PROGRESS TOWARDS TASK-LEVEL COLLABORATION BETWEEN ASTRONAUTS AND THEIR ROBOTIC ASSISTANTS

Robert Effinger[(1)], Andreas Hofmann[(1)], Brian Williams[(1)]

[(1)]MIT Computer Science and Artificial Intelligence Lab, MIT
32 Vassar St., rm. 32 - 275
Cambridge, MA 02139 (USA)
effinger@mit.edu, hofma@csail.mit.edu, williams@mit.edu

ABSTRACT

In the future, NASA envisions robotic assistants seamlessly interacting with astronauts. These robots must be capable of understanding abstract tasks, and must also reliably execute the tasks. We make progress towards these goals by firstly developing a task-level programming language, called RMPL, that robots can directly interpret and understand. Secondly, we develop a hybrid executive that can execute the tasks reliably, even while adapting to disturbances and execution uncertainties.

1. INTRODUCTION

Current research in space robotics has shown that many of the tasks performed by astronauts could also be performed by robotic assistants. In the lab, these robots have demonstrated the capability to perform space-truss assembly, EVA setup and teardown, Shuttle tile inspection, and ISS maintenance and repair operations [1][4]. Robotic assistants have the potential to decrease astronaut workload and increase human safety by performing time-consuming and dangerous on-orbit tasks in place of the astronaut. To this purpose, they must be capable of seamlessly interacting with the astronaut team. This means that they must be able to 1.) interpret task-level commands issued by astronauts, and 2.) execute the tasks in a safe and reliable manner, even under disturbances and execution uncertainties.

We make progress along both fronts by presenting current research on an activity planner, named Kirk[3wil.et.al], which 1.) accepts as inputs task-level commands written in a task-level programming language called RMPL, and 2.) converts the task-level commands into a lower-level temporally flexible plan that autonomous robots are able to execute robustly while adapting to disturbances and uncertainties.

First, we introduce RMPL, a task-level programming language, and describe how task-level commands in RMPL are converted into a temporally flexible plan representation called a Temporal Plan Network (TPN). Next, we introduce an incremental algorithm that enables replanning at any point when the current TPN fails. Finally, we introduce a hybrid executive that exploits the temporal flexibility in the TPN to safely and reliably adapt to disturbances. We demonstrate the hybrid executive on a particularly challenging example of a robotic assistant, a bipedal walking machine.

2. A MOTIVATING EXAMPLE

To motivate the utility of robotic assistants in space, we re-examine the original Apollo Lunar Roving Vehicle (LRV) deployment sequence, but with a twist! The tasks originally performed by one of the astronauts will now be performed by a humanoid robotic assistant. In this ambitious example, we assume only one astronaut has landed on the Moon, but with a legged and capable humanoid robot as a sidekick. Fig. 1 shows an illustration of two Apollo astronauts deploying the LRV. The original deployment sequence is presented in Fig 2. First, one astronaut removes the heating blanket and operating tapes. Next, the two astronauts simultaneously lower the LRV and deploy the front and aft wheels by pulling on the deployment cables. Finally, one of the astronauts deploys the seats and footrests. In the following sections, we show how this task-level plan can be expressed in RMPL, converted into a TPN, and then executed by a hybrid executive even under disturbances and execution uncertainties.

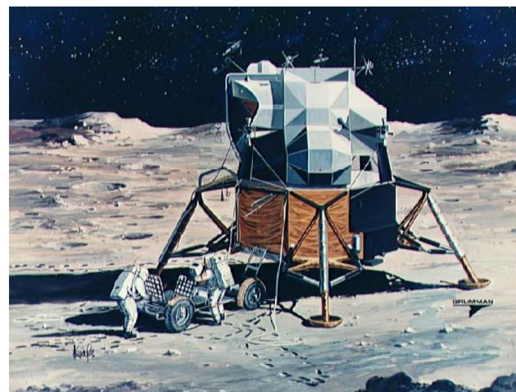


Fig. 1. LRV Deployment. (*courtesy of NASA)

Proc. 'ISAIRAS 2005 Conference', Munich, Germany,
5-8 September 2005 (ESA SP-603, September 2005)

