ICAPS05

WS7

Workshop on Plan Execution: A Reality Check

Sailesh Ramakrishnan
QSS Group Inc
NASA Ames Research Center

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

CONFERENCE CO-CHAIRS:

Susanne Biundo
University of Ulm, GERMANY

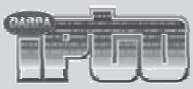
Karen Myers
SRI International, USA

Kanna Rajan
NASA Ames Research Center, USA

Cover design: L.Castillo@decsai.ugr.es

Workshop on Plan Execution: A Reality Check

Sailesh Ramakrishnan
QSS Group Inc
NASA Ames Research Center


Workshop on Plan Execution: A Reality Check

Table of contents

Workshop on Plan Execution: A Reality Check

Preface

Planning for realistic domains presents a varied set of challenges. Key among those challenges is understanding and representing the execution time behavior of the generated plans. Recent experiences in designing and deploying planning systems provide significant insight into the execution of plans generated by automated planners. This experience strongly suggests the presence of a gap between how plan execution is treated in the plan generation process, and what happens when the resulting plan is actually executed.

For automated planning systems to be successful in the real world, it is essential that the nature of this gap be understood and some techniques for bridging it be developed. Some of the relevant issues include the following questions:

1. *Is there a fundamental problem in our understanding of what planning is versus what execution is?*
2. *What is the range of possible semantics for execution? Are current domain modeling languages adequate to the task of representing execution?*
3. *How do planning systems fit architecturally with executives and hardware controllers? What are the strengths and weaknesses of current implementations, and where is more work needed?*
4. *How do planners cope with the mismatch between their representation of the domain and reality? (in terms of time latency, inaccuracies in modeling etc..) Is there a fundamental difference between how this prediction uncertainty is handled in control, and how it should be handled for planning?*
5. *What tools and practices may be adopted to bridge the gap? What can we learn from case studies, deployment experiences and other associated areas (such as hybrid controller design, real-time controls etc).*

The papers accepted to this workshop show a significant interest in the planning community in understanding these issues. The papers describe a range of approaches from novel algorithms to architectures that are robust under execution. The authors represent a coming together of different sub-communities in AI. During this workshop we expect a very educational discussion from different perspectives and hope to engender a cross pollination of ideas, approaches, tools and software.

I would like to thank the members of the Organizing Committee for their efforts in bringing this workshop together. I would also like to thank the conference and workshop chairs for their timely assistance in supporting and publicizing the workshop.

Organizer

- *Sailesh Ramakrishnan (chair), QSS Group Inc, NASA Ames Research Center.*

Programme Committee

- *Michael Beetz, Technical University Munich.*
- *Gautam Biswas, Vanderbilt University.*
- *Mark Boddy, Adventium Labs.*
- *Felix Ingrand, LAAS.*
- *Nicola Muscettola, NASA Ames Research Center.*
- *Issa A. D. Nesnas, JPL.*

Blocks in Space: Intelligent Self-Assembly Using Optimal Control Trajectory Planning

Ella M. Atkins, Gina D. Moylan

University of Maryland, Space Systems Laboratory
382 Technology Drive, College Park, MD, 20740
{ella | gmoylan} @ssl.umd.edu

Abstract

Many different layers of automation must be integrated to support future space missions. At the base layer, spacecraft must autonomously navigate; i.e., follow a specified trajectory using the spacecraft's actuators, sensors and knowledge of its dynamics. This trajectory must minimize precious fuel use and satisfy mission goals and environmental constraints. To facilitate an appropriate connection between task and optimal trajectory planning, we map the well-known Blocks World domain to the space environment by defining a simple task-level implementation that uses cost information from an optimal trajectory planner to make action choices. Our method is applicable at both the micro-level where obstacles must be efficiently circumvented and the macro-level where orbital dynamics dictate assembly task sequencing and trajectory design.

Introduction

Imagining a child stacking blocks on the floor is a pleasant exercise many people can relate to. Placing these same blocks in the space environment and having them self-assemble into particular "stacked" configurations is anything but child's play. When considering the necessary role automation must play in future space missions and endeavors, it is important to study basic scenarios that further our understanding of the challenges we must overcome to meet such objectives. We believe some of these challenges lie in the inherent disconnect between the planning of tasks and the development of the continuous trajectories that must be followed to accomplish these tasks. Before any other mission tasks/goals can be prioritized and fulfilled, spacecraft must be able to autonomously navigate; i.e., follow a specified trajectory using the spacecraft's actuators, sensors and knowledge of its dynamics. To be clear, we make an important distinction between path and trajectory:

- A **path** is the locus of waypoints followed during motion; i.e., a purely geometric motion description.
- A **trajectory** is a path that includes velocities and/or accelerations at each point according to the governing equations of motion; i.e., a geometric and temporal description of the object's motion.

Mission goals are fulfilled by selecting action choices that optimize fuel use and time given system and environmental constraints. While these action choices can be easily implemented with traditional AI planning tools, the optimization of the fuel/time resources required to move through space requires consideration of system dynamics and actuation capabilities involving complex physical motions governed by nonlinear differential equations. Optimal trajectory planners are specifically designed for this task, connecting spatial waypoints with a physically-achievable trajectory. However, they are not designed to optimize over a global assembly problem involving a large number of choices in terms of what gets assembled, when and where (Henshaw, 2003). Full automation—especially in a space environment—must involve the coordination of intelligent task and motion planning.

The problem we address in this paper is set within the context of the famous microworld domain known as "Blocks World" (BW), where an infinite table holds a finite set of unique blocks to be stacked in particular configurations (Slaney and Thiébaux, 2001):

> **Problem Definition:** A group of 4 self-actuating blocks are deployed so that they are in approximately the same orbit but have slightly different positions. From an initial 'snapshot,' the goal is to build a linear 4-block structure in a fuel-optimal 'stacking arrangement' using `block-a` as the anchor block (see Figure 1).

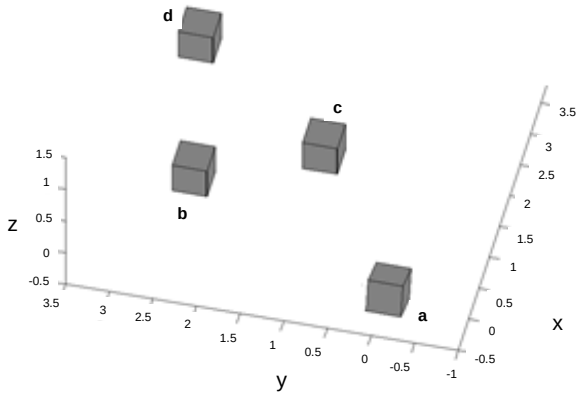


Figure 1: Initial local distribution of blocks.

Typical BW predicates such as `Stack(a,b)` are changed to operators like `Dock(a,b)` and so forth, with the "table" being the orbit in which the configuration resides. In our problem there is no robotic arm moving the blocks about as they are presumed self-actuated by thrusters located on each face.

The merit of considering 'simple' problems like these in terms of integrating sound trajectory planning and task planning components becomes apparent as the action choices of the problem scale (Cambon et al., 2003). It is also fundamental when one considers the many layers of reasoning required to perform even relatively mundane tasks in the complex harsh environment of space. By presenting the AI task planner with appropriate costs in terms of minimizing fuel use, the time-to-completion, and proximity to obstacles (for safety considerations), the trajectory planner relieves the task planner of the full representational details of block locations/motion that would typically overwhelm it, making optimal solutions prohibitive (Smith et al., 2000). Instead, the task planner uses cost values computed by the trajectory planner to influence decisions related to the optimal completion of the block docking sequence.

We lay the groundwork for facilitating a connection between AI task planning and optimal trajectory planning, after a brief discussion of related work, by defining a space-based symbolic domain representation of BW. We then present the system architecture and its component algorithms followed by results for the four-block assembly problem. The results are contrasted with assembly constructions in flat-space and in a central gravitational field (Keplerian model). We conclude with future work to extend our models/algorithms and to practically implement this system for space-based construction activities.

Some Related Work

This work embodies two main themes: self-assembly and the integration of task and optimal trajectory planning. There is a broad range of work encompassing many aspects of the self-assembly mechanics and automation required by this problem (Jones and Matarić, 2003; Suh et al., 2001; Shen, 2001; Butler and Rus, 2001; Shen et al., 2000; Rus and Vona, 1999). Some approaches focus on distributing path-planning and actuation within the system—developing a parallel local awareness (Butler and Rus, 2001), while others focus on global strategies. Both are needed in the space environment where the optimization of limited resources directly impacts mission success. To this end, either strategy requires the careful planning of the trajectories executed in the self-assembly process. One can think of this in other important contexts, such as assembling waypoints to meet military objectives or reconnaissance goals (Petterson and Doherty, 2004).

While there is much work on either problem there remains the persistent gap between the language we use for symbolic reasoning and that used for controlling autonomous motion. This 'gap' is beginning to be recognized and pursued.

Domain Description

BW is a generic domain where the blocks are representations of objects—be they freight, transportation devices, building materials, atoms, etc. The combination of abstraction and the basic premise of moving and assembling these "blocks" in particular configurations lends itself well to the problem of self-assembly.

Mapping BW to a 3D space environment necessitates a paradigm shift in the traditional representation of the 'infinite table.' Instead of a table on which all of the blocks—be they 'free' or part of a tower—reside, we introduce the notion of a 'target table' that will be the orbit in which the blocks are assembled. Blocks in other orbits may be considered free or on 'virtual tables.' Full 3D construction with local and global assembly entails such details as:

- Moving and docking/undocking block superstructures, changing the system dynamics
- Docking in any orbit
- Building structures with 'non-reachable' or variable configurations (e.g., cubic docking with an open center for unique configurations)

Before embracing these details, we have chosen to begin with a constrained construction that disambiguates block states and more closely mirrors the traditional BW paradigm—i.e., the construction of towers or linear assemblies relative to a specific anchor block freely drifting in space. This provides a simplified baseline from which to add the necessary details for unconstrained 3D construction in future work.

PDDL Domain Model

To provide a framework for discussion and to lend some familiarity for those experienced with BW planning, we define a 3D BW domain with a PDDL v.2.1 (Fox and Long, 2003) representation (see Figure 2), making the following assumptions for linear self-assemblies:

1. Blocks have two specific faces (+y,-y) to which any other block may dock.
2. Actions occur sequentially—only one block may be docked/undocked to/from another block at any time.
3. Only one connected structure may be assembled.

```
(define (domain blocks-in-space)
  (:requirements :equality)
  (:predicates
    (block ?b)                ; ?b is a block
    (orbit ?o)                ; ?o is an orbit
    (face ?f)                 ; ?f is a block face
    (in-orbit ?b ?o)          ; block ?b is in orbit ?o
    (clear ?b ?f)             ; face ?f of block ?y is clear (undocked)
    (docked ?b1 ?f1 ?b2 ?f2)  ; face ?f1 of block ?b1 is docked to face ?f2 of block ?b2
    (assembled ?b)            ; block ?b is part of the single assembled structure
    (free ?b))                ; block ?b is free to move and not assembled; both faces of ?b are clear

  (:action insert             ; insert ?block1 residing in ?orbit1 into ?orbit2
     :parameters      (?block1 ?orbit1 ?orbit2)
     :precondition    (and  (block ?block1) (free ?block1) (orbit ?orbit1) (orbit ?orbit2)
                            (in-orbit ?block1 ?orbit1))
     :effect          (and  (in-orbit ?block1 ?orbit2) (not (in-orbit ?block1 ?orbit1))))

  (:action dock               ; move free ?block1 so that its ?face1 docks to ?face2 of anchor ?block2
     :parameters      (?block1 ?face1 ?block2 ?face2)
     :precondition    (and  (block ?block1) (block ?block2) (face ?face1) (face ?face2) (free ?block1)
                            (clear ?block2 ?face2) (assembled ?block2) (not (= ?face1 ?face2)))
     :effect          (and  (docked ?block1 ?face1 ?block2 ?face2) (assembled ?block1)
                        (not (clear ?block1 ?face1)) (not (clear ?block2 ?face2)) (not (free ?block1))))

   (:action undock            ; move ?block1 to undock from anchor ?block2
     :parameters      (?block1 ?face1 ? ?block2 ?neg-face2)
     :precondition    (and  (docked ?block1 ?face1 ?block2 ?face2) (clear ?block1 ?neg-face)))
     :effect          (and  (free ?block1) (clear ?block1 ?face1) (clear ?block2 ?face2)
                          (not (docked ?block1 ?face1 ?block2 ?face2)) (not (assembled ?block1)))))
(define (problem assemble-four-blocks)
  (:domain blocks-in-space)
  (:objects       block-a, block-b, block-c, block-d, pos-y, neg-y, target-orbit)
  (:init         (block block-a) (block block-b) (block block-c) (block block-d) (face neg-y)
                 (face pos-y) (in-orbit block-a target-orbit) (in-orbit block-b target-orbit)
                 (in-orbit block-c target-orbit) (in-orbit block-d target-orbit) (clear block-a pos-y)
                 (clear block-a neg-y) (clear block-b pos-y) (clear block-b neg-y) (clear block-c pos-y)
                 (clear block-c neg-y) (clear block-d pos-y) (clear block-d neg-y)
                 (assembled block-a)  (free block-b) (free block-c) (free block-d))
```

Figure 2: 3D BW problem PDDL representation

4. Blocks either drift as part of the assembly or are free to maneuver, in which case they cannot be connected (docked) to any other block.
5. Spacecraft (blocks) have sufficient fuel for maneuvers and will always execute actions accurately.
6. All blocks are uniquely labeled. They may be interchangeable to allow random placement (as in our example), or they may have specific configuration requirements.

To achieve a goal sequence a set of constructive actions is performed given certain preconditions. When a constructive move is not possible the problem is in a 'deadlocked' state, necessitating the movement of some block before a constructive move is possible. Efficient search strategies employ methods to minimize the number of deadlocks encountered—i.e., backtracking (Slaney and Thiébaux, 2001). In this work, all moves are constructive. However, because optimal solutions are computationally expensive, there is an algorithmic tradeoff of efficiency for optimality due to the nature of the problem as discussed in the following section.

Architecture

In order to facilitate an appropriate link between task and trajectory planning, it was important to design an architecture (see Figure 3) that retained the dynamical state information for each planning state (search node) while keeping this information hidden—i.e., in a "black box"—from the task planner. For this initial implementation, a primitive C++ "task planner" interprets the PDDL domain (see Figure 2) and conducts an optimal search in which the "translator function" is invoked to acquire the cost J of the instantiated action by interfacing with a Matlab-based optimal trajectory planner (Henshaw, 2003). The task planner uses a uniform cost (Dijkstra's) search strategy with actual node *n* cost, *g(n),* set to the cost of the parent node plus additional cost *J* of transitioning from the parent to node *n*. Transition costs *J* (see Equation 2) are computed during the trajectory planning process for each action. The focus of the current task planner's

implementation is the information communicated between task and trajectory planners. We therefore discuss this communication and the design of the trajectory planner in more depth below.

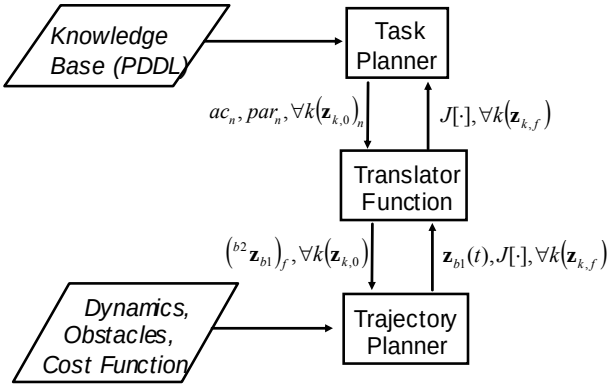


Figure 3: System Architecture

Task Planner and Translator Function

The task planner begins its search for an optimal block assembly sequence by using the knowledge base's initial full-state 'snapshot' $\mathbf{z}_{k,o}$ for all blocks k of the recently deployed system. A node queue is then constructed, with children of the initial root node formed from actions ac_n and parameter bindings par_n that are able to preferentially achieve a subgoal or the set of actions (ac_n,par_n) that meet all preconditions of ac_n if no subgoals can be directly achieved. For each uniform cost node expansion the queue must be ordered by total path cost, $g(n)$.

The translator function is then called to compute the cost J of transitioning from each parent to child node. The translator passes the continuous state $\mathbf{z}_{k,o}$ for all blocks directly to the trajectory planner. Based on the $\mathbf{z}_{k,o}$ and the action (ac_n,par_n) to be executed, the translator also computes the final (goal) state $({}^{b2}\mathbf{z}_{b1})_f$ to be achieved as the product of executing action ac_n. For all BW actions, *b1* is the block to be moved and *b2* is the anchor block or target orbit to which *b1* must maneuver. By expressing position vector $({}^{b2}\mathbf{z}_{b1})_f$ in *b2* coordinates (indicated by the leading *b2* superscript), the translator computes *b1* goal positions relative to its drifting target (*b2*), allowing maneuver time to be optimized by the trajectory planner rather than specified in advance. In the general case where all blocks drift over time (freely or in a gravitational field), all blocks will move as the trajectory for each ac_n is executed, requiring that the final state $\mathbf{z}_{k,f}$ of all blocks after executing ac_n be stored as the initial state "snapshot" for the offspring of node n. Should node *n* be part of the optimal solution, the optimal trajectory $\mathbf{z}_{b1}(t)$ for maneuvering block *b1* is archived along with the cost J returned by the translator as the cost of executing action (ac_n,par_n).

This might appear a lengthy computational process for such a simple assembly activity (Cambon et al, 2003). However, as we discuss below, incorporating complex dynamics is the only way to ensure optimal solutions worthy of space applications.

Trajectory Planner

Traditional trajectory planning strategies used in terrestrial applications for rover locomotion, etc., need to be supplemented to accommodate the unique challenges of navigating spacecraft. To navigate a *free path* from point *A* to point *B* without intersecting any obstacles, cell decomposition and roadmap methods such as Voronoi diagrams, etc. are usually employed with varying degrees of success (Latombe, 1991). However, when dealing with spacecraft one must take into account:

- Limited fuel resources/maneuverability,
- Possible encounter(s) with a wide range of obstacle operating velocities,
- Dynamic thruster constraints,
- Varied endpoint constraints/operating times.

Traditional methods fall short of meeting these dynamic constraints (Henshaw, 2003). This is especially realized when contrasting a relative 'straight-line' trajectory in flat space with the same trajectory in a central gravitational field as presented below. Additionally, the self-assembly of blocks in both environments requires the transfer of blocks from one orbit to another. For circular orbits, the optimal global planning strategy for maneuvering blocks to a target orbit is a straightforward Hohmann transfer (Miele et al., 2004).

