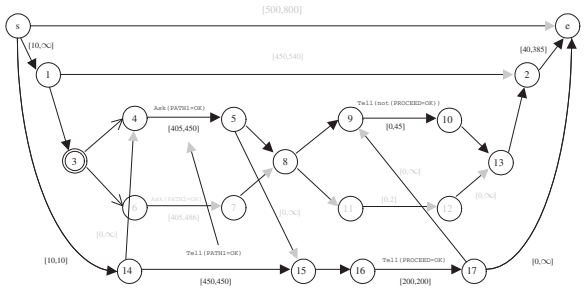
choose $\{A[l_1,u_1], B[l_2,u_2]\}$. Reduces to $A[l_1,u_1]$ or $B[l_2,u_2]$, non-deterministically.



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 [12].

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 searches for a sub-network that constitutes 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[13].

6.1 Phase 1: Select Plan Execution

The first phase selects a set of paths that go from the start-node to the end-node of the top-level activity, and that are temporally consistent. The planner handles this execution selection problem as a variant of a network search[1] rooted at the start-node of the TPN encoding of the top-level activity.

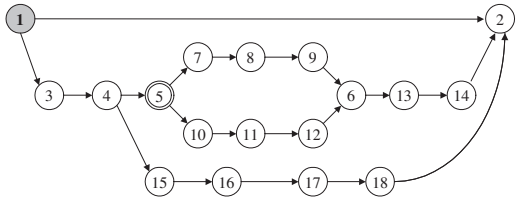
The network search begins at the start-node, and extends a branch of the path at each iteration. Decision nodes are treated by extending the path along

one out arc, while non-decision nodes are treated by branching the path and extending along all out arcs. As the paths are extended they are incrementally tested for temporal consistency as described in [6]. The search completes when the selected nodes and arcs define a set of paths from the start-node to the end-node of the top activity.

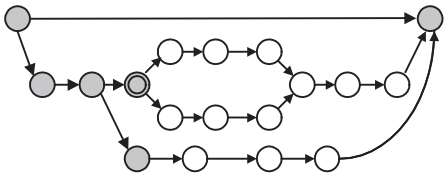
To check temporal consistency we note that any subnet of a Plan Network, minus its symbolic constraint labels, forms a Simple Temporal Network (STN)[3]. An STN is consistent if and only if its encoding as a distance graph contains no negative cycles [3]. The Kirk planner detects negative cycles using a variant of the generic label-correcting single-source shortest-path algorithm [1].

6.2 Example: Searching the Network

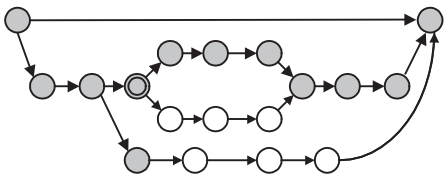
To illustrate this search, we return to an Enroute like input network, where node 1 is the start-node and node 2 is the end-node:



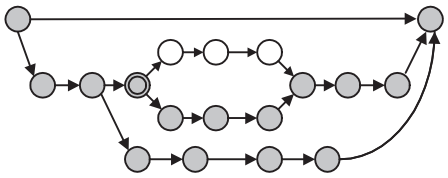
Initially, node 1 is selected, which is indicated by its darker shade. In the first iteration, Kirk chooses to expand node 1, and since node 1 is not a decision node, it selects all out-arcs. This continues until both node 5 and node 15 are selected:



At this point, Kirk chooses node 5 to expand. Since node 5 is a decision node, Kirk 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. In this example, this selected sub-network is temporally inconsistent, so Kirk 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:



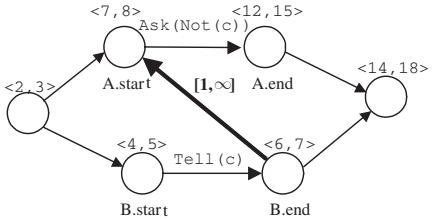
6.3 Phase 2: Threats and Open Conditions

Symbolic constraints – Ask(c) and Tell(c) – are handled analogous to threats and open conditions in causal link planning[13]. 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 Ask.

6.3.1 Resolving Threats

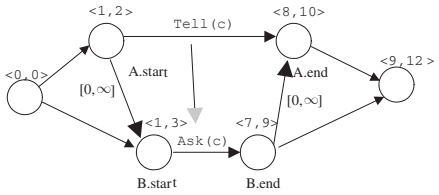
To detect threats the planner computes the feasible time bounds for each temporal event (node) in the network, and then uses these bounds to identify potentially overlapping intervals that are labeled with inconsistent constraints. These bounds are computed by solving an all-pairs shortest-path problem over the distance graph of the partially completed plan [2].

Once these feasible time ranges are determined, the planner detects which arcs may overlap in time. A threat consists of two arcs that may overlap and that are labeled with conflicting symbolic constraints. To resolve a threat we introduce a constraint that forces an ordering between the two activites, similar to promotion and demotion in classical planning[13]:



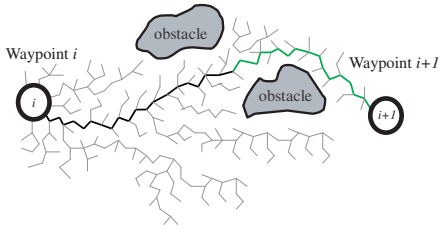
6.3.2 Closing Open Conditions

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 related to the way that the HSTS planner satisfies compatibilities [10].