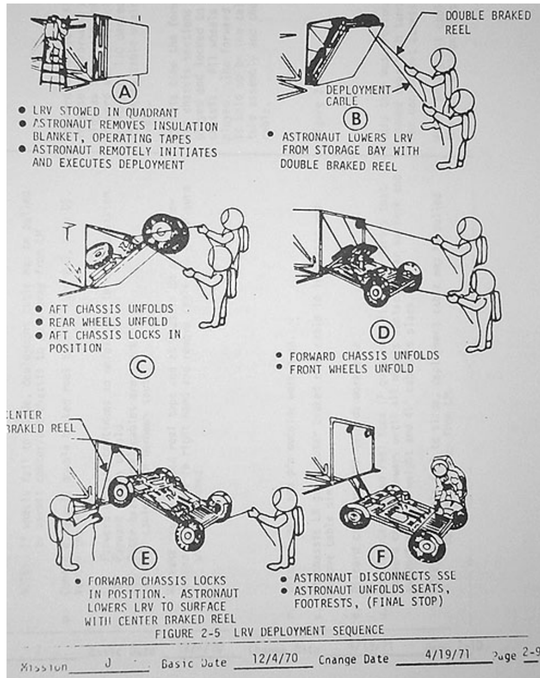
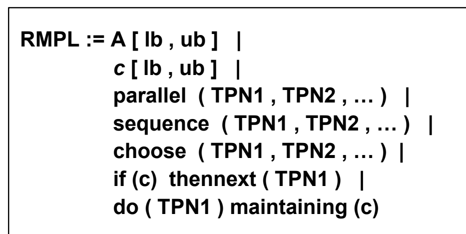


Fig. 2. Apollo 15 LRV Deployment Sequence.
(*courtesy of NASA)

3. RMPL: A TASK-LEVEL PROGRAMMING LANGUAGE

A robotic assistant must be capable of interpreting task-level commands issued by the astronaut. We progress towards this goal by developing the Reactive Model-based Programming Language (RMPL). RMPL allows humans to intuitively construct networks of tasks in a language that is interpretable by a robot. RMPL supports concurrency, flexible execution times, maintaining conditions, synchronization, metric constraints and contingencies. These constructs are shown in Fig. 3, and are described briefly in the following paragraph. A more thorough description and analysis of RMPL can be found in [11].



RMPL := A [lb , ub] |
***c* [lb , ub] |**
parallel (TPN1 , TPN2 , …) |
sequence (TPN1 , TPN2 , …) |
choose (TPN1 , TPN2 , …) |
if (c) thennext (TPN1) |
do (TPN1) maintaining (c)

Fig. 3. A Formal Definition of RMPL.

Lower case letters and words, like ***c***, denote primitive activities and conditions, and upper case letters and words like **A** and **TPN1** denote well-formed RMPL expressions. **A [lb, ub]** is the basic construct for expressing timing requirements, and implies that **A** *must not* finish executing before **lb** time units, and *must* finish executing before **ub** time units. The **parallel** and **sequence** constructs are for concurrent and sequential tasks, and **choose** is used to express multiple strategies and contingencies. Together, ***c*** and **if (c) thennext (TPN1)** provide the constructs for synchronization, and **do (TPN1) maintaining (c)** acts as a maintenance condition.

Next, we encode the Apollo LRV deployment sequence in RMPL in Fig. 4. Each task beginning with *R:* is to be performed by the humanoid robot, and each task beginning with *A:* is to be performed by the astronaut. The **sequence** and **parallel** constructs provide the basic building blocks for piecing together the network of tasks, the **do … maintaining** construct ensures simultaneous lowering and deployment of the LRV, and the **choose** construct allows multiple strategies in the plan, for example, either the astronaut *or* the robot can deploy the seats and footrests.

```
LRV-deployment-sequence() [5,20] = {
 sequence(
   R: Remove insulation blanket [1,3],
   R: Remove operating tapes [1,3]
   parallel(
     A: Lower LRV w/braked reel [1,5],
     do (
       sequence(
         R: deploy aft wheels [0.5,2],
         R: deploy front wheels [0.5,2]
       )
     )maintaining (tension on cable)
   )
   choose(
     A: deploy seats & footrests [1,5],
     R: deploy seats & footrests [5,10]
   )
 )
} [5,20]
```

Fig. 4. LRV deployment sequence in RMPL.

The key observation here is that the LRV deployment sequence is now expressed in a language directly interpretable by the robotic assistant! Next, we show how a robot interprets this task-level plan specification.

4. TEMPORAL PLAN NETWORKS

A robot interprets an RMPL program by compiling it into a temporally flexible plan graph called a temporal plan network (TPN). A TPN is similar to a simple temporal network (STN)[3], that is, it includes activities, predecessor and successor relations between activities, and simple temporal constraints that relate the start and end times of activities. Additionally, a TPN represents options or contingencies in a plan by augmenting the STN with choice nodes. Fig. 5