To fully address these challenges we chose a trajectory planning algorithm specifically designed to solve end-to-end orbital docking problems involving both orbital maneuvering and proximity operations using realistic saturating thrusters (Henshaw, 2003). Robust numerical methods and the use of Calculus of Variations allows the planner to develop a cost functional that penalizes fuel use, obstacle clearance distance, and arrival time while enforcing dynamic orbital constraints. Six degree of freedom paths are found by deriving Euler-Lagrange equations corresponding to the cost functional, then solving the associated boundary value problem using collocation (implemented with the Levenberg-Marquardt algorithm) and continuation techniques, allowing for the optimization of arrival time and fuel use. To avoid error effects, a feedback control algorithm was implemented (using Pontryagin's minimum principle).

The 6-DOF dynamic equations for the moving vehicle are:

$$\dot{\mathbf{z}}[(t)] = \begin{bmatrix} \dot{\mathbf{v}} \\ \dot{\mathbf{p}} \\ \dot{\omega} \\ \dot{\sigma} \end{bmatrix} = \begin{bmatrix} -\mu\mathbf{p}/\|\mathbf{p}\|^3 \\ \mathbf{v} \\ -\mathbf{H}^{-1}S(\mathbf{H}\omega)\omega \\ \mathbf{G}_\sigma(\sigma)\omega \end{bmatrix} + \begin{bmatrix} \mathbf{R}(\sigma)\mathbf{u}(t)/m \\ 0 \\ \mathbf{H}^{-1}\tau(t) \\ 0 \end{bmatrix} \qquad (1)$$

where:

p is vehicle position relative to the anchor/target[1]

v is vehicle translational velocity relative to the target

σ is a modified Rodrigues vector (a 3-element vector representing vehicle attitude without singularities)

ω is the rotational rate vector in the body frame

m is vehicle mass,

H is the rotational inertial matrix,

$R(\sigma)$ is a rotation matrix that converts body to inertial coordinates,

S the matrix representation of cross product $\mathbf{H} \times \omega$

$\mathbf{G}_\sigma(\sigma)$ is the dynamic equation for the Rodrigues vector

$\mathbf{u}(t)$ is the force vector produced by saturating thrusters

$\tau(t)$ is the limited torque vector.

To generate the desired trajectory, a cost functional is minimized subject to the dynamic constraint, $\dot{\mathbf{z}}(t)$:

$$J[\cdot] = \int_{t_0}^{t_f} \left(L_{control}[\mathbf{z}, v, \tau] + L_{obstacle}[\mathbf{z}] + L_{time} + \lambda^T f(\mathbf{z}, v, \tau) \right) dt \qquad (2)$$

where:

$f(\mathbf{z},v,\tau)$ represents vehicle dynamics from Eq. (1)

$L_{control}[\mathbf{z},v,\tau]$ penalizes control effort—fuel use

$L_{obstacle}[\mathbf{z}]$ penalizes obstacle clearance distance

L_{time} penalizes completion time

Finally, the boundary conditions, specified by the Translator (see Figure 3), are defined by the initial state in each search node and the final waypoint computed from the initial state and choice of task to execute. Although the translator need not specify time of arrival at the final state, motion of the target must be known as a function of arrival time which is reasonable given the target vehicle's natural orbital motion. The BW domain as defined for this work specifies trajectory planning problems with fixed arrival locations and either fixed or free arrival times, allowing a relatively simple form of the transversality boundary condition $L_\alpha(\cdot) = 0$ to hold.

4-block Assembly Results

The *4*-block linear assembly problem defined in Figure 2 was cast in a circular equatorial orbit (`target-orbit`) with an orbital radius of 6767.06km (388.92km above the Earth). All blocks/modules had edge length 0.2 km, and the faces were presumed tangent (no separation) when docked.[2] Table 1 lists each block's initial position $\mathbf{p}_i(0)$ relative to anchor `block-a` and inertial orientation $\mathbf{R}(\sigma)$. Initial block relative velocities and angular velocities/accelerations were assumed zero since all blocks approximately occupy the same orbit.

Table 1: Initial block locations

Block-*i*	$\mathbf{p}_i(0)$ (km)	R(σ)
`block-a`	(0.0, 0.0, 0.0)	(0.0, 0.0, 1.0)
`block-b`	(1.0, 2.5, 1.0)	(0.0, 0.0, 1.0)
`block-c`	(2.0, 1.5, 0.0)	(0.0, 0.0, 1.0)
`block-d`	(3.0, 3.0, 1.5)	(0.0, 0.0, 1.0)

Before examining the full 4-block planning process, we motivate the use of the full-state trajectory planner by examining a single action: `(dock block-d neg-y block-a pos-y)` with only the `block-d` and `block-a` positions shown in Table 1—i.e., no other blocks acting as obstacles. The optimal trajectory and cost were compared for the dock conducted in flat-space (no gravity) where a simpler path planner might suffice, with the same problem in the circular orbit specified above.

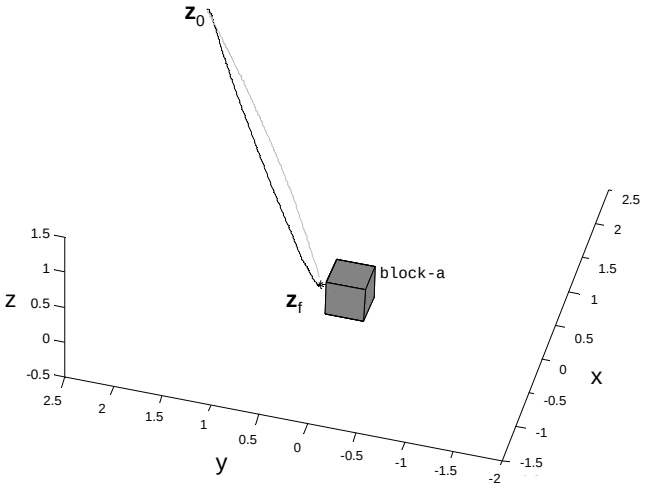


Figure 4: Paths with gravity (black) versus flat-space (grey) models.

A comparison of paths and force profiles for the two cases is shown in Figure 5. Although the physical paths through space do not differ greatly, there is substantial difference in cost. As shown in Figure 4, the force at any given time is nearly two orders of magnitude higher for the solution with gravity, however, this difference is mitigated to some extent by the reduced time to dock, thereby also lowering the time over which gravitational forces act on the block. Without gravity, the range of single docking operations for the four blocks with initial states given by Table 1 exhibited minimal difference in assembly cost. Conversely, tasks executed with the gravitational model had cost that varied by up to 80% for different docking tasks.

1 The docking target block (or target orbit) may be in motion, but this motion must be modeled a priori—a reasonable assumption for insertion into a known orbit or docking to a controlled spacecraft.

2 Each "block" has rather enormous dimensions for our example. Such size and separation distance values were chosen to simultaneously illustrate the effects of obstacle avoidance and orbital dynamics on cost without modeling significantly more than four blocks.

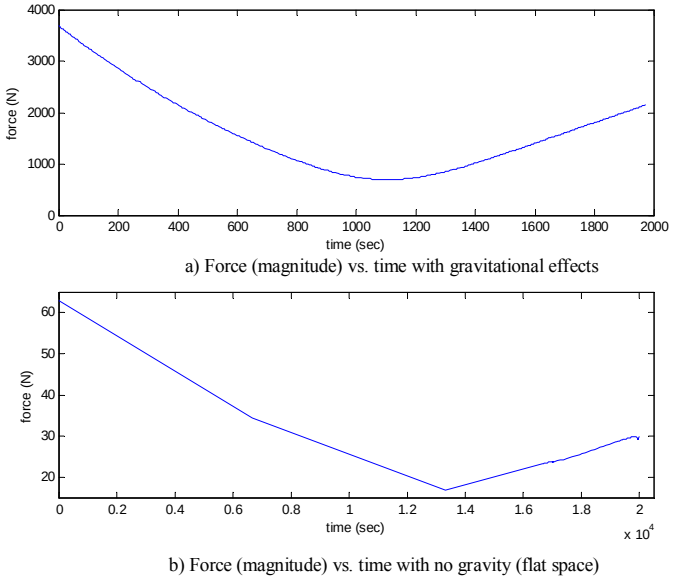


Figure 5: Comparative force plots for `(dock d -y a -y)`

Next we ran the planner (search engine) with the blocks in their circular orbit. A representation of the full search-space is shown in Figure 5 along with the total cost for a number of complete paths through the search space. This 4-block example had branching factors 6, 4, and 2 at search levels 1, 2, and 3, respectively, representing the possible combinations of `dock` operations each unassembled block could achieve. The optimal plan was computed to be: 1) `(dock block-b neg-y block-a pos-y)` (cost J=4.67E3), 2) `(dock block-c neg-y block-b pos-y)` (J=5.91E3), then 3) `(dock block-d neg-y block-c pos-y)` (J=2.29E4), with total assembly cost g=3.35E4. Note that not all Level 2 or 3 nodes were actually expanded with the uniform cost engine; select cost data is provided to illustrate the search-space and facilitate assembly structure and cost comparison.

For non-optimal assemblies, there was a significant range of individual docking task costs: a minimum = 4.67E3 (part of the optimal plan), a maximum = 7.78E4, and an average = 2.25E4. Further, the same final assembly structure does not ensure consistent cost (e.g., assembly **c-a-b-d** in Figure 4). As discussed in Section 3.2, the trajectory planner may return a different cost for the same final assembly based on obstacle avoidance requirements, illustrating the importance of task ordering choices for an optimal assembly.

The layout of this 4-block problem was chosen to demonstrate the functionality of our system and to clearly illustrate the need for the integration of task and trajectory planning. However, this integration comes at the cost of computational time; not uncommon to systems needing any degree of trajectory or even path planning fidelity (Pettersson and Doherty, 2004). The worst runs need over an hour to compute, becoming cumbersome, or even prohibitive, as the number of assembly sequences scales with the number of blocks. Even with this computational burden, there are ways to mitigate the necessity of running all assembly possibilities in the trajectory planner. Making use of selective cost information, combining global and local assembly strategies, and incorporating pre-processing that uses system dynamics to deliver intelligent, best guess cost estimates are some of the ways the system can work in both offline and online capacities.

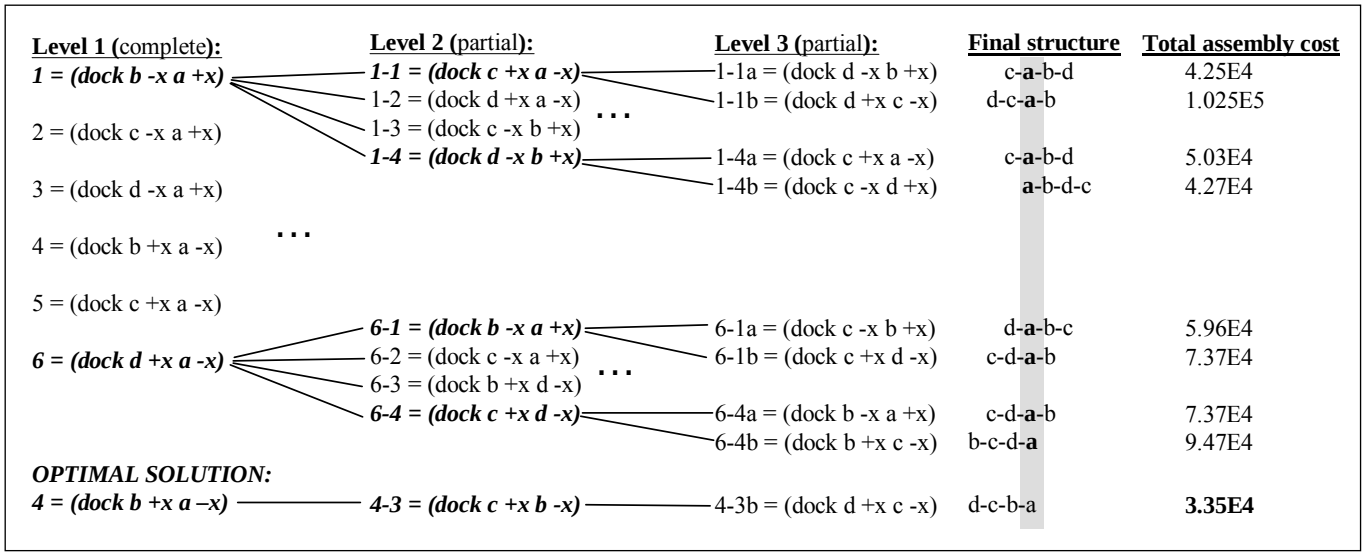


Figure 5: Four-block linear assembly search-space and assembly costs

Conclusions and Future Work

Intelligent in-space self-assembly requires careful integration of task-level and physics-based reasoning systems. We have argued that assembly task planning necessarily requires the incorporation and knowledge of complex system dynamics in the overall task planning/sequencing strategy. A framework for establishing a connection between an optimal control trajectory planner and task planner in this domain was established, from which we presented a simple assembly problem that clearly illustrated the need for a trajectory planner designed to work with continuous dynamical system models of in-space assembly problems as well as a task planner to propose action sequences that will successfully achieve assembly goals.

This work began with planning algorithms and a PDDL model designed for a small set of blocks and sequential assembly choices due to the computationally-intensive planning processes involved. Its primary contribution lies in the integration of optimal task and trajectory planners, specifically the knowledge representation and interface language that enable the task planner to manage complex trajectories while processing only a small set of symbolic features and a single measure of cost for each planning state. A secondary contribution is the PDDL 3D BW domain definition (Figure 2). Although a sophisticated trajectory planner is already in place, a more capable task planner will be required to increase search efficiency and enable parallel, multi-tasked activity schedules—an important capability when many "blocks" are assembled.

Our aim for future work is to improve algorithmic efficiency by utilizing (admissible) heuristics to reduce search-space size while maintaining optimality and expanding the task planner to handle parallel assembly task execution. The architecture and BW representation presented builds a foundation on which both AI and control systems researchers can build such extensions.

References

A. Miele, M. Ciarcià, J. Mathwig. 2004. "Reflections on the Hohmann Transfer," Journal of Optimization Theory and Applications Vol. 123, Issue 2, pp. 233 – 253.

Stéphane Cambon, Fabien Gravot and Rachid Alami. 2004. "A Robot Task Planner that Merges Symbolic and Geometric Reasoning," Proceedings of the 16th Annual Conference on Artificial Intelligence, Spain.

Pettersson, P-O., Doherty, P. 2004. "Probabilistic Roadmap Based Path Planning for Autonomous Unmanned Aerial Vehicles," Proceedings of the 14th International Conference on Automated Planning and Scheduling.

Chris Jones and Maja J. Matarić. 2003. "From Local to Global Behavior in Intelligent Self-Assembly," Proceedings of the IEEE International Conference on Robotics and Automation.

R. Lampariello, S. Agrawal, G. Hirzinger. 2003. "Optimal Motion Planning for Free-Flying Robots," International Conference on Robotics and Automation, Taiwan.

Fox, M. and Long, D. 2003. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains," Journal of Artificial Intelligence Research 20, pp. 61-124.

Henshaw, C.G. 2003. "A Variational Technique for Spacecraft Trajectory Planning," PhD Dissertation: Department of Aerospace Engineering, University of Maryland College Park.

John W. Suh, Samuel B. Homans and Mark Yim. 2002. "Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics," Proceedings of the 2002 IEEE International Conference on Robotics and Automation, Washington DC.

Jacobsen, S., Lee, C., Zhu, C., and Dubowsky, S. 2002. "Planning of Safe Kinematic Trajectories for Free Flying Robots Approaching an Uncontrolled Spinning Satellite" Proceedings of the ASME 27th Annual Biennial Mechanisms and Robotics Conference, Montreal.

John Slaney and Sylvie Thiébaux, "Blocks World Revisited," Artificial Intelligence, 125, pp. 119-153, 2001.

Wei-Min Shen. 2001. "Metamorphic Robotic Systems for Space Exploration," ICASE/USRA/LaRC Workshop On Revolutionary Aerospace Systems Concepts for Human/Robotic Exploration of the Solar System.

Zack Butler and Daniela Rus. 2001. "Distributed motion planning for 3D modular robots with unit-compressible modules," Proceedings of the Internnational Conference on Intelligent Robots and Systems.

Wei-Min Shen , Yimin Lu , Peter Will. 2000. "Hormone-based control for self-reconfigurable robots," Proceedings of the Fourth International Conference on Autonomous Agents, Barcelona.

David E. Smith, Jeremy Frank, and Ari Jónsson. 2000. "Bridging the Gap Between Planning and Scheduling," Knowledge Engineering Review 15(1).

Daniela Rus & Marsette Vona. 1999. Self-reconfiguration Planning with Compressible Unit Modules," Proceedings of the 1999 IEEE International Conference on Robotics and Automation.

J. Scott Penberthy and Daniel S.Weld. 1994. "Temporal Planning with Continuous Change," AAAI.

Jean-Claude Latombe. 1991. Robot Motion Planning, Kluwer.

Alternatives to Re-Planning: Methods for Plan Re-Evaluation at Runtime

Emmanuel Benazera
RIACS, NASA ARC, Moffet Field, CA 94035
ebenazer@email.arc.nasa.gov

Abstract

Current planning algorithms have difficulty handling the complexity that is due to an increase in domain uncertainty, and especially in the case of multi-dimensional continuous spaces. Therefore, they produce plans that do not take into account numerous situations that can occur at runtime, such as faults or other changes in the planning domain itself. Thus there is a gap between the plan generation and the reality experienced at runtime. Here we present two methods that allow the plan conditionals to be revised w.r.t. uncertainty on the system as estimated at runtime.

Introduction

The need for autonomy and robustness in the face of uncertainty is growing as planetary rovers become more capable and as missions explore more distant planets. Recent progress in areas such as instrument placement (Pedersen *et al.* 2003; 2005) makes it possible to visit multiple rocks in a single communication cycle. This requires reasoning over much longer time frames, in more uncertain environments. Simple unconditional plans as used by the Mars Exploration Rovers (MER) will probably have a low probability of success in such context, so that the robot would spend almost all its time waiting for new orders from home.

In the last decade, architectures for future planetary rover missions include a planner/scheduler, a health monitoring system, and an executive. The planner/scheduler generates a control program/plan that describes the sequence of run-time actions necessary to achieve mission goals. Since the rover's environment is highly uncertain (Bresina *et al.* 2002), the control programs (also called *plans*) are contingency plans (Dearden *et al.* 2003) in that they involve conditional branches that are based on decision functions of the system state that the executive can evaluate in real time. The executive is responsible for the execution of the control programs, taking into account the current state of the system as estimated by the health monitoring system. This capability includes deciding the best branch in a plan when reaching a branch point, given an estimate of the current system state, inserting and replacing plan portions to react to faults and other unpredictable events.