7 Incorporating Path Planning

A TPN incorporates a range of activities; some of which require scout rovers to drive between locations. For example, nodes 4 and 5, and nodes 6 and 7 from the Enroute activity call for the rovers to traverse either path 1 or path 2 along a sequence of waypoints, while avoiding obstacles. In order to accomplish such a task we extend Kirk with the ability to plan a path from one way point to the next. This involves unifying Kirk's TPN representation and search algorithms with a randomized path planning algorithm based on Rapidly-Exploring Random Trees (RRTs) [8].



An RRT is built by randomly generating a point and connecting the nearest vertex in the existing RRT to that point as shown in the figure below. A new vertex is created and added to the RRT by reaching out a distance epsilon towards the random point. The RRT approach refines ideas from both probabilistic roadmap methods and potential fields methods. Two key features of RRTs is that they are probabilistically complete, and are are heavily biased towards exploring unexplored regions within the path planning space.

The path planning problem is defined in terms of its configuration space where there are obstacle regions and free space regions. The idea behind the algorithm is to create a rapidly exploring random tree, T, that starts at an initial configuration and grows into the goal configuration. This is done by randomly generating a point in the configuration space and connecting it with an edge to the closest vertex within T. This process is repeated until a solution is found.

We are adapting an RRT based path planning algorithm to fit within the TPN framework. Since the underlying data structure for both the path planner and the TPN is based on a graph representation the two can be merged. The sub activities that represent going between waypoints call the path planner for each waypoint. The TPN is updated by including the nodes (which are RRT vertices) generated by the path planner, between each pair of waypoints. This is depicted in the figure shown above. The figure shows what occurs if path 1 of the Enroute activity is selected. A waypoint i between nodes 4 and 5 grows an RRT into the next waypoint $i+1$. A path is found following the trajectory along the black line and the nodes along that path are added into the TPN.

Initially the path planner would receive data from a satellite that surveys the region terrain yielding a coarse resolution map of the area landscape (60 cm/pixel). The problem is then converted into a configuration space problem, where the rover is specified as a point at some start location and the resolved obstacles are identified as the obstacle regions, while the other space is considered free space. This, in turn, is the input into the path planner.

In addition, we are extending Kirk's onboard plan execution algorithms to perform local path planning using on board instruments, such as a laser range finder, sonar, etc. The rovers use these instruments to perform reactive obstacle avoidance while driving between waypoints along the computed path.

8 Testbed

The MIT Space Systems Lab has a testbed consisting of 4 RWI rovers (1 ATRV and 3 ATRV Jr.) and blimps for indoor and outdoor multi-vehicle experi-

ments. An indoor test area is used to study issues on target area exploration, for example, site map refinement with two Scout rovers (two ATRV Jr.) and cooperative exploration by the Scout rovers and the laboratory rovers (ATRV and one ATRV Jr.). The ATRV Jr. rovers have already beed used as an outdoor formation to study relative navigation, communication, control and autonomy issues associated with multi-vehicle fleets. The rovers will be combined with blimps to build a heterogeneous team for surface exploration. The blimps can be used tethered or free flying to provide additional ground coverage.

The Kirk compiler, written in Lisp, converts the RMPL program to TPN specification files (off-line). Kirk's planner, written in C++, generates a plan from the TPN and checks consistency. Kirk's executive takes the resulting partially ordered temporal plan and executes it on the multi-vehicle testbed. The rovers are equipped with sufficient on-board computing resources for vehicle control, sensor data processing, on-line execution of the Kirk planner, of Kirk's executive and the RTT path planning.

9 Discussion

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.

TPNs reduces online planning to graph search. Our current implementation, which uses chronological search, performs well with no search guidance up to TPNs of the size of the enroute example (about 100 nodes in the expanded TPN after planning). We are therefore exploring a reimplementation that applies a more sophisticated search strategy.

An important element of practical temporal planners in the space domain, such as HSTS[9] and IxTeT[7], is the ability to plan with depletable resources. Can RMPL and TPNs be similarly extended to support decision theoretic planning and agile maneuver planning, common to robotic vehicles?

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, Lustre and Signal 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

This research was supported in part by the ONR under contract N00014-99-1-1080, by the DARPA MOBIES program under contract F33615-00-C-1702, and by NASA's Cross Enterprise Technology Development program under contract NAG2-1466.

References

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

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

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

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

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

[6] P. Kim, B. C. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of IJCAI-01*, 2001.

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

[8] S. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Dept. of Computer Science, Iowa State University, 1998.

[9] N. Muscettola. HSTS: Integrating planning and scheduling. In *Intelligent Scheduling*. Morgan Kaufmann, 1994.

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

[11] N. Muscettola, P. Nayak, B. Pell, and B. C. Williams. The new millennium remote agent. *Artificial Intelligence*, 103(1-2):5–48, 1998.

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

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

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

[15] 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.