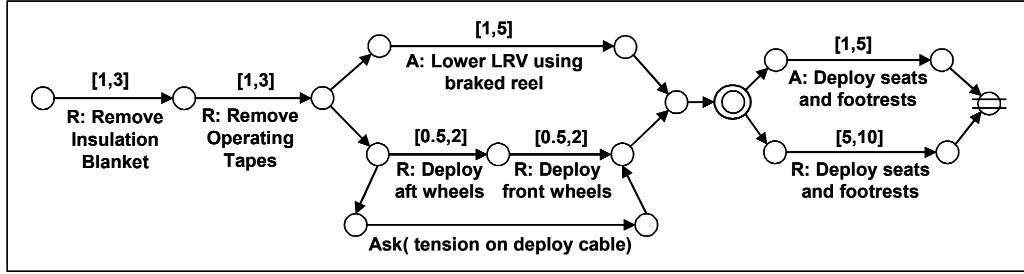


Fig. 6. Temporal Plan Network of the Apollo 15 LRV deployment sequence.

presents a concise definition of the TPN; the complete development of a TPN, which includes mutex support is available in [11]. The RMPL program for the LRV deployment sequence is converted into a TPN in Fig. 6.

RMPL	TPN
A [lb , ub]	A [lb , ub]
c [lb , ub]	[lb , ub] Tell (c)
sequence (TPN1 , TPN2 , …)	TPN1 TPN2 …
parallel (TPN1 , TPN2 , …)	TPN1 TPN2 …
choose (TPN1 , TPN2 , …)	TPN1 TPN2 …
if (c) thennext (TPN1)	Ask (c) TPN1
do (TPN1) maintaining (c)	TPN1 Ask (c)

Fig. 5. Conversion from RMPL to TPN.

5. TASK DECOMPOSITION

If a task in an RMPL program is abstract, it must be decomposed into lower-level tasks that are executable by the robot. For example, the *R: Deploy seats and footrests* task in Fig. 6 needs to be decomposed into to be decomposed into several simpler, sequential tasks. Since RMPL is a recursively defined language, the abstract task is simply expanded into its component sub-tasks using recursive RMPL constructs as shown in Fig. 8.

So far, we have shown how a robot interprets an RMPL plan specification by converting it into a temporally flexible plan (a TPN) and also how abstract tasks in the TPN can be decomposed into a lower-level network of tasks that the robot knows how to execute. Next we show how the robot determines if the decomposed network of tasks is temporally consistent.

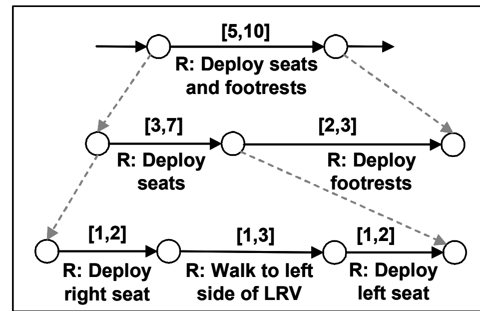


Fig. 8. An Example of Task Decomposition.

6. TEMPORAL CONSISTENCY OF A TPN

A TPN with each choice assigned is exactly an STN, thus, we can determine temporal consistency in polynomial time just as with STNs [3]. It is also proven in [3] that an inconsistent distance graph causes the shortest path calculation to loop continuously, creating a negative cycle. This cycle can be detected quickly by looking for self-loops in the set of support [2]. Next, we introduce an algorithm that determines temporal consistency of distance graphs incrementally.

7. INCREMENTAL CONSISTENCY OF A TPN

An important enabler for fast replanning is the ability to quickly locate and repair inconsistencies in the plan. We enable fast replanning by introducing an incremental temporal consistency algorithm(ITC). ITC supports fast consistency testing through an incremental set of support [2], and through a set of incremental update rules. ITC's update rules can both update a consistent distance graph *and* repair an inconsistent distance graph. A simple example of each is given below, and the full pseudocode for ITC is developed in [9].

7.1 Incrementally Update a Consistent Graph

ITC has three incremental update rules to update a consistent distance graph. The three rules are divided based on how a change affects the current shortest-path: (1) no effect to the current shortest-path, (2) improves the shortest-path, and (3) invalidates the current shortest-path.

(1) Arc Change without Effect to Shortest Path
An arc can change in such a way that the shortest-path to a node is unaffected. The graph in this case requires no updates, because the shortest-path distances do not change. For example, in Fig. 9, the current best way to reach node j is to go through node g, as specified by the predecessor pointer (p=g) of node j. The arc change has no effect on the shortest-path, and no further updates need to be performed.

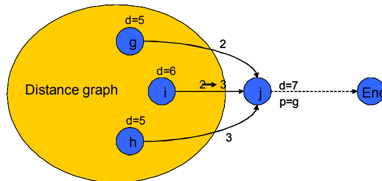


Fig. 9. No Effect to Shortest-Path.

(2) Arc Change Improves Shortest-Path

An arc distance decrease can improve the shortest-path to one or more nodes. In this case, the shortest-path distance value of the node at the head of the changed arc needs to be updated, and this updated distance value should be propagated to successor nodes. For example, in Fig. 10, arcij reduces in cost from 3 to 0. With this change, the shortest-path distance to node j can be decreased from 7 to 6, through node i. Since the successor nodes of node j could be affected, the outgoing arcs from node j must be examined.