However, planners have difficulties handling certain situations, such as actions that carry no utility (typically used for responding to unlikely situations) and fault occurences, or to prepare for a belief state update[1]. First, actions with no reward can possibly be inserted anywhere in the plan at low cost, so the greedy approach that seeks to maximize the expected utility fails to position them efficiently. Second, planner domains describe a very limited set of faults, thus relying on a mostly nominal model of the world and system actions (e.g. no stuck wheels, broken navigation system, rocky environment,...). Moreover, fault models exponentially increase the complexity of the planning even if the faults have low probability of occurence as they can occur at any time during the plan execution. Finally, the health monitoring system returns an ever changing belief state over time that has to be taken into account. For these reasons, the response to unlikely situations and faults is better decided at execution: the health monitoring system passes a belief over the system state to the executive that decides which portion of the plan to execute, sometimes inserting/replacing wanted/unwanted plan blocks.

More recent architectures try to mitigate these problems by moving towards unified planning and execution frameworks (Alami *et al.* 1998; Muscettola *et al.* 2002; Estlin *et al.* 2005). Several of these architectures are discussed at the end of this paper, however it is well understood that uncertainty in future values forces an agent to plan locally. For example, to mitigate this problem, (Muscettola *et al.* 2002) allows plans to include explicit calls to a deliberative planner. This comes back to finding place where to insert a branch, and as demonstrated in (Dearden *et al.* 2003), the branch point is usually not situated at the point that has the highest probability of failure. Now note that if the process of estimating a good branching point does not forcely require to do the planning, it doesn't cost much to pre-plan the branch once the point has been identified. Therefore, the branch can be pre-planned and its values later updated during execution. As it will be explained later in this paper, re-evaluation of a plan is in no way equivalent to re-planning, but a re-evaluated plan can be found that is optimal w.r.t. the information on the uncertain system state and the original plan.

We said that most planners do not handle well the complexity due to the presence of faults in a model and therefore rarely include faults within their planning domain. Moreover, major faults are well known and recoveries can be efficiently con-

[1] Partially Observable Markov Decision Processes (POMDPs) allow the latter but are often untractable.

structed before execution. At runtime, a fault detection system, or more generally, a state estimator will return a state estimate that triggers one or more plan fragments for system recovery or opportunistic science. These plan fragments are often referred to as floating contingencies whose execution can be conditionned upon resources (including time) and/or system behavioral modes. Therefore in this paper we will refer to two types of contingencies: pre-planned branches on resources that are part of the main plan, and floating contingencies, that trigger in response to certain events and resource values. The paper focuses on techniques to re-evaluate the former, and studies the complexity added to them by the latter.

The problem can be seen as one of re-evaluating the plan values, such as its utility, and updating the plan conditionals, i.e. the branch conditions. Typically, at runtime, the probability mass of the state estimate shifts among regions of the hybrid space (continuous resources plus discrete state). We adapt the pre-computed branch conditions to these changes by projecting the changes forward and backing up the resulting states. Our first approach is an adaptation of the classical Monte Carlo (MC) technique (Sutton & Barto 1998; Thrun 2000). Our second approach is based on decision theoretic techniques and converts the problem into a small Partially Observed Markov Decision Problem (POMDP)(see (Kaelbling, Littman, & Cassandra 1998) for an introduction and more references) whose solving at runtime returns probabilistic decision lines that are optimal given the initial plan.

Preliminaries

Here a plan can be seen as a tree whose nodes are known as the branch points. The value function for a node is a continuous function over the multi-dimensional resource state, i.e. a mapping from the resource space to the utility space, and that depends on downstream node value functions. Planning determines a set ofpolicies that maximize the expected utility of the plan. At branch points, this leads to conditions over the resource space that discriminate among branches.

Typically, planning proceeds to a mapping from the system state space to the utility space, i.e. the utility obtained by executing the plan, that it seeks to maximize. Noting the system state $s = (x, r)$ with $x \in X$ the discrete state (or system modes), and $r \in R$ the multi-dimensional continuous state (including time), the utility earned by executing a branch b_i starting at s can be noted:

$$V_{b_i}(s) = \sum_{x' \in X} \int_R p((x', r') \mid s, a_{i1})[U(a_{i1}, (x', r')) + V_{B_i}(x', r')]dr' \quad (1)$$

with a_{i1} the first action of branch b_i, B_i the remaining portion of the branch, $U(a_{i1}, (x', r'))$ the utility earned, and s' the system state after executing a_{i1} following the probability distribution $p(s' \mid s, a_{i1})$. Over a belief state $\pi(s)$, as estimated by the health monitoring system, we have:

$$V_{b_i}(\pi(s)) = \sum_{x \in X} \int_R V_{b_i}(x, r)\pi(x, r)dr \quad (2)$$

And at a branch point where n branches are available, the best branch is decided according to:

$$b^* = arg \max_{i \in [1,n]} V_{b_i}(\pi(s)) \quad (3)$$

This is similar to the Bellman equations for POMDPs (Boyan & Littman 2000). Each value function $V(b)$ maps the resource space to the utility of the branch b. The $\max$ operator of relation (3) defines an upper bound on the branch point overall utility value, and branch conditions are found at the functions intersections. At execution, deviations from the planning domain and information of the state estimate move these decision lines.

There are several conditions and situations under which the plan value must be re-evaluated. First, when the execution encounters a branch point, any change in the Bellman equation functions, such as the belief b over the state s, the reward model U, the action cost model, requires that all branch functions at this branch point are re-evaluated. Second, if not at a branch point, but if a floating branch has to be inserted, then the plan equation is changed and the remaining portion of the currently executed branch as well all future branch conditions must be re-evaluated. For example, when inserting a branch b_f, equation (1) becomes:

$$V_{b_f}(s) = V_{b_f}(s) + \sum_{x' \in X} \int_R p((x', r') \mid s, b_f)V_B(x', r')dr' \quad (4)$$

where B is the remaining portion of the current plan to be executed after b_f. The local value of b_f is the expected reward from the actions within the floating branch itself. The remaining term is a representation of the end state of the local plan, including the probability of the resources remaining after executing the local plan.

The remaining of the paper studies approaches to the fast re-evaluation of these decision lines.

The Monte-Carlo Approach to the Re-Evaluation of Contingency Plans

Approximating branch average utility

Applying Monte Carlo techniques to the approximation of equation (2) is straightforward: the integral over the multi-dimensional continuous space is turned into a sum by sampling N times from $b(s)$ and $p(s' \mid s, a)$, and the utility is averaged over the successive runs. We note:

$$\hat{V}_{b_i}(\pi(s)) = \sum_{x \in X} \sum_{x' \in X} [U(a_{i1}, s'_j) + \hat{V}_{B_i}(s'_j)] \quad (5)$$

where $s'_j \sim p(s' \mid s_j, a_{i1})$ and $s_j \sim b(s)$. The larger the N, the better the fit to the underlying probability distributions, and the better the approximation.

Plan simulation

For simulating branches with MC, we use a prioretized pile of events including plan actions, and a set of constraints among them. The pile is filled up with actions whose execution is simulated by testing their temporal constraints and sampling their consumption before being rewarded and popped out.

Sampling decisions

We sample the decision by deciding the path with highest utility for each sample. We write:

$$\hat{V}^{dec}(\pi(s)) = \frac{1}{N}\sum_{j=1}^{N} \max_{i\in[1,n]} \hat{V}_{b_i}(\pi(s)) \quad (6)$$

In algorithm 1, each path is explored by each sample for the evaluation of the $\max$ operator. The averaged returned utility is near optimal, but the sampled decision for the best branch (the $\arg$ operator) depends on the sampled resource space that must be partitioned into subregions of identical decision.

1: **for all** $j < N$ **do**
2: Proceed with MC on the first branch.
3: **for all** branches b_i at branch point **do**
4: Apply this algorithm recursively to b_i, with $j = 1$.
5: Return the highest utility at this branch point ($\max$).
6: Return the averaged utility of the plan.

Algorithm 1: Recursive procedure for sampling decisions

Floating contingencies

Floating contingencies are a challenge to the simulator because they can trigger at anytime. The simulator uses random events to trigger these branches and specific dynamic constraints to handle their insertion. The complexity increase due to floating branches is a product of the number of plan actions, actions in the branch, and the number of these branches. The next section covers the retrieval of the decision lines in the multi-dimensional resource space.

Bounding the resource space for deciding future branches

Decision at branch points can be made based on the simulation results by executing the branch with the highest earned utility average. Simulation provides sufficient information for computing branch conditions at future branch points. This operation is performed at virtually no cost and can spare future simulations by constraining future decisions.

Approximating branch decision lines thru piecewise constant value function approximation Our solution is to slice the resource domain into rectangular bins and to fit the branch value functions in each bin with a piecewise constant function, based on the MC samples. Function intersections are found at bin edges. Noting Δ_r a bin in the resource space, we can write b_i's value:

$$\hat{V}_{b_i}(\pi(s)) = \sum_{\Delta_r}\sum_{x\in X} p(b_i \mid \Delta_r)p(\Delta_r)\hat{V}_{b_i}(\Delta_r, x) \quad (7)$$

i.e. as the sum of the average utilities of b_i in each bin when it is the branch with the highest expected utility. More precisely:

$$\hat{V}_{b_i}(\Delta_r, x) = \frac{1}{n_{r_{\Delta_r}}}\sum_{r_j\in\Delta_r}\sum_{x'\in X} \hat{V}_{b_i}(s_j) \quad (8)$$

with $s = (x, r)$ and $s_j = (x', r_j)$, is the average utility of b_i on bin Δ_r from the $n_{r_{\Delta_r}}$ samples r^j it contains,

$$p(b_i \mid \Delta_r) = \frac{1}{n_{r_{\Delta_r}}}\sum_{r_j\in\Delta_r} \delta(b_i = arg\max_{i\in[1,n]} \hat{V}_{b_i}(\pi(s_j)) \quad (9)$$

where δ is the Dirac function, is the probability for b_i to be the branch with the highest utility over the samples of the bin,

$$p(\Delta_r) = \frac{n_{r_{\Delta_r}}}{N} \quad (10)$$

is the probability of the bin itself. An optimal bin size W is obtained, in the sense that it provides the most efficient unbiased estimation of the probability distribution function formed by the samples. We used $W = 3.49\sigma N^{-1/3}$ where σ is the standard deviation of the distribution, here estimated from the samples (D. 1976; A.J. 1991). The overall strategy is presented on algorithm 2.

1: Proceed with algorithm 1 and collect samples at branch point.
2: **for all** branch points in the contingency plan **do**
3: Compute the optimal bin size and slice the space into bins.
4: Compute statistics with equations (8), (9) and (10).
5: Evaluate equation (7) for each branch.
6: In each bin, identify the branch with the highest value.
7: Identify new branch conditions where successive bins have different highest utility branches.

Algorithm 2: Branch conditions approximation thru piecewise constant value function approximation

Branch conditions are obtained by comparing the branch with the highest utility for each bin: if two successive bins return different results, a branch condition exists at their edge. Thus, the precision of the approximation is directly dependent on the optimal bin size, that depends on the number of samples. Stutter at decision point can be overcome by fitting the successive piecewise constant approximations with more smoothly curve.

Belief update on re-evaluated branch conditions The re-evaluated decision functions are inequalities of the form $r' \leq (\geq) g(r)$. Given a state estimate $\pi(s)$ at branch point, decision over n branches follows:

$$\begin{aligned} b^* &= arg\max_{i\in[1,n]} \sum_{x\in X}\int_{r\leq g(r)} V_{b_i}(x, r)\pi(x, r)dr \\ &\approx arg\max_{i\in[1,n]} \sum_{x\in X}\sum_{\Delta_{r'}} p(b_i \mid \Delta_{r'})p(\Delta_{r'})\hat{V}_b(\Delta'_r, x)\pi(r(11) \end{aligned}$$

with r' such that $\forall r' \in \Delta_{r'}$, $r' \leq g(r)$.

Discussion

The major drawback of the Monte-Carlo approach is that it provides a probabilistic guarantee of its results, that is never absolute. This is a problem that we partially address in the next

section with the use of a decision theoretic formulation. Another work, (Jain & Varaiya 2004) finds bounds on the number of samples for the convergence of the expected reward for a class of policies.

Decision theoretic approach to plan re-evaluation

Another problem with the MC approach is that the decision is made based on a mapping from the continuous resource space to the utility space that forces the approximation of the decision lines. An alternative is to use a mapping from the belief space over the decisions to the utility space. The decision space is finite, made of the branch conditions of the original plan. The belief space over the decision is continuous and of dimension the number of decisions minus one. This formulation leads to an enlarged space but allows the use of decision theoretic techniques to directly incorporate the belief space in the computation of optimal decision lines. More precisely our problem can now be casted into a small POMDP whose actions are the plan branches, the states the branch conditions, the observations the system states.

Plan reduction to a POMDP

A standard POMDP is made of a set of actions, a set of states, a set of transitions among states per action, and a set of observations. In our model, we abstract away the actions and use a branch an action for the POMDP. Our POMDP is then defined as a tuple $(F, S, B.L, T, R)$ where:

- F is a finite set of branch decision outcomes (as states),
- S is a finite set of system states (as observations),
- B is finite set of branches (as actions),
- $P(s \mid b, f')$ is the probability of state s given that branch b has been executed and has landed in f',
- $P(f' \mid b, f)$ is the probability of entering outcome f' after taking branch b in outcome f,
- $R(f, b)$ is the reward for taking branch b while in outcome f.

The POMDP belief update can be expressed as:

$$\pi_b(f', s) = \frac{P(s \mid b, f') \sum_{f \in F} P(f' \mid b, f)\pi(f)}{p(s \mid b, \pi)} \quad (12)$$

where π is a probability distribution (belief) over F, given s and b, and:

$$P(s \mid b, f') = \frac{P(f' \mid b, s)p(b, s)}{p(f')} \quad (13)$$

The value of executing branch b under decision f and state s is:

$$V(f, s) = R(f, b, s) + \gamma \sum_{f' \in F} P(f' \mid b, f) \sum_{s^i \in S} P(s^i \mid f', b)V(s^i, f') \quad (14)$$

where in the absence of floating contingencies (because f can only lead to b):

$$P(f' \mid b, f) = P(f' \mid b) = \sum_{s' \in S} P(f' \mid b, s')p(s') \quad (15)$$

and $R(f, b, s) = V_b(b(s))$, from equation (2). Finally the value of executing branch b from some belief state π and observing s is:

$$V_s(\pi_b) = \sum_{f \in F} \pi(f, s)V(f, s) \quad (16)$$

and the optimal value function is given by:

$$V(\pi) = \max_{b \in B} \sum_{s \in S} p(s)V_s(\pi_b) \quad (17)$$

Simulation

The successor states s' and and the $p(s')$ of equation (15) are unknown and must be obtained through simulation. As a simulator we use the MC algorithm of the previous section and generate both the $\hat{V}_b(s)$ and the s' in a depth first forward search in the plan tree.

Solving

The solving of this POMDP returns a piecewise linear convex value function that is a mapping from the belief space over the decision outcomes to the highest expected plan uility. Optimal branch conditions are found at the intersections of maximized value functions and are now conjunctions of inequalities of the form $P(r \leq h(r)) \leq c$ where $r \leq h(r)$ is the branch condition from the original plan and c a constan in $[0, 1]$. For any belief over an outcome, the solution returns the optimal policy, w.r.t. the original plan.

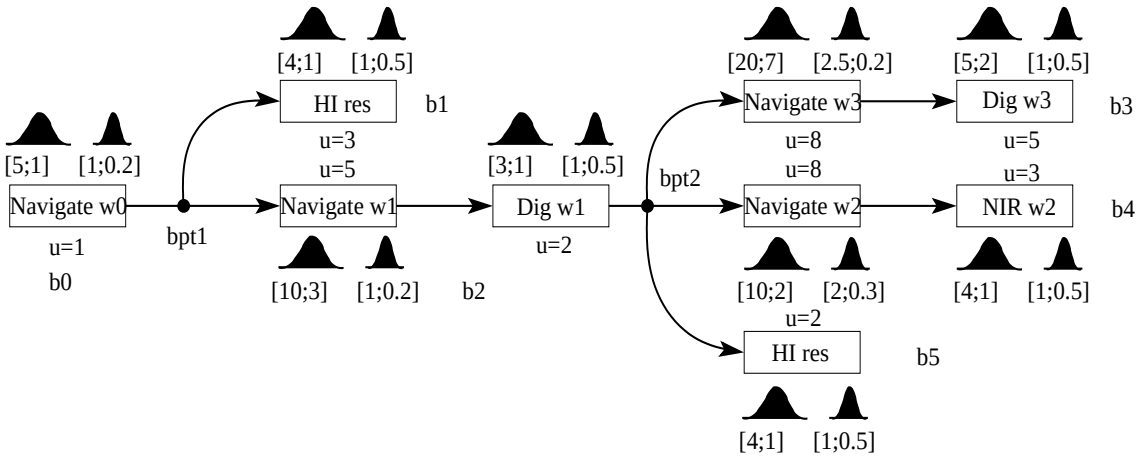
Floating contingencies

Floating contingencies pose a serious problem to the decision theoretic approach because the possible interruption of any action within a branch leads to a potentially infinite number of actions (breaking up a branch an infinite number of times over resource and time values with non null probability). Approaches like (Younes & Simmons 2004) can be used here to handle the asynchronous events, but do not allow to include the events (here floating contingencies) within the policy (therefore the computation of their conditions is not possible). While we are not yet sure about the range of solutions to this problem, it seems realistic to research approximations of floating conditions over a single branch.