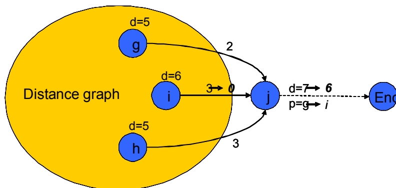


Fig. 10. Shortest-Path Improvement.

(3) Arc Change Invalidates Shortest-Path
In the third case, an increase in arc distance can invalidate the current shortest-path to a node. In this case, all nodes depending on the invalidated node for support must also be invalidated. Then, new shortest path values must be propagated by re-examining the invalidated node's parents. For example, in Fig. 11, an increase to arcij invalidates node j's shortest-path value. First, all nodes that use node j in their shortest path must be invalidated. Then, node j's parents are added to the Q so new shortest-path values and predecessor pointers will be propagated to all of the invalidated nodes.

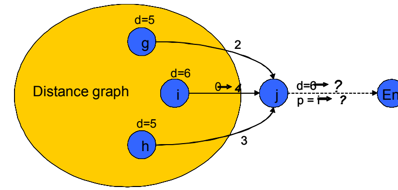


Fig. 11. Shortest-Path Improvement.

This completes a brief overview of ITC's three update rules to update a consistent distance graph. Next we describe how ITC reasons incrementally to repair an inconsistent distance graph.

7.2 Incrementally Repair an Inconsistency

When ITC discovers a negative cycle, three steps must be performed to repair the distance graph and maintain a correct set of support: 1) reset every node in the negative cycle by setting d(n) to ∞, and the predecessor pointer to unknown, 2) reset all nodes that depend on the negative cycle by recursively invalidating supports,

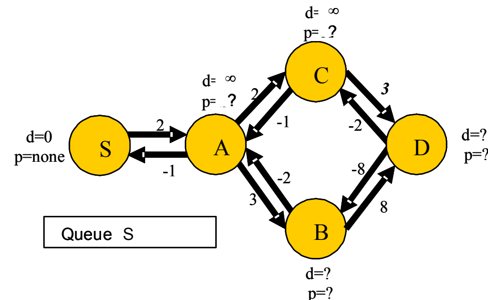


Fig. 12. Shortest-Path Improvement.

and 3) for every node that was reset in steps 1 or 2, insert onto the Q any parent of that node that has not also been invalidated. For example, Figure 12, shows a distance graph just after ITC has detected the negative cycle, ABDCA, and then performed all three incremental repair steps.

3.2 Experimental Results of ITC

The Kirk planner was tested on a set of realistic aerial vehicle missions, and also a set of randomly generated plans, shown in Fig. 14. In both test cases, planning with ITC improves Kirk's planning speed by an order of magnitude. A detailed analysis of these experimental results is available in [9].

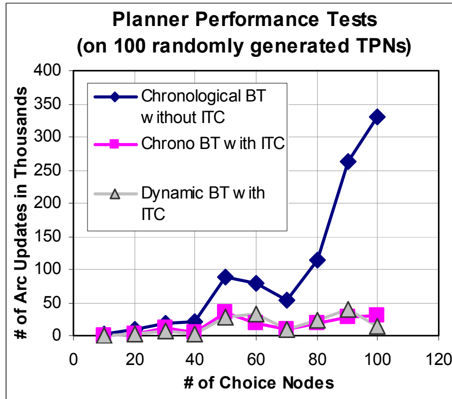


Fig. 13. Randomly Generated Plan Results.

8. MOTION EXECUTION

Movement of the robot is most naturally specified as a sequence of qualitative behaviors culminating in the task goal. Consider, for example, the walking task subgoal in Fig. 8. A walking gait cycle can be described as a sequence alternating between single support and double support. We call each step in such a sequence a *control epoch*, and the overall sequence, a *qualitative state plan,* QSP. Qualitative behavior within an epoch is specified with a set of state-space region and temporal range constraints. Thus, a qualitative state plan is an extension of the previously described temporal plan; besides specifying temporal constraints, a QSP specifies spatial requirements using state-space region constraints.

Translating such a qualitative description into control actions is a challenging problem because the robot is highly nonlinear, has high dimensionality and has input constraints that limit controllability. Systems of this type include continuous, as well as discrete state variables, so we refer to such systems as *hybrid*. Whereas an executive for a discrete state system ignores the detailed dynamics of the system being controlled, an executive for a hybrid system must take these into account, along with associated controllability limits.