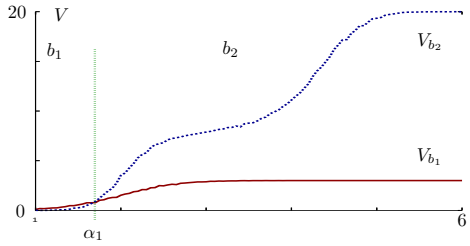
Results

A contingency plan for the Mars exploration domain

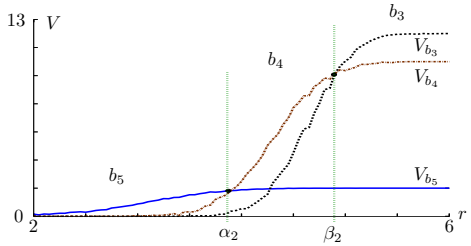
Our application is on a planetary rover plan. Consider the plan for a Mars rover on figure 1. It tells the rover to first navigate to a waypoint w_0, and there to decide whether to take a high resolution image of the point (HI res) or to move forward to a second waypoint w_1 depending on the level of resources (here energy and time). After reaching w_1 and digging in the soil, it



(a) Contingency plan for the Mars rover domain



(b) Value functions of branches at branch point 1 (bpt1)



(c) Value functions of branches at branch point 2 (bpt2)

Figure 1: Branch value functions at branch point for a detailed rover problem

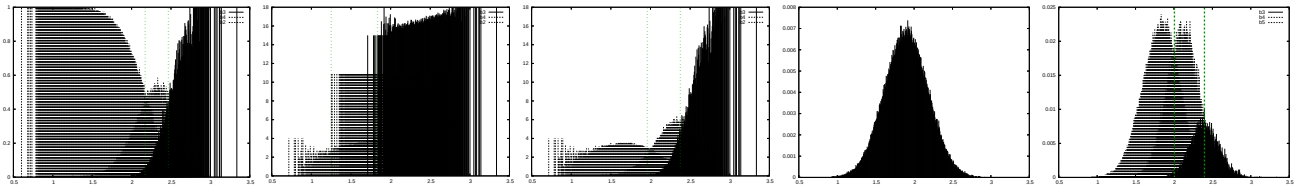
must decide whether to move forward to waypoints w_3 or w_2 or to simply get an image at w_1 and wait for further instructions. NIR is a spectral image of a site or rock. Action time and energy consumptions are represented as Gaussian bumps of empirical mean and variance. In this example branch conditions at branch points $bpt1$ and $bpt2$ have the following parameters: $\alpha_1 = 0.1$, $\alpha_2 = 2.1$ and $\beta_2 = 2.2$.

Decision sampling

Branch conditions re-evaluation at branch points: bounds/bins are generated with the sampling decision algorithm, and verified by running a classical Monte-Carlo simulation, that does not maximize the utility, but follows the new branch conditions and averages the earned utility. Simulation also returns the failure probability of the plan. The error is the difference to the optimal plan value in percentage. The piecewise constant approximation of the branch value functions returns godd utility (Table 1). Figure 2 pictures results for the second branch point of our rover problem (the energy is pictured and the time line is omitted) and shows the shifting branch conditions on the horizontal axis that is the energy line.

N	Value	Time	V dec	err
100	14.21	0.03	10.9	23.3
500	13.618	0.16	9.732	28.53
2500	13.8244	0.78	11.2992	18.26
12500	13.8008	4.08	11.9542	13.27
62500	13.7835	20.79	12.156	11.8
312500	13.7717	120.3	12.1214	12
500000	13.777	223.89	12.1814	11.58

Table 1: Monte-Carlo decision sampling and branch condition re-evaluation based on MC samples. Results are as follows: N is the number of samples, V is the mean expected highest value obtained for the plan, $Vdec$ is the value obtained when using the re-evaluated branch conditions, err the error percentage to the simulated best value.



(a) $p(b_i \mid \Delta_r)$ (b) $\hat{V}_{b_i}(\Delta_r)$ (c) $p(b_i|\Delta_r)\hat{V}_{b_i}(\Delta_r)$ (d) $p(\Delta_r)$ (e) $\hat{V}_{b_i}(\pi(s))$

Figure 2: Piecewise constant approximation of branch value functions from simulation samples.

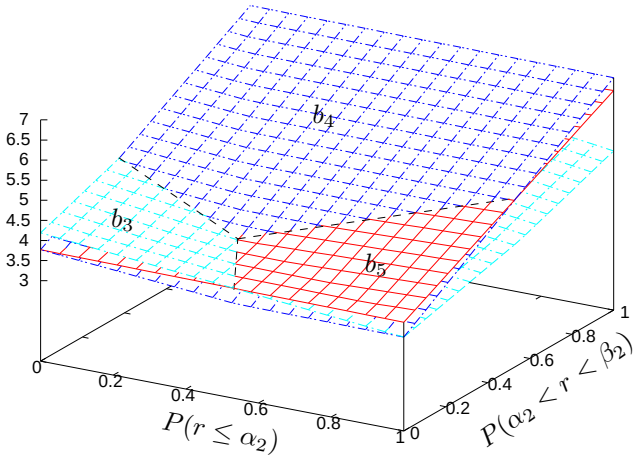


Figure 3: Optimal value function over the original plan branch conditions: x and y axis represent the probability of decision outcomes $P(r \leq \alpha_2)$ and $P(\alpha_2 \geq r \leq \beta_2)$. $P(r \geq \beta_2)$ is deduced from them.

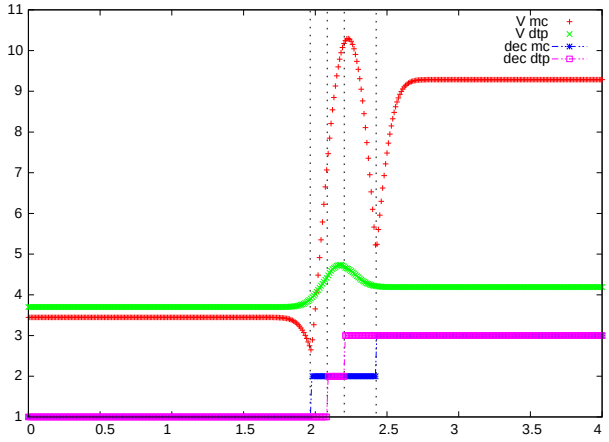


Figure 4: Comparition between the Monte-Carlo and the decision theoretic re-evaluation methods.

Decision theoretic approach

We converted our example to a POMDP and simulated the observations and rewards, respectively the system states and branch value functions. Starting from a fixed level of resources, figure 3 shows the convex value function solution for the second branch point (bpt_2).

Comparison and Discussion

To compare the two approaches, we moved a gaussian belief of fixed variance 0.1 along the resource (energy) line and studied the decision for each resource value. Results are presented on figure 4. *V mc* and *dec mc* respectively denote the value obtained and the decision based on the Monte-Carlo method with $b_5 = 1$, $b_4 = 2$ and $b_3 = 3$; *V dtp* and *dec dtp* are based on the decision theoretic planning (dtp) approach. First, the difference in value between the two methods is due to the high level of branch failure (i.e. resource gets to zero) in the simulation for the decision theoretic approach (since it is based on the original branch conditions). This is of medium importance only when we study the decision making: we observe that decision to go to branch b_4 then to b_3 based on MC are respectively slightly early and slightly late, and this is visualized as two sudden drops in expected value; the decision based on dtp switches later to b_4, and earlier to b_5 right at the highest value point. Overall the dtp-based decision leaves less room for branch b_4, which can be surprising when looking at the large surface corresponding to b_4 on figure 3 but is explained by the fact that a high probability for on decision $2.1 < r < 2.2$ denotes a more accurate belief (given the fixed variance) than for other branches. Finally it is difficult to fully assess the dominance of one method over the other. At this point of our research we lean in favor of the MC approach for plans with a small number of actions and a high number of decision outcomes, and for the dtp approach when a high number of actions in a plan makes the price of successive MC simulations costly.

Related Work and Conclusion

We have presented a simple strategy for the robust execution of contingency plans under uncertainty. It re-evaluates branch value functions at branch point and re-estimates branch conditions whenever necessary. The framework allows runtime insertion/replacement of plan portion thru the use of floating

branches but much work on their full integration into the re-evaluation process should follow. This is the first step towards the development of more powerful techniques for planning and execution under uncertainty. The MC approach is flexible and provides good results in any situation given that a sufficiently high number of samples is used. The algorithms presented are a baseline capability, and will be used later to assess the quality of more complex and focused approaches.

Related Work

Other works on plan re-evaluation include (Gough, Fox, & Long 2004) that studies plan execution with uncertainty on the resource consumption. However, the executed plans are no contingency plans as branch execution is not conditionned on decision functions over the resource state. (Washington & Lees 2004) develops a fast method for plan portions insertion/replacement, but partly fails here as it relies on pre-computed value functions (this is not always possible as faults change the model of actions).

Mixed planning/execution include (Alami *et al.* 1998) that uses a deliberative planner and an executive on top of a set of reactive controllers. (Estlin *et al.* 2005) presents the Closed-Loop Execution and Recovery (CLEaR) system that is intended to run on rovers with little communication with ground. CLEaR closely integrates the CASPER continuous planner (Chien *et al.* 2000) and the TDL executive system (Simmons & Apfelbaum 1998). Plan re-evaluation and the methods described in this paper can be seen as an alternative to the iterative plan repair of CASPER. We view plan re-evaluation as an intermediate step between execution of pre-planned contingencies and re-planning. Re-planning will always be necessary as if a situation occurs on-board for which there is no pre-planned contingency, the rover must wait for instructions. In that sense, plan re-evaluation complements architectures such as (Muscettola *et al.* 2002) and (Estlin *et al.* 2005).

For solving the decision theoretic problem, fast techniques such as (Feng & Zilberstein 2004) allow the solving of rather large problems. Given we abstract away actions within the branches when formulating the POMDP, we see our problems (not including the floating contingencies) as being of a small size.

Future Work

Future work includes dealing with floating contingencies within the decision theoretic framework, pre-computing more advanced branch value functions at planning time (Feng, Meuleau, & Washington 2004) and using them at runtime. Another hot topic remains the re-evaluation of plans that contain concurrent actions. Also note tha new class of problems recently arised in the rover domain, where the robot is able to satisfy only a subset of the goals (Smith 2004). In that case, re-evaluating the plan is not as efficient anymore because the change in resource consumption would in general lead to the selection of a different subset of goals.

Acknowledgements

Ideas in this paper are based on the experience and work of a group of current and past researchers at NASA Ames Research Center. The author thought it was time to bring some of these ideas to life and share the accumulated experience, and thanks R. Washington, R. Dearden, S. Narasimhan, H. Cannon, T. Willeke and D. Roland.

References

A.J., I. 1991. Recent developments in non parametric density estimation. *Journal of the American Statistical Association* 413(86):205–224.

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. 1998. An architecture for autonomy. *International Journal of Robotics Research* 17(4).

Boyan, J., and Littman, M. 2000. Exact solutions to time-dependent mdps. In *Advances in Neural Information Processing Systems 13*, 1–7.

Bresina, J.; Dearden, R.; Ramkrishnan, S.; Smith, D.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for ai. In *Proceeddings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling, Breckenridge, CO*.

D., S. 1976. On optimal and data-based histograms. *Biometrika* 66:605–610.

Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D.; and Washington, R. 2003. Incremental contingency planning. In *ICAPS-03: Proceedings of the Workshop on Planning under Uncertainty and Incomplete Information*, 415–428.

Estlin, T.; Gaines, D.; Chounard, C.; Fisher, F.; Castano, R.; Judd, M.; Anderson, R.; and Nesnas, I. 2005. Enabling autonomous rover science through dynamic planning and scheduling. In *to appear in IEEE Aerospace 2005*.

Feng, Z., and Zilberstein, S. 2004. Region-based incremental pruning for pomdps. In *20th Conference on Uncertainty in Artificial Intelligence (UAI-04)*, 146–153.

Feng, Z.; Meuleau, N.; and Washington, R. 2004. Dynamic programming for structured continuous markov decision problems. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*.

Gough, J.; Fox, M.; and Long, D. 2004. Plan execution under resource consumption uncertainty. In *Proceedings of the Workshop on Connecting Planning Theory with Practice at ICAPS-04*, 24–29.

Jain, R., and Varaiya, P. 2004. Simulation-based value function estimates of discounted and average-reward mdps. In *Proceedings of the Conference on Decision and Control, 2004*.

Kaelbling, L.; Littman, M.; and Cassandra, A. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101:99–134.

Muscettola, N.; Dorais, G.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. Idea: Planning at the core of autonomous reac-

tive agents. In *in Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*.

Pedersen, L.; Bualat, M.; Lees, D.; Smith, D.; and Washington, R. 2003. Integrated demonstration of instrument placement, robust execution and contingent planning. In *Proceedings of the 7th Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*.

Pedersen, L.; Smith, D.; Deans, M.; Sargent, R.; Kunz, C.; Lees, D.; and S.Rajagopalan. 2005. Mission planning and target tracking for autonomous instrument placement. In *Submitted to 2005 IEEE Aerospace Conference*.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *Proceedings of the Intelligent Robots and Systems Conference, Vancouver, CA*.

Smith, D. 2004. Choosing objectives in over-subscription planning. In *Proceedings of ICAPS-04*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, 1998.

Thrun, S. 2000. Monte carlo POMDPs. In Solla, S.; Leen, T.; and Müller, K.-R., eds., *Advances in Neural Information Processing Systems 12*, 1064–1070. MIT Press.

Washington, R., and Lees, D. 2004. Utility-based plan insertion for continuous resources. In *Proceedins of the IEEE 2004 International Conference on Robotics and Automation*.

Younes, H., and Simmons, R. 2004. Solving generalized semi-markov decision processes using continuous phase-type distribution. In *In Proceedings of the NineTeenth National Conference on Artificial Intelligence - AAAI-04*.

State-Based Models for Planning and Execution

Matthew B. Bennett, Russell L. Knight, Robert D. Rasmussen, Michel D. Ingham

Jet Propulsion Laboratory, NASA
4800 Oak Grove Drive
Pasadena, CA 91109
{Matthew.B.Bennett, Russell.L.Knight, Robert.D.Rasmussen, Michel.D.Ingham}@ jpl.nasa.gov

Abstract

Many traditional planners are built on top of existing execution engines that were not necessarily intended to be operated by a planner. The Mission Data System has been designed from the onset to have both an execution and planning engine and provides a framework for defining state-based models that can be used to coordinate planning and execution. The models provide a basis for ensuring the consistency of assumptions made by the execution engine and planner, and a basis for run-time communications between the planner and execution engines.

Introduction

Many traditional planners are built on top of existing execution engines that were not necessarily intended to be operated by a planner. The planner must model the execution engine s behavior, must make the same assumptions about the real world (which may be hidden in the code), and must be aware of its quirks. For example, a command may be issued by an execution engine and have side effects on the execution of subsequent commands. The effects may be both on the behavior of the execution engine and on what is being controlled in the real world. A planner must be aware of these effects to generate plans that will succeed when executed. Because there is generally no well-defined structure in the execution engine to model real-world effects, and because planners generally have no rigorous model of the executive s behavior, it makes it difficult to build planning and execution engines that work together under the same assumptions. Furthermore, it is difficult to keep them consistent in a parallel development effort.

The Mission Data System (MDS) project at the Jet Propulsion Laboratory has developed a control architecture, modeling framework, and systems engineering methodology for developing state-based models of real-world behavior, effects, and execution engine behavior, which are used to inform both planning and execution. More specifically, these models provide the basis for ensuring the design-time consistency of assumptions in the execution engine and planner, and are the basis for run-time communications and coordination between planners, schedulers, and execution engines. MDS calls these models State Effects Models and the methodology by which they are developed State Analysis. The modeling framework has been designed to be an open architecture for applying various formalisms and algorithms for spacecraft operations planning and execution.

In MDS, state-based models provide a basis for communication between planners, schedulers, and the engines that execute scheduled plans as follows:

(1) MDS has clearly defined roles in the architecture for planning and execution.
(2) MDS has defined semantics of execution in terms of state-histories that are represented using state constraints (goals). The MDS architecture has well-defined interfaces for exchanging information between the execution and planning engines.
(3) MDS handles inaccuracies in modeling by explicitly representing uncertainty in state estimates produced during execution, using this uncertainty in controllers, planning for uncertainty, and monitoring and enforcing the planned level of uncertainty during execution using estimators.
(4) MDS deals with uncertainty in activity duration and event timing using flexible time, uncertain time intervals, and worst-case state predictions limited by temporal constraints that impose deadlines.
(5) MDS State Analysis is a systems engineering methodology that with the MDS software frameworks bridges the gap between how execution is treated in the planning process, and what happens when the resulting plan is actually executed [2].

Execution and Planning Semantics

MDS uses constraint-based semantics to model execution behavior. A constraint (goal) on a state variable represents a set of possible state trajectories over an interval of time. A state trajectory is estimated during execution and must agree with the planned constraints for the plan to execute successfully. A state trajectory for state variable is represented in MDS using a state value history (see figure 1)[4], and is updated by a state variable s estimator during execution to reflect the system s best estimate of what the

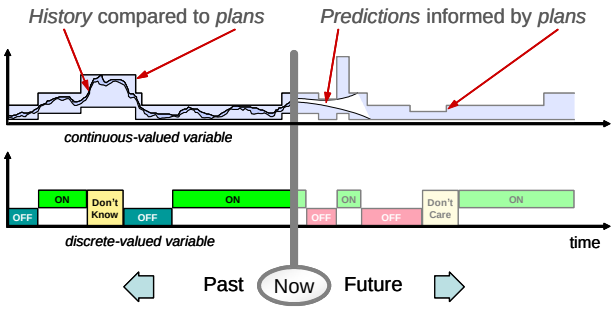


Figure 1: State value histories and plans shown as timelines

state actually was. A state variable s controller and estimator enforce the planned constraints during execution.

Plans consist of constraint networks containing both temporal and state constraints. State constraint semantics are based on set theoretic operations (such as union and intersection) over sets of possible state histories. These semantics are used by the planner to generate plans.

The MDS architecture has designed goal networks to fully describe plans as well as being directly executable. Planned goal nets are monitored by the execution engine, which issues goals to state variables when preconditions are met. Temporal preconditions are represented directly in the goal network. State preconditions are monitored by the goal checker by consulting state information stored in state value histories, and by consulting the readiness of a state s achievers to begin enforcing a constraint.

Roles and Interfaces between the Planning and Execution Engines

MDS has clearly defined roles in the architecture for planning and execution. State variables and achievers provide well-defined interfaces between the planner and the execution engine. State variables store state history during plan execution. State variables also reference plans as they are developed by the planner and make them available for inspection by the execution engine. Achievers execute planned state constraints as part of the execution engine and model their own execution behavior. State variables are consulted by the planner for modeling information about the physics of the state to be controlled (state effects model) and the execution behavior of the state s achievers. For states that are actively estimated or controlled, the state variable consults its achievers (estimators and controllers) for determining the execution behavior, otherwise the physics in the state effects model (discussed in the next section) is sufficient to describe a state s behavior.

The goal checker is part of the execution engine. It issues goals to state variables to be executed when certain preconditions are satisfied. Some of these preconditions are timing constraints as developed in the plan, the others are dependent on current state information and the capabilities of achievers. These other preconditions are provided through state variables. The state variables consult the achievers for a subset of preconditions related to modeling the execution capabilities of achievers (such as whether or not an achiever can execute a goal). During the execution of a goal, an achiever can use the same model for determining which commands must be issued, and what algorithms must do to control and estimate a state. The state variables and achievers provide centralized places for storing modeling information that is consistent for use in both planning and execution. Rule and procedural based approaches could be embedded within achievers to model and inact achiever execution behavior.

State Effects Models

The MDS architecture defines the states of the system to be controlled using state effects models. The state effects model provides the basis for communication between planners and execution engines. By defining state effects to be modeled in a single unified fashion ensures that both the planner and execution engines can make the same assumptions about the system they are controlling.

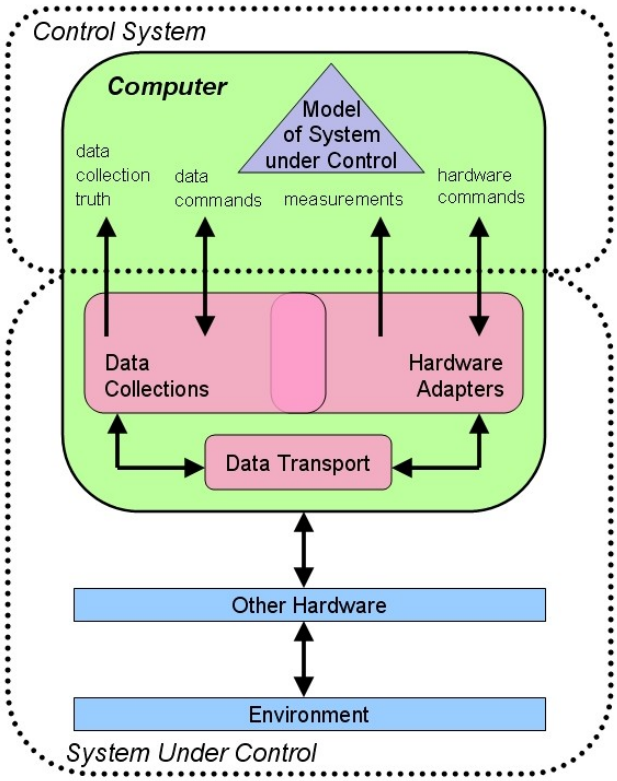


Figure 2: State effects model the physics of the system under control and the interface to the system under control in terms of commands and measurements.

State effects models define

(1) Physical models of the dynamic behavior of states, including the physical effects between states,
(2) Measurements that the control system uses to estimate the states being controlled, and how states affect measurements, and
(3) Commands that the control system uses to control the states of the system under control, and how the commands affect states.

The state effects model is used during planning and execution to

(1) Decompose the user s intent into a plan of coordinated constraints on affecting states needed to achieve the intent,
(2) Validate the plan against predictions of states based on initial conditions and predictions of affecting states, and
(3) Generate state predictions to be checked and optionally enforced during execution.

Plan Decomposition

Decomposition of user intent is called by MDS the process of elaboration. Each high-level goal on a state elaborates to a constraint network on states that affect the state, as defined by the state effects model. For example, if the user intent is to have a new picture, then this high-level goal may elaborate to goals on the power state of the camera, the operational mode state of the camera, the data storage resource state, etc.

Plan Validation

A plan is validated against its predictions by using the state effects model to compute projections for each state in the plan. Projections are represented as state constraints. For

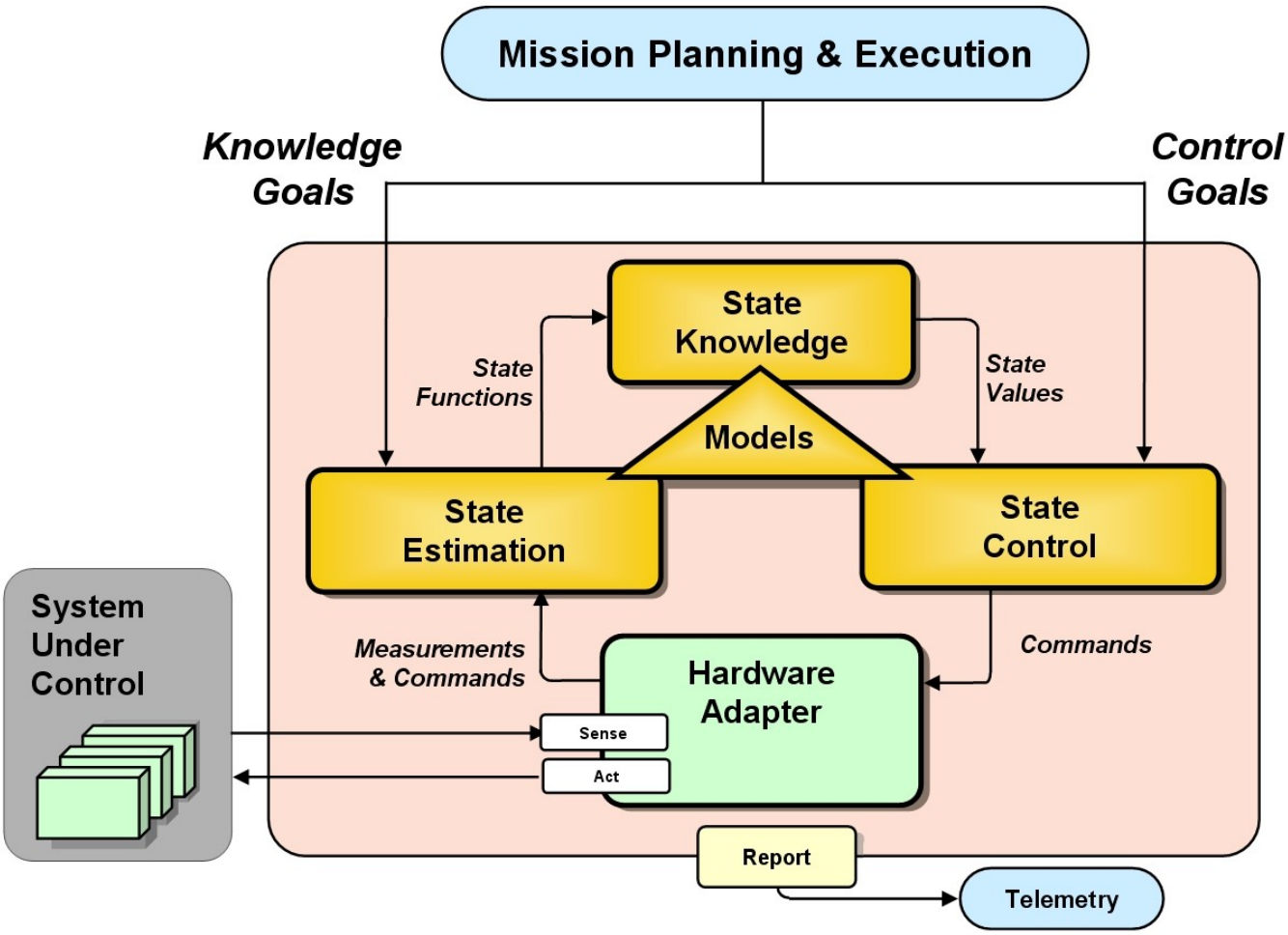


Figure 3: The MDS control system architecture.

example, a state effects model for a battery state of charge is defined as being affected by the all the power sink and source states. Thus, the projection for the battery state of charge is computed according to a state effects model where the battery state is equal to the integral of the sum of all power sources minus the integral of the sum of all power sinks. The plan for power sources and sinks is integrated and checked against an overall constraint on the battery state of charge to be above 10%.

Prediction Enforcement

During execution, this projection can be checked as a constraint on the battery state of charge. If it appears that the charge is falling below its projection, the execution engine can decide whether or not to recompute the projection to see if there is a real problem, or to exercise a another plan.

State Estimation

State estimators in the execution engine use state effects models for measurements, commands, and state-to-state effects to estimate the states to be controlled. The execution engine checks the state estimates against the plan to ensure that the plan for having proper state knowledge is executing properly, and to determine when the plan can progress. An example state effects measurement model is a camera s power switch measurement that is affected by its corresponding switch open/closed physical state. The estimator for the power switch state uses this measurement to estimate the state of the power switch. In the absence of a measurement, the estimator could use the power switch s state effects command model. In this case, the state effects model would describe how the state of the power switch is affected by the power switch command. The estimator would use this model to estimate the state of the power switch based on the last power switch command issued (and potentially the health state of the power switch). In the absence of either a command or a measurement, the state estimator could infer the state of the switch by consulting the camera operating mode state variable. It would be correct do this by reasoning from the following chain of state-to-state effects in the state effects model: the power switch state affects the power use, which in turn affects the camera operating state.

State Control

State controllers in the execution engine use the state effects models for commands to determine when to issue commands. When the execution engine issues constraints to a controller to change a state in the system under control, the controller responds by issuing the proper commands. Per the example above, the state effects model for the power switch command would show that the power switch command effects the power switch. The power switch state controller would use this model to determine that it needs to issue the power switch command to change the state of the power switch.

Refinement of Projections to Incorporate Execution Engine Behavior

In addition to the state effects model, projections can reflect the behavior of the execution engine. The planning engine consults state variables to compute projections, which in turn can consult their achievers in the execution engine. The achievers contain a model of their behavior when they execute goals, and can use this along with the state effects model to refine a state projection that would otherwise be based purely on the state effects model. This refined state projection reflects not only the state constraint, and the state effects model, but also what the achiever does to the physical state when the acheiver executes the state constraint. This refined state projection should be a subset of the original state constraint for the plan to be valid, otherwise the constraint is noted as unachievable, and an alternative plan is considered. In this way the execution engine models its capability to execute constraints and provides this information to the planner. By using the modeling information, the planner can insure that the plans are consistent with the capabilities of the execution engine.

State Uncertainty

MDS handles inaccuracies in modeling by explicitly representing uncertainty in state estimates produced during execution. A state variable s history contains uncertainty associated with a estimate for each point in time. An estimator always updates the history with a measure of uncertainty.

A state s controller has access to this uncertainty, and must control to bounds specified in the planned state constraints. The bounds specified in planned state constraints are checked by the controller against the current estimated state and its uncertainty. The controller takes actions to ensure that the bounds are met given the uncertainty in the state estimate. For example, if a control constraint is to keep an actuator s position within a deadband, and the current uncertainty is expressed as a range, then the controller must actually control to a narrower range to account for the uncertainty in the knowledge of the position.

If a given control constraint requires a certain state uncertainty, then the planner s elaboration of the control constraint includes a constraint on the uncertainty of the knowledge of the state. This sort of uncertainty constraint is called a knowledge goal, and is executed by the estimator that is responsible for estimating the state. It is the responsibility of the estimator to achieve the planned level of uncertainty during execution. This is monitored by

the execution engine, which will flag deviations to the planner if the knowledge constraint is not being met.

Timing Uncertainty

MDS deals with uncertainty in the timing of plans using flexible time. Each time point in the planned goal network has a range of possible times, to either allow for the uncertainty in execution times of constraints or to accommodate flexibility in the execution system. Uncertain temporal intervals are appropriately labeled. The state projections produced by the planner take into account the range of times. The planner assumes times that would produce the worst-case state projections. The worst-case projections are bounded by the operators by imposing deadlines on activities in the form of temporal constraints. An example is a rover traverse followed by a fixed time Earth communication window. The energy use during the traverse is limited by the deadline imposed by the communication window, when the rover must be immobile. If the traverse is not completed by the communication window, the traverse constraint ends early, and is replanned to start after the communication window.

Determining that a plan with flexible time will execute successfully boils down to determining dynamic controllability of the temporal constraint network [6]. How this execution actually ensues is called timepoint firing, and is described in detail in [12].

State Analysis

State Analysis [2] improves on the current state-of-the-practice by producing requirements on system and software design in the form of explicit models of system behavior, and by defining a state-based architecture for the control system. It provides a common language for systems and software engineers to communicate, and thus bridges the traditional gap between software requirements and software implementation.

State Analysis provides a uniform, methodical, and rigorous approach for:

(1) discovering, characterizing, representing, and documenting the states of a system under control,
(2) modeling the behavior of states and relationships among them, including information about hardware interfaces, operations, and achiever behavior,
(3) capturing the mission objectives in detailed scenarios motivated by operator intent,
(4) keeping track of system constraints and operating rules, and
(5) describing the methods by which objectives will be achieved.

The state analysis methodology recognizes the need for specifying execution behavior and planning specifications in terms of common models. State effects models are developed in a spiral process of state discovery until all of the states of the system to be controlled are known and their physical models are well understood. The execution and planning engine software is then specified in terms of these physical models. This includes specifications for estimation and control algorithms, elaborations, constraint semantics, projection algorithms, and the other information exchanged between the execution and planning engine as discussed in the previous sections.

Conclusion

We have discussed the MDS control architecture, modeling framework, and systems engineering methodology for developing state-based models of real-world behavior, effects, and execution engine behavior, which are used to inform both planning and execution. These models provide a basis for ensuring the design-time consistency of assumptions in the execution engine and planner, and are the basis for run-time communications and coordination between planners, schedulers, and execution engines. The modeling framework has been designed to be an open architecture for applying various formalisms and algorithms for spacecraft operations planning and execution.

In MDS, state-based models provide a basis for communication between planners, schedulers, and the engines that execute scheduled plans as follows:

(1) MDS has clearly defined roles in the architecture for planning and execution
(2) MDS has defined semantics of execution in terms of state-histories that are represented using state constraints (goals). The MDS architecture has well-defined interfaces for exchanging information between the execution and planning engines, including methods on achievers called by the planner.
(3) MDS handles inaccuracies in modeling by explicitly representing uncertainty in state estimates produced during execution, using this uncertainty in controllers, planning for uncertainty in knowledge goals, and monitoring and enforcing the planned level of uncertainty during execution using estimators.
(4) MDS deals with uncertainty in activity duration and event timing using flexible time, uncertain time intervals, and worst-case state predictions limited by temporal constraints that impose deadlines.
(5) MDS State Analysis is a systems engineering methodology that with the MDS software frameworks bridges the gap between how execution is treated in the planning process, and what happens when the resulting plan is actually executed.

Acknowledgments

The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. We wish to thank the rest of the Mission Data System development team, and the Mars Science Laboratory mission personnel who have participated in the maturation of MDS.

References

[1] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, Software architecture themes in JPL's Mission Data System, *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, number AIAA-99-4553, 1999.

[2] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada, Engineering Complex Embedded Systems with State Analysis and the Mission Data System, *Proceedings of the 1st AIAA Intelligent Systems Technical Conference*, number AIAA-2004-6518, 2004.

[3] A. Barrett, R. Knight, R. Morris, and R. Rasmussen, Mission Planning and Execution Within the Mission Data System, *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004.

[4] D. Dvorak, R. Rasmussen, and T. Starbird, State Knowledge Representation in the Mission Data System, *Proceedings of the IEEE Aerospace Conference*, 2002.

[5] B.C. Williams, M. Ingham, S. Chung, and P. Elliott, Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers, *Proceedings of the IEEE*, 91(1):212-237, 2003.

[6] 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, 2001.

[7] A. Meiri, R. Dechter, and J. Pearl, Temporal Constraint Networks, *Artificial Intelligence*, 49:61--95, 1991.

[8] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee, "Iterative Repair Planning for Spacecraft Operations in the ASPEN System," *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS 1999)*, Noordwijk, The Netherlands, June 1999.

[9] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using Iterative Repair to Improve Responsiveness of Planning and Scheduling," *International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, Breckenridge, CO, April 2000.

[10] A. K. Jonsson, P. H. Morris, N. Muscettola, K. Rajan, and B. Smith, "Planning in Interplanetary Space: Theory and Practice," *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, 177-186.

[11] N. Muscettola, P. Morris, B. Pell, and B. Smith, Issues in temporal reasoning for autonomous control systems, In F. Anger, editor, Working Notes from the *AAAI workshop on Spatial and Temporal Reasoning*, 1997.

[12] R. Knight, Evaporating tasks during execution of dynamically controllable networks, *AAAI Workshop on Plan Execution*, 2005.

Safe Execution of Temporally Flexible Plans for Bipedal Walking Devices

Andreas Hofmann and Brian Williams

Computer Science and AI Lab, Massachusetts Institute of Technology
Author(s) Address(es) Go(es) Here in 9 Point Times Roman
Author(s) Address(es) Go(es) Here in 9 Point Times Roman
hofma@csail.mit.edu, williams@mit.edu

Abstract

Plans with temporal flexibility have been used to allow discrete systems to adapt to disturbances that occur while the plan is being executed. To control more complex devices, such as bipedal walking machines, we must extend this execution paradigm to the control of hybrid (discrete/continuous) systems. Systems of this type are difficult to control for two reasons: 1) their high dimensionality and nonlinearity make control complex, even under nominal circumstances; and 2), operation of such systems in unstructured environments requires robustness to significant disturbances.

We introduce a novel approach to hybrid temporally flexible plan execution that achieves robustness by transforming the high-dimensional system into a set of low-dimensional weakly-coupled systems. This allows us to apply dynamic controllability concepts previously applied to discrete systems. We accomplish this decoupling using three components: 1) a feedback linearizing controller which provides basic decoupling, 2) a hybrid plan dispatcher which utilizes plan flexibility to adjust control settings for individual decoupled variables, and 3) plan compilation which computes bounds for the dispatcher's adjustments to control settings that satisfy plan requirements. We show the interaction of these components in control of a bipedal walking machine.

Introduction

Effective use of autonomous robots in unstructured, human environments requires that robots: 1) have sufficient autonomy to perform useful tasks independently, 2) have sufficient size, strength, and speed to accomplish such tasks in a timely manner, and 3) operate safely. A particularly challenging example of such a robot is a bipedal walking machine (Fig. 1a).

An example task for such a system is to walk to a soccer ball and kick it. If the system encounters a significant force disturbance while performing this task, it will have to compensate by changing its stepping pattern, or leaning the body as shown in Fig. 1b. The disturbance may cause a delay (allowing another player to kick the ball). At a more subtle level, it may interfere with movement synchronization; a lateral disturbance may cause synchronization problems with forward motion and stepping.

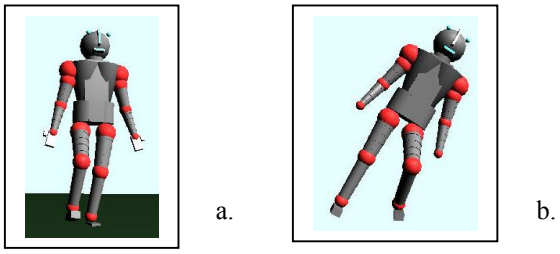


Fig. 1 – a. Biped, b. reaction to disturbance

This type of problem presents a number of technical challenges. First, movement task goals and safety requirements, which are most naturally specified in terms of state-space region, and temporal range constraints, must be translated into control actions that achieve these goals. This is difficult because the system is highly nonlinear, has high dimensionality, and has input constraints that limit controllability. Second, the dual goals of timely task execution and safety are often in conflict, and must be judiciously balanced. Finally, the system must be robust to significant disturbances.