8.1 Approach

As with traditional temporal plans used for discrete state systems [8], a qualitative state plan is composed of activities related by simple temporal constraints. However, a qualitative state plan differs in that an activity specifies the legal evolution of a state variable, rather than an executable activity. As shown in Fig. 14a, foot placement constraints define qualitative poses such as double support, or left single support, but the details of the joint positions are omitted. A goal region for the forward position at the end of the gait sequence defines the goal. A temporal range constraint specifies task completion time requirements.

Such a qualitative behavior description is much more convenient than an activity plan, but translation into control actions is difficult. We solve this problem by synthesizing a set of adaptive controllers, customized for the state plan. Performing such a synthesis on-line is not feasible because this process is computationally intensive. On the other hand, fixing all control parameters offline limits flexibility. Therefore, we use a mixed on-line/off-line approach, where an off-line plan compiler computes a set of bounds on control parameters, and an on-line hybrid dispatcher efficiently adapts control settings, within these bounds, to respond to disturbances. The plan compiler and hybrid dispatcher comprise a model-based executive [12], as shown in Fig. 14b.

To simplify controller synthesis, we utilize a feedback linearizing multivariable controller, which transforms the highly nonlinear, tightly coupled biped system into a loosely coupled set of linear 2nd-order single-input single-output (SISO) systems [6]. This *SISO abstraction* greatly simplifies the plan compiler's job, because the control parameters computed are for a small set of linear control laws, rather than for a complex nonlinear system.

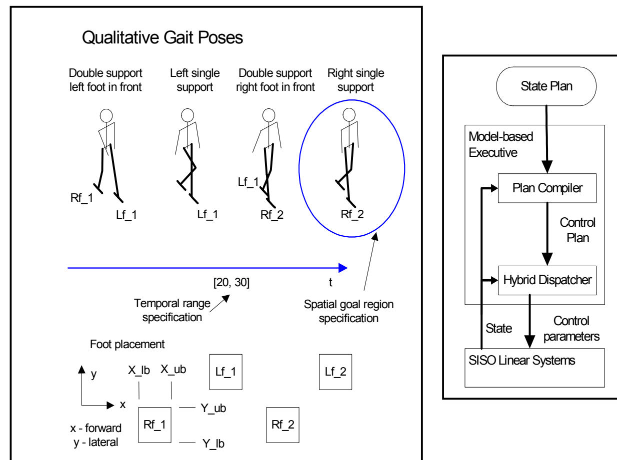


Fig. 14 – a. Qualitative task specification (left), b. Model-based executive (right)

8.2 Qualitative State Plan

A qualitative state plan is used to represent the desired state evolution of the plant. Given a state plan and a plant, the execution task is to generate a sequence of control actions that moves the plant to a state consistent with that required by the plan.

A qualitative state plan specifies state evolution using sequences of *activities* as shown in Fig. 15. Each activity is part of a control epoch (column in Fig. 15). The control epochs in Fig. 15 correspond to the qualitative poses shown in Fig. 14. Vertical bars in Fig. 15, between rows, represent synchronization constraints.

Each activity may have a temporal duration range constraint, indicated by [lb, ub] (lb indicates lower bound, ub, upper bound). In addition, range constraints on initial and goal regions for quantities may be specified using rectangles in position/velocity phase space. Note that the duration and region constraints are optional, and these constraints are omitted for many activities. In Fig. 15, we care about the initial and final region of the CM, but not about the details in between. Similarly, we care that the gait cycle be completed within time range [t_lb, t_ub], but not about the specific duration of each activity.

Formally, we define a state plan as a set, A, of activities, $a(i,j)$, where i refers to the quantity, and j to the control epoch. An activity is defined by the tuple $\langle R_{init}, R_{tube}, R_{goal}, R_{temporal}, a_{next} \rangle$ where R_{init}, R_{tube}, and R_{goal} specify, respectively, initial, operating, and goal regions in state space for the controlled variable associated with the activity, and $R_{temporal}$ specifies temporal constraints. The activity a_{next} is the activity to transition to when the current activity is finished.

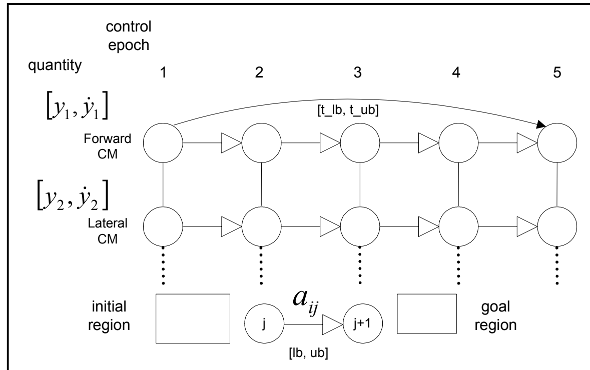


Fig. 15 – Qualitative State Plan

The region constraints, R_{init}, R_{tube}, and R_{goal} are of the form $(y_{i\min} \le y_i \le y_{i\max}) \wedge (\dot{y}_{i\min} \le \dot{y}_i \le \dot{y}_{i\max})$ for controlled quantities, and $(y_{i\min} \le y_i \le y_{i\max})$ for input quantities. An activity may begin if its quantity is in the region defined by R_{init}. An activity cannot end unless the quantity is in R_{goal}. The quantity must stay within R_{tube} for the entire duration of the activity.