Dynamic optimization techniques have been used to generate humanoid motion plans for animation applications [Popovic and Witkin, 1999]. However, these methods produce very detailed and inflexible reference trajectories, and are therefore not robust to disturbances. Robustness requires plan flexibility.

A powerful set of methods has been developed for discrete systems, for safe execution of temporally flexible plans [Morris, 2001]. These methods guarantee successful plan execution, as long as temporal uncertainty of activities is appropriately bounded. For example, in Fig. 2a, a car and sailboat leave Boston for P-Town at the same time. Duration of the sail is uncertain, but the uncertainty is bounded (between 6 and 12 hours). Likewise, the drive is between 3 and 4 hours. Synchronization in P-Town is assured because the car can wait there indefinitely for the sail boat.

In such systems, state is represented by logical variables. Constraints include logical constraints on these variables, and continuous temporal constraints. Executives for such

systems operate by scheduling start times of activities dynamically. They assume that at the end of an activity (like drive), state will not change (the car will remain in P-Town).

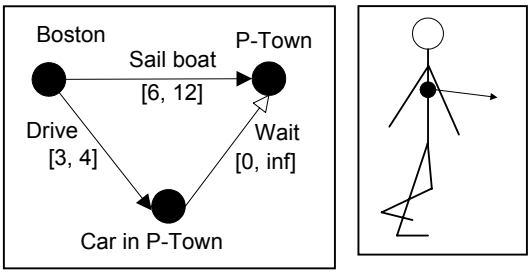


Fig. 2 – a. Dynamically controllable plan for discrete system (left), b. Underactuated dynamic system cannot always wait (right)

This approach is appropriate for many kinds of applications, but it does not work for agile underactuated dynamic systems like bipeds, where movement is fast and controllability is limited. In particular, the lack of equilibrium points in such systems means that state is constantly changing, and the executive cannot assume the system will wait in a particular state at the end of an activity. For example, in dynamic walking, it is not possible to instantly stop forward movement in the middle of a step, as shown in Fig. 2b. The stepping foot must move out in front, or the biped will fall.

Systems of this type include continuous, as well as discrete state variables, so we refer to such systems as *hybrid*. Continuous constraints on the continuous state variables express the system's dynamics, and specify valid regions of operation. An executive for such a system must take into account the system's dynamics and controllability limits. Rather than directly scheduling activity start times, the dispatcher controls timing indirectly. By adjusting control parameters appropriately, the dispatcher guides trajectories of interest into goal state-space regions at the right time.

Approach

We allow specification of task goals in terms of state-space region and temporal range constraints, as shown in Fig. 3. Foot placement constraints define qualitative poses such as double support, or left single support, but the details of the joint positions and trajectories are omitted. A state-space goal region for the forward position of the center of mass at the end of the gait sequence defines the task goal. A temporal range constraint specifies task completion time requirements.

To translate these specifications into control actions, we use a mixed on-line/off-line approach. The off-line component is a compiler that synthesizes a set of adaptive controllers. The on-line component is a hybrid dispatcher that efficiently adapts control settings in response to disturbances.

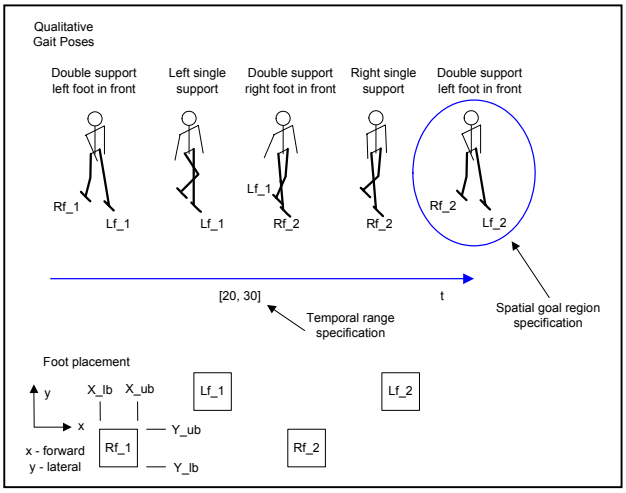


Fig. 3 – Task Specification

A key insight in our approach is to use a feedback linearizing multivariable controller [Hofmann, 2004] to transform the highly nonlinear, tightly coupled biped system into a loosely coupled set of linear 2nd-order single-input single-output (SISO) systems. This *SISO abstraction* greatly simplifies the task of the hybrid dispatcher, allowing it to focus on a few quantities of interest (center of mass position), which appear to behave in an independent and linear manner, rather than on the details of joint movement. In particular, the SISO abstraction allows the dispatcher to control the system by adjusting a small set of linear control law parameters for each quantity of interest. The multivariable controller then takes care of the details of computing torques that achieve the desired motion.

Qualitative State Plan and Plant

State-space and temporal constraints that specify goal regions are assembled into a *qualitative state plan*, which represents the desired state evolution of the *plant* (the system being controlled). The state plan is qualitative in that it specifies behavior in terms of state regions with common characteristics, rather than with specific states. The problem, given a state plan and a plant, is to generate a sequence of control actions that move the plant to a state consistent with that required by the plan, while avoiding unsafe regions.

Plant

The plant is represented by a set of dynamic equations that specify state evolution as a function of inputs. We define a plant by the tuple $\langle \mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{h}, \mathbf{f} \rangle$ where $\mathbf{x}, \mathbf{y}, \mathbf{u}$ is the set of state, output (controlled), and input variables, respectively,

$\mathbf{h}$ is a set of algebraic equations that relate outputs to inputs and state, and $\mathbf{f}$ is a set of 1st-order differential equations that describe state evolution. The input vector, $\mathbf{u}$, includes control inputs, such as joint torques, and also environment inputs (ground contact forces). The state vector, $\mathbf{x}$, includes continuous variables, such as joint positions and velocities, and discrete variables, to indicate the presence of environment forces. The output vector, $\mathbf{y}$, contains the variables to be controlled, such as forward and lateral center of mass (CM) position and velocity. Our simulated walking biped plant has 18 degrees of freedom, and is highly nonlinear [Hofmann et al., 2002]. We assume that plant state is available from sensors, or can easily be estimated.

Qualitative State Plan

A qualitative state plan specifies state evolution using sequences of *activities* as shown in Fig. N. Each sequence (each row in this diagram) specifies behavior for a particular quantity. Two types of quantities, *controlled quantities*, and *input quantities*, can be specified. Controlled quantities are position/velocity pairs corresponding to elements of $\mathbf{y}$. Input quantities are scalar functions, $g_i(y_i, \dot{y}_i)$, that represent control laws for controlled quantities, and thus, correspond to elements of $\mathbf{u}$. In Fig. 4, forward and lateral CM are examples of controlled quantities, and forward and lateral CP (center of pressure) are examples of input quantities.

Each activity in a sequence is part of a control epoch (column in Fig. 4). The control epochs shown in Fig. 4 correspond to the qualitative poses shown in Fig. 3. Thus, epoch 1 represents double support with the left foot in front, epoch 2, left single support, epoch 3, double support with the right foot in front, and epoch 4, right single support. Epoch 5 repeats epoch 1, but is one gait cycle forward. Vertical bars in Fig. 4 between rows represent synchronization constraints, so that all quantities advance to the next control epoch at the same time.

Each activity may have a temporal duration range constraint, indicated by [lb, ub]. This specifies the lower bound (lb) and upper bound (ub) on the activity's duration. In addition, range constraints on acceptable initial and goal regions for quantities may be specified. For controlled quantities, regions are specified using rectangles in position/velocity phase space. For input quantities, regions are specified using scalar ranges. Note that the duration and region constraints are optional. In fact, these constraints are omitted for many of the activities in the state plan. In Fig. 4, we care about the initial and final region of the CM, but not about the details in between. Thus, for the CM quantities, an initial region for epoch 1 may be specified, and a final region for epoch 5, but the intervening regions may be left unspecified. Similarly, we care that the gait cycle be completed within time range [t_lb, t_ub], but not about the detailed durations of each activity, so these are unspecified as well.

Each activity may also have a *tube* constraint specifying a required region for the associated quantity over the entire duration of the activity. Such constraints are useful, for example, for limiting the range of forward and lateral CP, over the entire course of a qualitative pose. This is important because CP is an input quantity that is limited by foot placement (see also Fig. N-1). For example, in epoch 2 (left single support), the CP is restricted to the support region under the left foot.

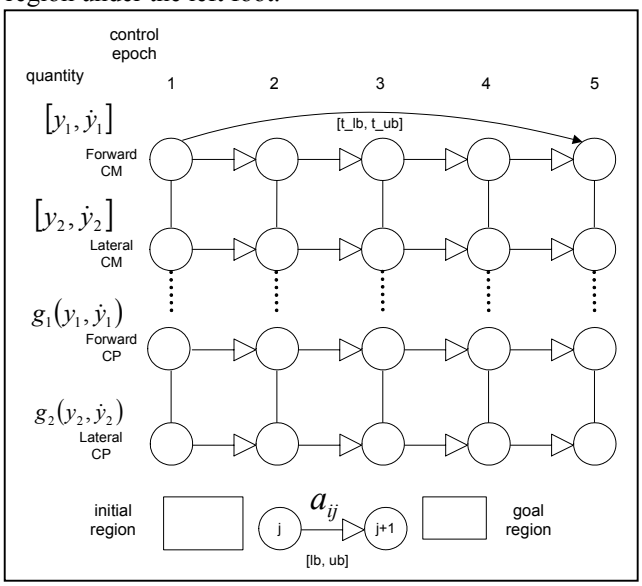


Fig. 4 – Qualitative State Plan

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.

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. Use of such unary constraints with constant bounds results in rectangular regions in position/velocity state-space. This implies more conservative bounds than would be possible if the constraints were multivariable and nonlinear, but it also greatly simplifies planning. 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 $[lb, ub]$ specifying the permissible range of activity duration, and $A_{parallel}$ is a set of activities that must finish simultaneously with the current one. $A_{parallel}$ provides the capability to synchronize multiple concurrent activity sequences (vertical bars in Fig. N). For example, for a biped, movement of the stepping foot must be synchronized with forward movement of the center of mass.

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 $y(ts), \dot{y}(ts)$ satisfy R_{init}, $y(tf), \dot{y}(tf)$ satisfy R_{goal}, and $y(t), \dot{y}(t)$ 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 $A_{parallel}$, $tf(i,j) = tf(k,j)$. Fig. 5 shows lateral CM and CP trajectories for a nominal execution of the state plan shown in Fig. 4.

Plan Compiler

A qualitative state plan cannot be executed directly because it is missing control information, and because much of the trajectory information is still under-specified. The plan compiler adds this missing information to the qualitative state plan, producing a qualitative control plan. A qualitative control plan consists of the activities from the state plan, with the two following additions: 1) control 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 information is of the form $\langle k_{1\min}, k_{1\max}, k_{2\min}, k_{2\max} \rangle$; bounds on control parameters for the linear control laws used in the SISO abstraction. Thus, the qualitative control plan contains all the information needed to control the SISO system, and to monitor its status with respect to region and temporal bounds.

In computing bounds on the control parameters, the compiler is, in effect, performing an adaptive controller synthesis. The hybrid dispatcher utilizes the flexibility of the control parameter ranges to adapt control settings as needed. In order to maximize robustness to disturbances, the compiler attempts to maximize the size of initial regions, and tubes, minimize the size of goal regions, and maximize controllable temporal activity duration ranges. Maximizing initial regions and minimizing goal regions results in a contraction; the family of trajectories in the tube "contract" to each other as time advances. Maximizing controllable temporal range makes synchronization with other controlled quantities easier.

In generating trajectories, the compiler must take into account dynamics. We want fast performance, but due to the underactuated nature of the system, this leads to a reduction in controllability. Thus, future consequences of current actions become increasingly important as speed of movement is increased. The compiler must take into account future epochs in order to plan feasible trajectories. In this respect, the plan compiler is similar to receding horizon model-predictive controllers [ref. From Thomas' paper].

The compiler proceeds in two steps. First, it computes a nominal trajectory that reaches the goal state from the initial state, and that satisfies all state-space and temporal constraints. Second, to provide robustness to disturbances, the compiler expands the nominal trajectory, creating regions and tubes not specified explicitly in the qualitative state plan, attempting to maximize initial regions, minimize goal regions, and maximize controllable temporal range. For both steps, we utilize an SQP (Sequential Quadratic Programming) optimizer, and formulate the problem as an NLP (Nonlinear Program). These two steps are now described in detail.

Nominal Trajectory Computation

The NLP formulation for the nominal trajectory computation is as follows. For each activity, a_{ij}, associated with a controlled quantity, parameters to optimize are $\langle y_{init}, \dot{y}_{init}, y_{goal}, \dot{y}_{goal}, t_{goal}, k_1, k_2 \rangle$, where $y_{init}, \dot{y}_{init}$ are the initial state of the quantity associated with the activity, $y_{goal}, \dot{y}_{goal}$ are the quantity's final state, t_{goal} is the duration of the activity, and k_1, k_2 are parameters for the linear control law that achieves the trajectory. Constraints on these parameters are as follows. The trajectory is defined by the analytic solution to the linear 2nd-order differential equation formed by applying the linear control law to the SISO abstraction. This yields an equality constraint that relates goal to initial state, control parameters, and duration:

$$\begin{aligned} y_{goal} &= f_1(y_{init}, \dot{y}_{init}, k_1, k_2, t_{goal}) \\ \dot{y}_{goal} &= f_2(y_{init}, \dot{y}_{init}, k_1, k_2, t_{goal}) \end{aligned} \quad (1)$$

Continuity from one epoch to the next is expressed as

$$\begin{aligned} y_{goal}(i,j) &= y_{init}(i,j+1) \\ \dot{y}_{goal}(i,j) &= \dot{y}_{init}(i,j+1) \end{aligned} \quad (2)$$

Inequality constraints for R_{init}, R_{tube}, R_{goal}, and $R_{duration}$ are as described previously. Synchronization constraints across quantities are expressed as

$$\forall i,j : t_{goal}(i,j) = t_{goal}(i+1,j) \quad (3)$$

Finally, the temporal constraint on overall state plan execution time is given by

$$\forall i,j : t_{lb} \le \sum_i t_{goal}(i,j) \le t_{ub} \quad (4)$$

The cost function includes terms that maximize the distance to the R_{tube} boundaries.

An example of a nominal trajectory computed in this way is shown in Fig. 5.

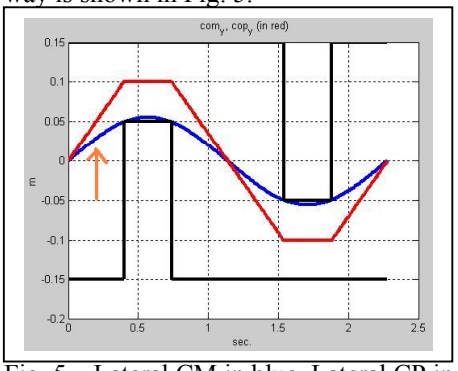


Fig. 5 – Lateral CM in blue, Lateral CP in red

Qualitative Control Plan Computation

The NLP formulation for the control plan computation is as follows. For each activity, a_{ij}, associated with a controlled quantity, parameters to optimize are: $\langle y_{init\,\min}, \dot{y}_{init\,\min}, y_{init\,\max}, \dot{y}_{init\,\max} \rangle$ (parameters of R_{init}), $\langle y_{goal\,\min}, y_{goal\,\min}, y_{goal\,\max}, y_{goal\,\max} \rangle$ (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. Note that these are similar to the ones in the nominal computation, except that they are now ranges rather than nominal values.

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. For both these trajectories, it is assumed that velocity does not change sign (position is monotonically increasing or decreasing).

Consider the initial and goal regions shown in Fig. 6. For the GFT, the worst-case starting point in the initial region is point B, which corresponds to $y_{init\,\min}, \dot{y}_{init\,\min}$. This represents the slowest possible start. By accelerating as quickly as possible, the GFT reaches point D in the goal region, which corresponds to $y_{goal\,\min}, \dot{y}_{goal\,\max}$. This represents the fastest finish point in the goal region. For the GST, the worst-case starting point in the initial region is point A, which corresponds to $y_{init\,\max}, \dot{y}_{init\,\max}$. This represents the fastest possible start. By accelerating as slowly as possible, the GST reaches p;oint c in the goal region, which corresponds to $y_{goal\,\max}, \dot{y}_{goal\,\min}$. This represents 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

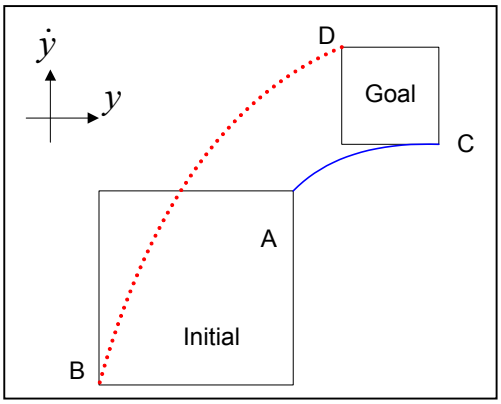


Fig. 6 – GFT (dotted) and GST (solid)

beginning from anywhere in the initial region can be guaranteed to be in the goal region. The existence of the 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 very simple control law. Suppose that the only control input allowed is a single acceleration spike (of an appropriate size). If this spike is applied at the beginning of the trajectory, then maximum velocity is reached immediately. This corresponds to the GFT, as shown in Fig. 7. If this spike is applied at the end, then the trajectory will progress at minimum velocity until the end. This corresponds to the GST.

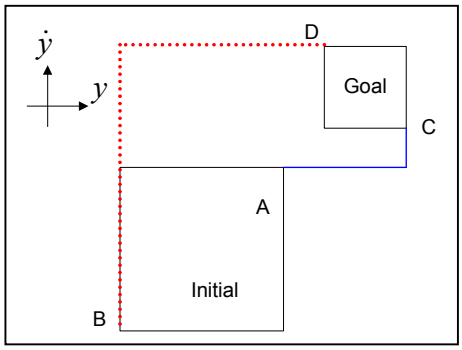


Fig. 7 - GFT (dotted) and GST (solid) for acceleration spike control laws

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

$$\begin{aligned} y_{init\,\min} &< y_{init\,\max} \\ \dot{y}_{init\,\min} &< \dot{y}_{init\,\max} \\ y_{goal\,\min} &< y_{goal\,\max} \\ \dot{y}_{goal\,\min} &< \dot{y}_{goal\,\max} \end{aligned} \quad (5)$$

To guarantee contraction from one control epoch to the next, the goal region of an activity must fit inside the initial region of its successor.

$$\begin{aligned} y_{goal\,\min}(i,j) &\geq y_{init\,\min}(i,j+1) \\ \dot{y}_{goal\,\min}(i,j) &\geq \dot{y}_{init\,\min}(i,j+1) \\ y_{goal\,\max}(i,j) &\leq y_{init\,\max}(i,j+1) \\ \dot{y}_{goal\,\max}(i,j) &\leq \dot{y}_{init\,\max}(i,j+1) \end{aligned} \quad (6)$$

Constraints representing the GFT are expressed as

$$\begin{aligned} y_{goal\,\min} &= f_1(y_{init\,\min}, \dot{y}_{init\,\min}, k_{1\max}, k_{2\max}, t_{\min}) \\ \dot{y}_{goal\,\max} &= f_2(y_{init\,\min}, \dot{y}_{init\,\min}, k_{1\max}, k_{2\max}, t_{\min}) \end{aligned} \quad (7)$$

Constraints representing the GST are expressed as

$$\begin{aligned} y_{goal\,\max} &= f_1(y_{init\,\max}, \dot{y}_{init\,\max}, k_{1\min}, k_{2\min}, t_{\max}) \\ \dot{y}_{goal\,\max} &= f_2(y_{init\,\max}, \dot{y}_{init\,\max}, k_{1\min}, k_{2\min}, t_{\max}) \end{aligned} \quad (8)$$

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

$$\begin{aligned} &(t_{\min}(1,j) \leq t_{trans\,\min}(j) \leq t_{trans\,\max}(j) \leq t_{\max}(1,j)) \wedge \ldots \\ &(t_{\min}(i,j) \leq t_{trans\,\min}(j) \leq t_{trans\,\max}(j) \leq t_{\max}(i,j)) \end{aligned}$$

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, minimizes goal region size, and maximizes controllable temporal range.

Fig. 8 shows regions for lateral CM computed in this way.

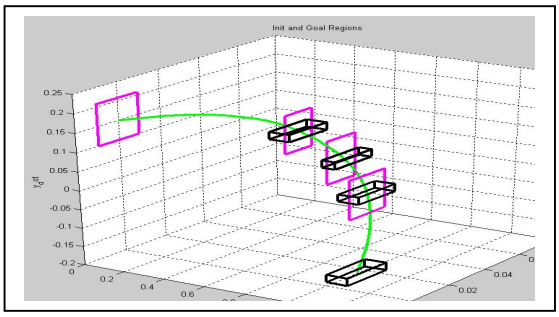


Fig. 8 – Lateral CM regions ($y, \dot{y}$ vs. time)

Temporal Uncertainty

If controllability is very limited, it may be difficult to achieve a large enough initial region. This situation can be improved by relaxing the constraint $t_{GFT} < t_{GST}$, as shown in Fig. 9.

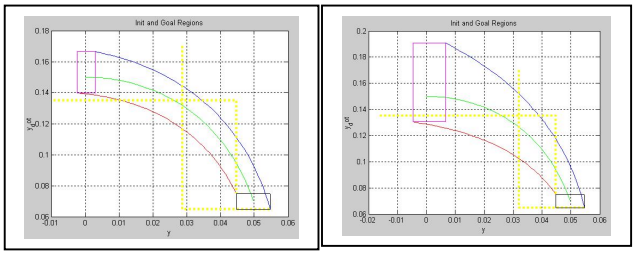


Fig. 9 – a. t(GFT) = t(GST), left, b. t(GFT) > t(GST), right

The intuitive explanation for why this happens is that as GFT is allowed to take longer, the B point of the initial region is allowed to stretch to the left (position decreases). Similarly, if GST is allowed to complete faster, point A moves to the right (position increases).

Unfortunately, relaxing this constraint means that we lose temporal controllability; the time of arrival in the goal region can no longer be precisely controlled. However, the uncertainty on arrival time is bounded by $[t_{GST}, t_{GFT}]$.

This may still be ok if controllability of other controlled quantities is strong enough to compensate for uncertainty. When this is the case, the system can be considered to be dynamically controllable [Morris, 2001]. This situation can be represented using an STNU (simple temporal network with uncertainty), as shown in Fig. 10.

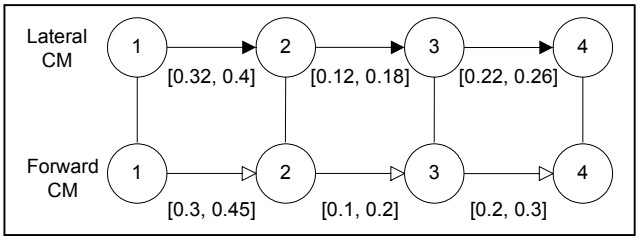


Fig. 10 – STNU for lateral and forward CM

In this STNU, the temporal durations for lateral CM are uncertain but bounded. The temporal durations for forward CM are certain, and they are large enough to compensate for the uncertainty in the lateral CM. Thus, transition synchronization can still be guaranteed, though this will require adjustment by the hybrid dispatcher once the uncertainty is resolved.

Hybrid Dispatcher

The hybrid dispatcher executes the qualitative control plan. It acts as an adaptive controller, adjusting control parameters within the limits specified in the control plan, 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. Note that unlike dispatchers for discrete systems [Morris, 2001], the hybrid dispatcher is not able to directly schedule start times for activities. Rather, it indirectly controls timing by adjusting control parameters.

At the beginning of a new control epoch, j, the dispatcher computes a target transition time, $t_{trans}(j)$. When there is no temporal uncertainty, it chooses a time in the range $[t_{trans\min}(j), t_{trans\max}(j)]$. When there is uncertainty in one of the controlled quantities, the transition time is determined when the uncertainty for this controlled quantity is resolved.

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. If all trajectories execute successfully, then when all controlled quantities are in their respective goal regions, the dispatcher transitions to the next control epoch.

Results and Discussion

Our tests on the simulated biped show that a small lateral CM disturbance (100 N for 0.01 sec) can be handled by the multivariable controller without any immediate action by the dispatcher. A larger (250 N) disturbance requires gain adjustment by the dispatcher. A disturbance of 300 N is too large for the dispatcher to handle by itself; a new dispatchable plan is required.

These tests demonstrate compliance to disturbances at three levels: 1) use of low-gain control, 2) flexibility in the dispatchable state plan allowing for adjustments by the hybrid dispatcher, and 3) fast reactive planning. This allows for integrated handling of disturbances of varying degrees of severity.

References

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

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

[Hofmann et al., 2002] A. Hofmann, M. Popovic, H. Herr. Humanoid Standing Control: Learning from Human Demonstration. *Journal of Automatic Control*, 12(1), 16–22

[Hofmann et al., 2004] 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

[Morris et al., 2001] 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).

[Muscettola et al., 1998] 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

[Popovic et al., 1999] Z. Popovic and A. Witkin. Physically based motion transformation. Siggraph 1999

[Popovic et al., 2004] M. Popovic, A. Hofmann, H. Herr. Zero spin angular momentum control: definition and applicability. (Humanoids). Los Angeles (CA, USA).

[Slotine and Li, 1991] J. Slotine and W. Li. Applied Nonlinear Control. Ch. 6, Prentice Hall, NJ, USA

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

Robot Actions Planning and Execution Control for Autonomous Exploration Rovers

Matthieu Gallien **Félix Ingrand** **Solange Lemai**
LAAS-CNRS*
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4, France

Abstract

To achieve the ever increasing demand for science returns, extraterrestrial exploration rovers require more autonomy to successfully perform their missions. Indeed, the communication delays are such that teleoperation is unrealistic. Although the current rovers (such as MER) demonstrate a limited navigation autonomy, and mostly rely on ground mission planning, the next generation (e.g. NASA Mars Science Laboratory and ESA Exomars) aims at "beyond the field of view" autonomous navigation. Other exploration missions which cannot rely on human teleprogramming, will even require activity planning, repair and replanning to be made onboard.

In this paper, we propose and give experimental results of an original approach for temporal planning and execution control, including plan repair and replanning, fully integrated onboard a robot performing rover exploration like missions. Our claim is twofold. First these planning/plan repair methods and techniques are now mature enough to be considered to solve real world problems. Second they can be integrated in existing architectures and used onboard a fully operational robot, with currently available hardware.

Introduction

Extraterrestrial exploration rovers have an increasing need for high level autonomy. If one compares the navigation capabilities of Sejourner and MER, one can already see that some modest, yet real, navigation autonomy has been introduced. Moreover, higher science return, and the communication latency of deep space mission[1] are pushing to get some of the traditionally high level activities planning performed on board. For example in the MER mission, an automated planning system (MapGen (Ai-Chang *et al.* 2003)) was used on the ground to produce the daily activities for Spirit and Opportunity. The operational results of MapGen are quite encouraging, as it allowed a 25% increase in science returns compared to a human generated plan (Rajan 2004). As of today, the ESA Exomars project (part of Aurora) aims at having the rover navigating over its "field of view", in one day, with navigation decisions taken on board. NASA MSL will also push the autonomy cursor further than for MER. Last, the "Human on Mars" goal will require the deployment of a large number of autonomous systems to "prepare" and study the planet before a human can set foot on it. The "future" of exploration rovers and probes clearly lies in an increased autonomy addressing the problems of action planning, and plan execution control.

Meanwhile, automated actions planning has made some progress since the early days of Shakey and STRIPS. There are now planners able to take into account time, resources, constraints and to solve real world problems. Still, planning is only one aspect of the problem. Plans, even flexible or contingent one, are bound to fail. Plan repair and replanning are thus needed to ensure that the system is able to recover from unexpected plan execution failure.

In this paper we present IxTeT, a temporal planner which includes an execution controller, as well as some plan repair and replanning capabilities. The resulting system has been integrated in the LAAS architecture(Alami *et al.* 1998) and implemented onboard Dala, our iRobot ATRV Robot. Such a planner is in charge of producing plans composed of actions such as move, science activities (moving and operating instruments), communication with earth and an orbiter or a lander, while managing resources (power, memory, etc) and temporal constraints (communication visibility windows, rendezvous, etc).

Still, the execution of action as simple navigation task such as a move in an unknown environment implies complex processes (Lacroix *et al.* 2003; Goldberg, Maimone, & Matthies 2002): localization, map building, motion generation, etc. The LAAS architecture (Alami *et al.* 1998) and its associated tools provide a support in order to design and integrate such a complete autonomous system.

Fig. 1 presents the architecture implemented for the experiment on Dala. The *functional level* includes all

*List of authors in alphabetical order. Part of this work has been funded by a grant from the ESF (European Social Fund), and is partially supported by CNES and Astrium

[1]Unlike most navigations on the ground, a comet landing phase can hardly be suspended.

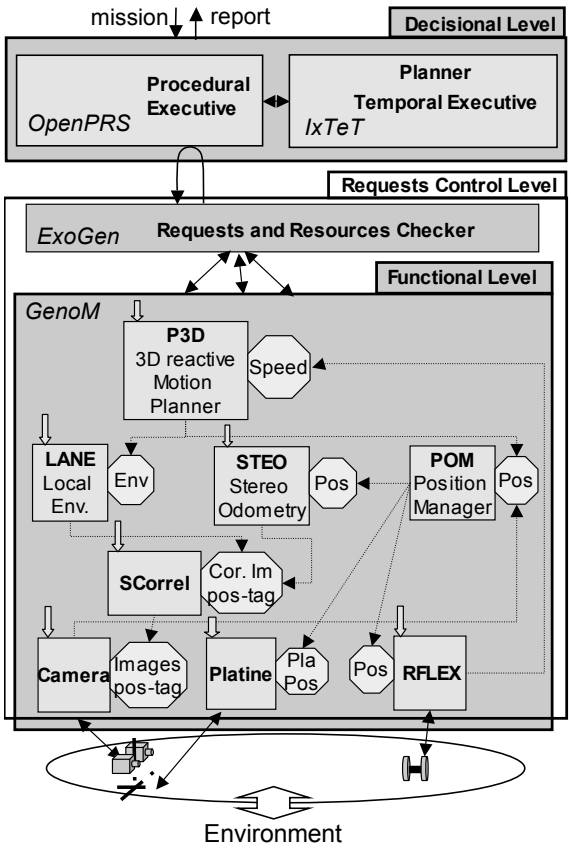


Figure 1: The LAAS architecture on Dala, an iRobot ATRV.

the basic built-in robot action and perception capabilities, encapsulated into controllable communicating modules. These modules are activated by requests, send reports upon completion and export data. Tor example, the POM module computes the best position estimate from standard (RFLEX) and visual (STEO) odometry, while the wheels are controlled by RFLEX according to the reference velocity produced by the reactive motion planner (P3D). The *requests control level* filters the requests according to the current state of the system and a formal model of allowed and forbidden states (see (Py & Ingrand 2004)).

IxTeT has been integrated in the *decisional level* and interacts with the user and the functional level through a procedural executive (OpenPRS). First, IxTeT produces a plan to achieve a set of goals provided by the user. The plan execution is controlled by both procedural and temporal executives as follows. The temporal executive decides when to start or stop an action in the plan and handles plan adaptations. OpenPRS expands and refines the action into commands to the functional level, monitors its execution and can recover from specific failures. It finally reports to IxTeT upon the action completion.

The paper is organized as follows. The first section presents the core planner used as well as the 3DC+ algorithm. The following section focuses on the execution control part of the system as well as the repair and replanning mechanism. Then we present the experimentation (rover exploration planning), and the result of the integration of IxTeT on board the Dala robot. Last, we compare this work with similar works and present conclusions and possible prospectives.

The planner

The planner in IxTeT is a lifted POCL temporal planner based on CSPs (Laborie & Ghallab 1995). Its temporal representation describes the world as a set of *attributes*: logical attributes (e.g. robot_position($?r$)), which are multi-valued functions of time, and resource attributes (e.g. battery_level()) for which one can specify borrowings, consumptions or productions. We note $LgcA$ and $RscA$, respectively the sets of logical and resource attributes. $LgcA_g$ and $RscA_g$ designate the sets of all possible instantiations of these attributes.

The evolution of a logical attribute value is represented through the proposition *hold*, which asserts the persistence of a value over a time interval, and the proposition *event*, which states an instantaneous change of value. The propositions *use*, *consume* and *produce* respectively specify over an interval the borrowing, the consumption or the production at a given instant of a resource quantity.

```
task MOVE(?initL,?endL)(st,et){
  ?initL,?endL in LOCATIONS;
  event(ROBOT_POS():(?initL,IDLE_POS),st);
  hold(ROBOT_POS():IDLE_POS,(st,et));
  event(ROBOT_POS():(IDLE_POS,?endL),et);
  event(ROBOT_STATUS():(STILL,MOVING),st);
  hold(ROBOT_STATUS():MOVING,(st,et));
  event(ROBOT_STATUS():(MOVING,STILL),et);
  hold(PTU_POS():FORWARD,(st,et));
```

```
  variable ?di,?du,?dist;
  variable ?duration;
  distance(?initL,?endL,?di);
  distance_uncertainty(?du);
  ?dist = ?di * ?du;
  speed(?s);
  ?dist = ?s * ?duration;
  contingent ?duration = et − st;
}latePreemptive
```

Figure 2: Example of move action model.

```
task MOVE_PTU(?initL,?endL)(st,et){
  timepoint end_heat;
  ?initL,?endL in PTU_POSITIONS;
  hold(ROBOT_STATUS():STILL,(end_heat, st));
  event(PTU_STATUS():(COLD, HEAT),st);
  hold(PTU_STATUS():HEAT,(st,end_heat));
  event(PTU_STATUS():(HEAT,MOVING),end_heat);
  hold(PTU_STATUS():MOVING,(end_heat,et));
  event(PTU_STATUS():(MOVING,COLD),et);
  hold(PTU_INIT():TRUE,(st,et));
  hold(PTU_POS():?initL,(st,end_heat));
  event(PTU_POS():(?initL,PTU_POS_IDLE),end_heat);
  hold(PTU_POS():PTU_POS_IDLE,(end_heat,et));
  event(PTU_POS():(PTU_POS_IDLE,?endL),et);
  (end_heat − st) in [10,12];
  contingent (et − st) in [16,20];
}latePreemptive
```

Figure 3: Example of move_ptu action model.

As shown on Fig. 2, an action (also called *task*) consists of a set of *events* describing the change of the world induced by the action, a set of *hold* propositions expressing required conditions or the protection of some fact between two events, a set of resource usages, and a set of constraints on the timepoints and variables of the action. Note the *conteingent* keyword used to express that this duration should not be modified by the planner.

A plan relies on two CSP managers. A Simple Tem-

poral Network (STN) handles the timepoints and their binary constraints (ordering, duration, etc.). The other CSP manages atemporal symbolic and numeric variables and their constraints (binding, domain restriction, sum, etc.). Mixed constraints between temporal and atemporal variables can also be expressed (Trinquart & Ghallab 2001) (e.g. the relation between the distance, speed and duration of a move $?dist = ?speed * (et - st)$). These CSP managers compute for each variable a minimal domain which reflects only the necessary constraints in the plan. Thus the plan is least committed and as much as possible flexibility is left for execution.

The plan search explores a tree $\mathcal{T}$ in the partial plan space. In a POCL framework, a partial plan is generally defined as a 4-tuple (A, C, L, F), where A is a set of partially instantiated actions, C is a set of constraints on the temporal and atemporal variables of actions in A, L is a set of causal links[2] and F is a set of flaws. A partial plan stands for a family of plans. It is considered to be a valid solution if all its possible instances are coherent, that is F is empty.

The root node of $\mathcal{T}$ consists of: the initial state (initial values of all instantiated attributes), expected availability profiles of resources, goals to be achieved (desired values for specific instantiated attributes) and a set of constraints between these elements. The branches of $\mathcal{T}$ correspond to resolvers (new actions or constraints) inserted into the partial plan in order to solve one of its flaws. Three kinds of flaws are considered:
– *Open conditions* are events or assertions that have not yet been established. Resolvers consist in finding an establishing event (in the plan or a new action) and adding a causal link that protects the attribute value between the establishing event and the open condition.
– *Threats* correspond to pairs of *event* and *hold* which values are potentially in conflict. Such conflicts are solved by adding temporal or binding constraints.
– *Resource conflicts* are detected as over-consuming sets of potentially overlapping propositions. Resolvers include insertion of resource production action, etc
Thus, a planning step consists in detecting flaws in the current partial plan, selecting one, choosing a resolver in its associated list of potential resolvers and inserting it into the partial plan. This planning step is repeated until a solution plan is found. When a dead end is reached (flaws remain but no resolver are available), the search backtracks on a previous choice. The algorithm is complete and the flaw and resolver choices are guided by diverse heuristics discussed in (Laborie & Ghallab 1995). Note that the search is stopped as soon as a valid plan is found.