$R_{temporal}$ is of the form $\langle R_{duration}, A_{parallel} \rangle$, where $R_{duration}$ is a simple temporal constraint $[lb, ub]$, and $A_{parallel}$ is a set of activities that must finish simultaneously with the current one (vertical bars in Fig. 15). For example, for a biped, movement of the stepping foot must be synchronized with forward movement of the center of mass.

A valid plan execution is defined as follows. An activity finishes if its R_{goal}, $R_{duration}$, and $A_{parallel}$ constraints are satisfied. After it finishes, it transitions to the successor, a_{next}, immediately. An activity, a, is executed successfully iff there exists a start time, ts, and a finish time, tf, for the activity, such that $lb \le tf - ts \le ub$, and there exists a trajectory for the associated controlled variable y such that $y(ts)$ and $\dot{y}(ts)$ satisfy R_{init}, $y(tf)$ and $\dot{y}(tf)$ satisfy R_{goal}, and $y(t)$ and $\dot{y}(t)$ satisfy R_{tube} for $ts \le t \le tf$. A state plan is executed successfully iff each activity, $a(i,j)$, is executed successfully, and the associated finish time, $tf(i,j)$, is such that if the activity has a successor, $a(i,j+1)$, then $tf(i,j) = ts(i,j+1)$, and for any parallel activity, $a(k,j)$, listed in $A_{parallel}$, $tf(i,j) = tf(k,j)$.

8.3 Plan Compiler

Restrictions on control parameters are determined at compile time and are captured in a control plan. The precise control parameter setting for each activity is then determined by the dispatcher at run-time. A control plan consists of the activities from the state plan, with two additions: 1) SISO control parameter information is included, and 2) the region constraints, R_{init}, R_{tube}, and R_{goal}, and the duration constraint, $R_{duration}$, specify non-infinite bounds. Control parameter constraints are of the form $\langle k_{1\min}, k_{1\max}, k_{2\min}, k_{2\max} \rangle$, and specify bounds on control parameters for the linear control laws used in the SISO abstraction.

In computing bounds on the control parameters, the compiler performs an adaptive controller synthesis. In order to maximize robustness to disturbances, the compiler attempts to maximize the size of initial regions, and tubes, and maximize controllable temporal activity duration ranges. Maximizing controllable temporal range makes synchronization with other controlled quantities easier. To perform this optimization, we utilize an SQP (Sequential Quadratic Programming) optimizer, and formulate the problem as an NLP (Nonlinear Program).

The NLP formulation for the control plan computation is as follows. For each activity, a_{ij}, parameters to optimize are: $\langle y_{init\min}, \dot{y}_{init\min}, y_{init\max}, \dot{y}_{init\max} \rangle$ (parameters of R_{init}), $\langle y_{goal\min}, \dot{y}_{goal\min}, y_{goal\max}, \dot{y}_{goal\max} \rangle$ (parameters of R_{goal}), $\langle t_{\min}, t_{\max} \rangle$ (parameters of $R_{duration}$), and $\langle k_{1\min}, k_{1\max}, k_{2\min}, k_{2\max} \rangle$, the bounds on the control parameters. In order to understand how this computation works, it is necessary to understand two trajectories that represent extremes of behavior: the *guaranteed fastest trajectory* (GFT), and the *guaranteed slowest trajectory* (GST). The GFT represents a lower bound on the time needed to get from anywhere in the initial region, to somewhere

in the goal region. The GST represents the corresponding upper bound.

Consider the initial and goal regions shown in Fig. 16a. For the GFT, the worst-case starting point in the initial region is point B, which corresponds to $y_{init\min}, \dot{y}_{init\min}$. By accelerating as quickly as possible, the GFT reaches point D. This represents the fastest finish point in the goal region. For the GST, the worst-case starting point is point A. This represents the fastest possible start. By accelerating as slowly as possible, the GST reaches p;oint C, the slowest finish point.

The times for each trajectory are designated t_{GFT} and t_{GST}. If $t_{GFT} < t_{GST}$, then there exists a temporal range, $[t_{GFT}, t_{GST}]$, during which the endpoint of a trajectory beginning from anywhere in the initial region can be guaranteed to be in the goal region. The existence of this temporal range is important for synchronizing with other controlled quantities. Thus, the GFT and GST are useful for determining a maximum initial region, given a particular goal region, such that the controllable temporal range exists.

GFT and GST can be understood intuitively by considering a single acceleration spike control. If this spike is applied at the beginning of the trajectory, then maximum velocity is reached immediately, resulting in the GFT, as shown in Fig. 16b. If it is applied at the end, then the trajectory will progress at minimum velocity resulting in the GST.

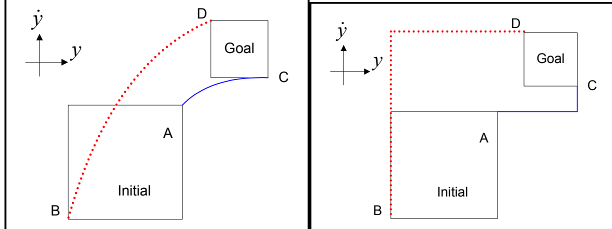


Fig. 16 – a. GFT (dotted) and GST (solid), left, b. for acceleration spike control laws, right

In the NLP formulation, existence of initial and goal regions is expressed as

$$y_{init\min} < y_{init\max},\ \dot{y}_{init\min} < \dot{y}_{init\max} \qquad (5)$$
$$y_{goal\min} < y_{goal\max},\ \dot{y}_{goal\min} < \dot{y}_{goal\max}$$

To guarantee contraction, the goal region of an activity must fit inside the initial region of its successor.

$$y_{goal\min}(i,j) \ge y_{init\min}(i,j+1) \qquad (6)$$
$$\dot{y}_{goal\min}(i,j) \ge \dot{y}_{init\min}(i,j+1)$$
$$y_{goal\max}(i,j) \le y_{init\max}(i,j+1)$$
$$\dot{y}_{goal\max}(i,j) \le \dot{y}_{init\max}(i,j+1)$$

Constraints representing the GFT are expressed as

$$y_{goal\min} = f_1(y_{init\min}, \dot{y}_{init\min}, k_{1\max}, k_{2\max}, t_{\min}) \qquad (7)$$
$$\dot{y}_{goal\max} = f_2(y_{init\min}, \dot{y}_{init\min}, k_{1\max}, k_{2\max}, t_{\min})$$

Constraints representing the GST are expressed as

$$y_{goal\max} = f_1(y_{init\max}, \dot{y}_{init\max}, k_{1\min}, k_{2\min}, t_{\max}) \qquad (8)$$
$$\dot{y}_{goal\max} = f_2(y_{init\max}, \dot{y}_{init\max}, k_{1\min}, k_{2\min}, t_{\max})$$

The requirement for temporal controllability is expressed as $t_{\min} < t_{\max}$. Synchronization constraints are

$$(t_{\min}(1,j) \le t_{trans\min}(j) \le t_{trans\max}(j) \le t_{\max}(1,j)) \wedge \ldots$$
$$(t_{\min}(i,j) \le t_{trans\min}(j) \le t_{trans\max}(j) \le t_{\max}(i,j))$$

Thus, $[t_{trans\min}(j), t_{trans\max}(j)]$ is the temporal range when transition out of control epoch j may occur.

The cost function maximizes initial region size and controllable temporal range.

8.4 Hybrid Dispatcher

The hybrid dispatcher executes the qualitative control plan. It acts as an adaptive controller, adjusting control parameters, to guide each controlled quantity into its goal region at the correct time. When all quantities are in their respective goal regions, the hybrid dispatcher transitions to the next control epoch.

At the beginning of a new control epoch, j, the dispatcher computes a target transition time, $t_{trans}(j)$. For each controlled quantity, the dispatcher then monitors progress by computing a prediction of the point in state space for the controlled quantity at $t_{trans}(j)$. This prediction is computed analytically in the same manner as eq. 1, so it is fast. If the predicted point is within the goal region, then the dispatcher does nothing. If it is outside the goal region, then the dispatcher adjusts the control parameters to attempt to move the predicted point back into the goal region. If this is unsuccessful, the plan execution fails, and the dispatcher requests a new plan.

8.5 Movement Execution Results and Discussion

Fig. 18 shows control plan regions computed by the second step of the plan compiler. Fig. 18a shows lateral CM (velocity vs. position), and Fig. 18b shows forward CM. As can be seen in Fig. 18b, good contraction is achieved for forward CM; the goal regions fit well inside the initial regions for the subsequent mode. Contraction is not as good for lateral CM; the goal regions barely fit inside the initial regions (Fig. 18a). This is due to the fact that controllability in the lateral direction is more limited, because the support base is narrower.

The motion sequence in Fig. 19 shows a nominal execution of this control plan, which results in dynamic walking at medium speed. The motion sequence in Fig. 20 shows a similar walking pattern, but with irregular foot placements due to the need to step on the stones. The motion sequence in Fig. 21 shows recovery from a lateral push disturbance.

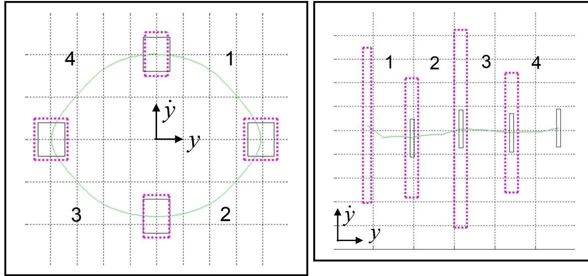


Fig. 17 – Initial (dotted) and goal (solid) regions for control epochs 1 – 4, a. lateral CM (left), b. forward CM (right), plots show region in velocity-position phase space.

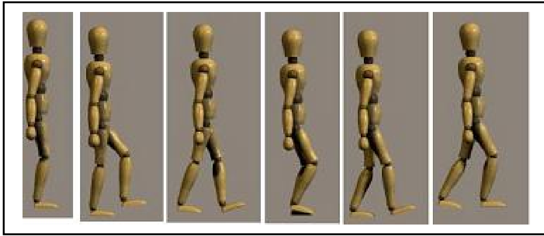


Fig. 18 Motion sequence for dynamic walking

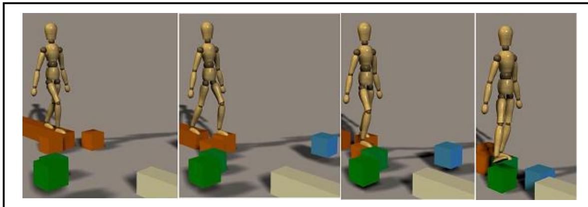


Fig. 19 – Motion sequence for plan requiring careful foot placement

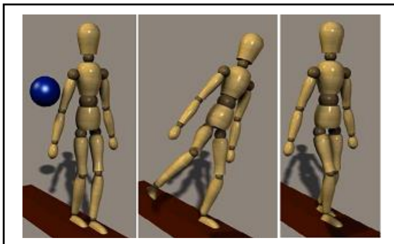


Fig. 20 – Motion sequence for recovery from lateral push disturbance

9. CONCLUSIONS AND FUTURE WORK

In this paper, we show how an abstract task-level plan, written in RMPL, can be interpreted and decomposed by a robot into a detailed temporally flexible plan, a TPN. We then show how that TPN can be updated incrementally, and then executed reliably by a hybrid executive. We demonstrate this approach on a particularly challenging example of a robotic assistant, a bipedal walking machine. In the future, we plan to demonstrate this system on two cooperating 4DOF robotic manipulators, and also the Robonaut Simulator, a simulation of a humanoid robot.

10. ACKNOWLEDGEMENTS

This research was supported in part by the DARPA MICA program under contract N66001-01-C-8075, and the NASA IS program under contract NCC-2-1235.

11. REFERENCES

1. Ambrose, R., Culbert, C., and Rehnmark, "An experimental investigation of dexterous robots using EVA tools and Interfaces, *AIAA Space 2001*, Albuquerque, NM.

2. Cesta, A. and Oddi, A., "Gaining Efficiency and Flexibility in the Simple Temporal Problem", *3rd Workshop on Temporal Representation and Reasoning*. TIME 1996.

3. Dechter, R.; Meiri, I.; Pearl, J., 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49:61-95.

4. Fredrickson, S.E., Lockhart, P.S., and Wagenknecht, J.D. "Autonomous extravehicular robotic camera (AERCam) for remote viewing. *AIAA International Space Station Service Vehicles Conference* 1999, Houston, TX.

5. A. Goswami. Postural stability of biped robots and the foot rotation indicator (FRI) point. *International Journal of Robotics Research*, July/August 1999

6. A. Hofmann, S. Massaquoi, M. Popovic, and H. Herr. A sliding controller for bipedal balancing using integrated movement of contact and non-contact limbs. *Proc. International Conference on Intelligent Robots and Systems* (IROS). Sendai, Japan

7. P. Morris, N. Muscettola, and T. Vidal. Dynamic control of plans with temporal uncertainty. *Proceedings of the 17th International Joint Conference on A.I.* (IJCAI-01). Seattle (WA, USA).

8. N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. *Proc. Of Sixth Int. Conf. On Principles of Knowledge Representation and Reasoning*, 1998.

9. Shu, I., Effinger, R., Williams, B., "Enabling Fast Flexible Planning through Incremental Temporal Consistency with Conflict Extraction". *ICAPS*, 2005.

10. E. Bradley and F. Zhao. Phase-space control system design. *Control Systems*, 13(2),39-46 1993.

11. Williams, et.al., "Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration." *iSAIRAS,* St-Hubert, Canada, June 2001.

12. B. Williams and P. Nayak. A Reactive Planner for a Model-based Executive. *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI, 1997)