The advantages of the CSP-based functional approach are numerous in the context of plan execution.

[2]A causal link $a_i \xrightarrow{p} a_j$ denotes a commitment by the planner that a proposition p of action a_j is established by an effect of action a_i. The precedence constraint $a_i \prec a_j$ and binding constraints for variables of a_i and a_j appearing in p are in C.

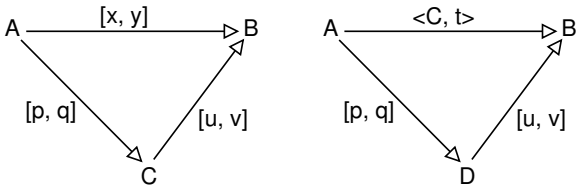


Figure 4: Two Network Examples

Besides the expressiveness of the representation (handling of time and resources), the flexibility of plans (partially ordered and partially instantiated, with minimal constraints) is well-adapted to their execution in an uncertain and dynamic environment. Plans are actually constrained at execution time. Finally, the planner, performing a search in the plan space, can be adapted to incremental planning and plan repair.

3DC+ algorithm

Nevertheles, there are still open problems such as how to handle the controllability issue. Regular propagation in STN, and by extension in the atemporal CSP, may shrink a temporal interval which may not be "controllable" by the planner. As a result, the execution may fail, not because the action model is wrong, but because the planner took some "freedom" with respect to what it is allow to control.

The 3DC+ algorithm was first introduced by (Vidal, Morris, & Muscettola 2001). Fig. 5 presents the general algorithm illustrated on the two examples on fig. 4.

Five various cases must be distinguished. If one considers the network on the left (fig. 4), with a contingent link AB:

Precede case This is the case where $u \geq 0$. In this case we must tighten AB to $[y - v, x - u]$.

Unordered case This is the case where $u < 0$ and $v \geq 0$. In this case and if $x < y - v$, we must add a ternary constraint, called a wait, on AB and of value $< C, y - v >$. It means that we must wait $y - v$ after the instantiation of A to instantiate B. We must also instantiate B at a time consistent with the constraints and after the observation of C.

If one now considers the network on the right (fig. 4):

Regression of wait Suppose a link AC has a wait $< C, t >$

- If a link DB (including AB itself) with an upper bound of q exists, then we must add a wait $< C, t - q >$ on AD.
- If a contingent link DB with $B \neq C$ and with p as lower bound exists, then we must add a wait $< C, t - p >$ on AD.

General reduction If a link AB has a wait $< C, t >$ and the lower bound of the contingent link that ends on C is l with $l < t$, then we must add a lower bound of l on AB.

Unconditional wait If a link AB has a wait $< C, t >$ and the lower bound of the contingent link that ends on C is l with $l > t$, then we must add a lower bound of t on AB and suppress the wait which is useless.

1. Compute the minimal STN. If it is not pseudo-controllable return false.
2. Select any triangle such that v (fig. 4) is non-negative. Introduce any tightenings required by the Precede case and any waits required by the Unordered case.
3. Do all possible regressions of waits, while converting unconditional waits to lower bounds. Also introduce lower bounds as provided by the general reduction.
4. If steps 2 and 3 do not produce any more tightenings, then return true, otherwise return to 1.

Figure 5: 3DC+ Algorithm

We have implemented the 3DC+ algorithm presented above in IxTeT, and we are thus able to produce plans which are dynamically controllable with waits. The resulting plans may not be as "efficient" as one produced without 3DC+, but, as we will see in the example section, it is more robust and still more efficient than a plan where all non controllable actions have been maximized.

Temporal executive, plan execution, repair and replanning

The temporal executive controls the temporal network of the plan produced by IxTeT by deciding the execution order of actions execution and by mapping the timepoints at their execution time. The execution of an action a with grounded parameters p_a, starting timepoint st^a, ending timepoint et^a, and identifier i_a is started by sending the command to the procedural executive. If the action is *non preemptive*, et^a is not controllable, and IxTeT just monitors if a is completed in due time. Otherwise et^a is controllable: if the action does not terminate by itself, it is stopped as soon (resp. as late) as possible if a is *early (resp. late) preemptive*.

IxTeT integrates in the plan the reports sent by the controlled system upon each action completion. A report returns the ending status of the action (*nominal*, *interrupted* or *failed*) and a partial description of the system state. If nominal, it just contains the final levels of the resources, if any, used by the action. Otherwise, it also contains the final values of the other state variables relevant to the action.

Besides completion reports, IxTeT also reacts to user requests to insert a new goal and sudden alterations of a resource capacity.

In any case, while execution is taking place, various events can forbid further execution of the plan:
– *temporal failures* The STN constrains each timepoint t to occur inside a time interval $[t_{lb}, t_{ub}]$. Thus two types of failure lead to an inconsistent plan: the corresponding event (typically, the end of an action) happens *too early* or *too late* (time-out).
– *action failure* The system returns a non nominal report.
– *resource level adjustment* If an action has consumed more or produced less than expected, the plan may contain future resource contentions.

When these occur, IxTeT starts and controls the processes of plan adaptation. To take advantage of the temporal flexibility of the plan, the dynamic replanning strategy has two steps. A first attempt is to repair the plan while executing its valid part in parallel. If this fails or if a timepoint times out, the execution is aborted and IxTeT completely replans from scratch.

Interleaving partial order planning and execution may insert flaws in the plan. We formally define under which conditions such a partial plan remains executable.

Definitions

We extend the previous definition of a partial plan to the definition of P_t: a **partial plan partially executed up to time t**.

Definition 1 $P_t = (RA_t, FA_t, S_t, G_t, C_t, L_t, F_t)$.

RA_t is the set of currently *running actions* ($a \in RA_t$ if $st^a_{ub} < t$ and $et^a_{ub} > t$), FA_t is the set of *future actions* ($a \in FA_t$ if $st^a_{ub} \geq t$). S_t represents the *state of the world* at time t. It is composed of 2 sets: $LgcS_t$ contains the last value of each attribute $la \in LgcA_g$, [3] $RscL_t$ contains the level at time t of each resource $r \in RscA_g$. G_t is the set of *goals* not yet completely achieved at time t (and eventually not established) [4]. C_t is the set of constraints on the variables appearing in FA_t, RA_t, S_t and G_t. L_t is the set of causal links supporting future actions. F_t is the set of flaws present in the partial plan at time t.

The level of a resource at a certain time in the future cannot be computed, since it depends on the partial order of actions using this resource. But at time t the past part of the plan is completely instantiated and linearized. Two cases have to be considered: if no running action modifies r, the exact level can be computed; if at least one action in RA_t requires the resource, only an estimate is available. We refer the reader to (Lemai & Ingrand 2004) for the details on how these evaluations are computed.

A timepoint in the temporal network may correspond to a goal timepoint or to an action starting or ending timepoint.

Definition 2 (executable timepoint) *A timepoint T is executable at time t if all timepoints T^p that must directly precede it in the temporal network have already been executed ($T^p_{lb} = T^p_{ub} < t$), if all positive waits on*

[3] In IxTeT, $LgcS_t$ contains the last executed *event* for each la.

[4] In IxTeT, a goal is represented by a grounded proposition $hold(GoalAtt(g){:}GoalValue, (st^g, et^g))$. G_t contains goals such that $et^g_{ub} \geq t$.

links with positive upper bound and which ends on T are enabled and if $t \in [T_{lb}, T_{ub}]$.

A goal is instantaneously achieved or persistent (achieve and maintain a property between st^g and et^g).

Definition 3 (achievable goal) *A goal g is achievable at time t if st^g is executable and if $g \notin F_t$.*

Let A_t^f be the set of actions that are involved in F_t. [5]

Definition 4 (executable action) *A future action a is executable at time t if its start timepoint is executable and if $a \notin A_t^f$.*

Definition 5 (executable plan) *A partial plan P_t is executable at time t if the constraint networks are consistent and if $RA_t \cap A_t^f = \emptyset$.*

Execution cycle

As previously explained, the system, when bootstrapped, produces a first plan (let us call it *ExecutingPlan*), and will only start execution afterward. The executive manages the messages received, the actions timeout, and the timepoints execution. Integrating messages in *ExecutingPlan* may partially invalidate it. If *ExecutingPlan* contains new flaws, a *plan repair* consists in keeping the structure of the plan (the ordering of actions) and taking advantage of the flexibility to try and find a solution plan. The user defines the maximum time allowed for plan repair (μ). If plan repair takes more than μ, it is suspended to allow reactivity to events and concurrent execution of the valid part of the plan.

Yet, to distribute planning on several cycles raises two problems:

Which plan does the concurrent execution rely on, especially if no solution has been found? This plan has to be *executable*. At each planning step, the node is labeled if the current partial plan is *executable*. When μ has elapsed, the last labeled partial plan becomes *ExecutingPlan*.

Which plan and which search tree the planning process rely on in the next cycle? If no change has been made meanwhile (no timepoint execution, no message reception), the search tree can be kept as is and further developed during the next *plan repair* part. However, if the plan has been modified, a new search tree whose root node is the new *ExecutingPlan* is used, and the planning decisions made in previous cycles are final.

The following subsections further detail the different phases of the executive loop. Basically, all modifications made to *ExecutingPlan* have to guarantee that an *executable* plan is available after each phase of the cycle. If this condition does not hold, the cycle is stopped and a complete replanning is mandatory. During a cycle without plan repair, *ExecutingPlan* remains a solution plan.

[5] The determination of A_t^f is straightforward in the case of open conditions and resource conflicts. In a threat case, an action a_k has effects in contradiction with the establishment of proposition p by the causal link $a_i \xrightarrow{p} a_j$ and $(a_i \prec a_k \prec a_j)$ is consistent. A_t^f contains a_k and a_j.

Message integration

A message can be a report upon action completion; a new goal request or a notification of a capacity alteration (we do not detail the two last ones, and refer the reader to (Lemai 2004) for a complete explanation on these).

A report is associated with the ending timepoint et^a of the corresponding action a. If the message is received inside the bounds $[et^a_{lb}, et^a_{ub}]$, et^a is set to the current time t (equivalent to posting the constraint $(et^a - origin) = t$ in the STN). Otherwise, two situations arise. If there is no flexibility left in the plan, it is not executable anymore. Else, a new end timepoint, set to t and constrained to occur before the executable timepoints, is created and the failed one is relaxed. The network is then recomputed. In IxTeT, such an operation keeps the network consistent, since the only constraint that can be specified between two actions a and a' is a precedence constraint which upper bound is flexible: $(st^{a'} - et^a)$ in $]0, +\infty[$. If the report contains information about the state, S_t is updated in the following way:

Resource level - For each resource r, the report returns the current "real" level l_r. l_r is compared to the forecasted evaluation (see (Lemai & Ingrand 2004)) which are properly updated accordingly. Plan repair is requested in case of over-consumption and in case of over-production of a reservoir resource (which may then overfill).

State variables - $LgcS_t$ contains the last value for each instantiated logical attribute. If the report is nominal, $LgcS_t$ is updated with the effects of a expected in the plan. Otherwise, it is updated with the values returned in the report. A value is not inserted if it leads to a non executable plan (that is it threatens some proposition of a running action a_r). In that case and if a_r is preemptive, its interruption is requested. Else, the value is inserted and causal links which contradict it are broken. This update leads to an executable plan with open conditions on which plan repair can be processed.

After message integration, the plan may contain flaws (open conditions and/or resource conflicts) on a set of grounded attributes Att^f, possibly repaired thanks to the insertion of new actions. Let us consider Att^i the set of the attributes appearing in the potentially inserted actions. Additional causal links, protecting propositions in the plan on attributes in Att^i, have to be broken to allow the insertion of these actions in the current plan structure.

The determination of Att^i is based on information given by an abstraction hierarchy verifying the Ordered Monotonicity Property (Knoblock 1994; Garcia & Laborie 1995) and generated offline from the model de-

scription. Notably, this hierarchy points out the primary effects of an operator, which justify its insertion to solve a flaw. Let us call *main attributes* of an action the attributes appearing in its primary effects. Att^i, initialized with Att^f, is computed by searching the action operators for which at least one attribute att_m in Att^i is a main attribute. This operator is partially grounded (by binding its corresponding parameter with att_m) and the (eventually grounded) attributes appearing in the operator and not yet taken into account are added to Att^i. The algorithm proceeds recursively until a fixed point is reached.

Finally, the partial plan is executable and the sets of actions that are independent from the failures remain executable.

Plan repair

The plan repair is similar to the IxTeT search process in the plan space. The root of the search tree $\mathcal{T}$ is *ExecutingPlan*, partially invalidated. Planning is distributed, if necessary, on several cycles and each time a new timepoint is inserted, it is constrained to occur after the end of the current cycle. Planning during one cycle is done one step at a time until it results into a dead-end (there is no solution), or a solution is found or a deadline is reached. This deadline corresponds to the user defined time (μ) allocated to the plan repair part of the cycle time.

Some aggregation mechanisms allow a reduction of the search space. In IxTeT, the establishing events are looked for in $LgcS_t$ and executed resource propositions are aggregated in one proposition.

This plan repair process is not guaranteed to find a valid plan, yet it can avoid aborting execution and completely replanning at each failure. By invalidating only a part of the plan, the amount of decisions is rather limited and a repaired plan may be found in a few cycles. Plan repair is especially efficient and useful for temporally flexible plans and plans with some parallelism. This mechanism is also efficient to compensate for inadequate models of actions. Consider a move(L_1,L_2) action, which is defined as a late preemptive action in the IxTeT model. If the robot takes longer than expected in the model (e.g. due to unexpected obstacle avoidance), the action is interrupted. The controlled system returns the intermediate location L_i and, if some temporal flexibility remains, a new move(L_i,L_2) is immediately inserted and launched. This example is representative of the failures that frequently break plan execution.

Action

Each timepoint is associated to an *execution time* t_{exec}. If T is a start or goal timepoint, or an end timepoint of an early preemptive action, $t_{exec} = T_{lb}$. If T is an end timepoint of a late preemptive action, $t_{exec} = T_{ub} - ts$. If T is an end timepoint of a non preemptive action, $t_{exec} = T_{ub}$. The executive determines the set of timepoints to execute during the current cycle (*ExecTPs*): these timepoints are executable and their execution time happens before the end of the cycle. *ExecTPs* is updated after each timepoint execution to take into account newly executable timepoints. The detail of a timepoint execution depends on its type and timeouts are raised when reports have not been received in time.

Complete replanning

Let us call $P_{t_s} = (\emptyset, FA_{t_s}, S_{t_s}, G_{t_s}, C_{t_s}, L_{t_s}, F_{t_s})$ the plan obtained once execution is stopped. An initial plan is extracted from P_{t_s} as:
$P_{t_i} = (\emptyset, \emptyset, S_{t_i}, G_{t_i}, C_{t_i}, \emptyset, F_{t_i})$, with
$S_{t_i} = S_{t_s}$, $G_{t_i} = \{g \in G_{t_s} / \textit{temporal constraints on } g \textit{ are coherent with current time}\}$, $C_{t_i} = \{c \in C_{t_s} / c \textit{ is a constraint just on variables appearing in } S_{t_i} \textit{ and } G_{t_i}\}$($C_{t_i}$ *notably contains constraints on origin and horizon timepoints)*, and $F_{t_i} = G_{t_i}$.

POCL planning cannot be interrupted at any time and come up with an applicable plan. Still we have to guarantee that at the end of the replanning process, there remains enough time to execute the solution plan and meet the goal deadlines. We propose to add a specific flexible timepoint T^{end} to P_{t_i}, that corresponds to the end of the planning process. T^{end} is only constrained to occur between t_i and the end of the horizon. Each time a new timepoint is inserted by the planning process, it is constrained to occur after T^{end}. Thus T^{end}_{ub} decreases as new actions or new temporal constraints are added, and there is not enough time to execute the current plan if $T^{end}_{ub} <$ current time. Note however that T^{end}_{ub} can increase when backtracking.

The strategy is then to plan one step at a time until it results into a dead-end, or a solution is found, or a time limit l is reached. l is defined as $l = T^{end}_{ub} - d$, d being a *slack* duration to save enough time at the end of planning for cycle initialization. l is updated after each planning step. Planning is stopped when l is reached unless the next step corresponds to a backtrack node. In that case, and if the next step increases l, planning is pursued.

If planning is aborted without finding a solution, some goals are rejected and a new attempt is done (Lemai 2004).

Integration and example of scenario

We illustrate the capabilities and the performances of IxTeT with an example of a scenario for a rover with an exploration mission. In such a domain, the quantitative effects and durations can be estimated in advance for planning but are accurately known only at execution time (e.g. the actual compression rate of an image or the actual duration of a navigation task), thus requiring regular updates and look-ahead capabilities to manage unforeseen situations and resource levels. We also illustrate the advantage of using the 3DC+ algorithm in order to produce a more robust plan and compare with a plan without temporal controllability.

IxTeT has been integrated in the decisional level of the LAAS architecture (Alami *et al.* 1998) and

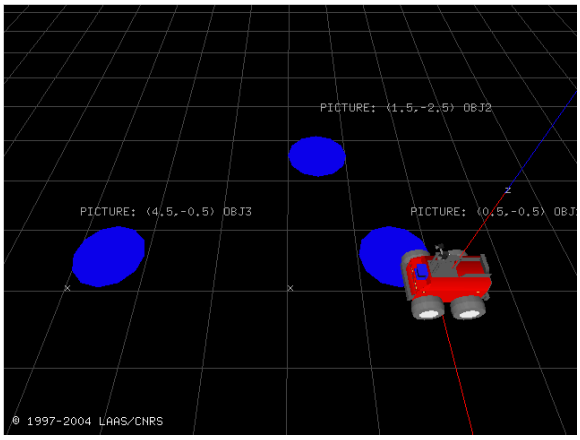


Figure 6: DALA GUI showing the goals of the exploration mission.

used to control an iRobot ATRV (see the first section and Fig. 8). We set up an exploration mission scenario which requires the robot to achieve three types of goals (see Fig. 6): "take pictures of specific science targets" (in locations (0.5,-0.5), (4.5,-0.5), (1.5,-2.5)), "communicate with a ground station during visibility window" ($W_1[117-147]$), and "return to location (0.5,-0.5) before time 500". Dala runs a 3 GHz Pentium IV (1 GB memory) under Linux and is equipped with the following sensors: odometry and a stereo camera pair mounted on a pan&tilt unit (PTU). Five main actions are considered at the mission planning level: take_picture, move_ptu, move (Fig. 2), download_images, communicate. The first three actions are performed by Dala, while the last two are realistically simulated.

There are specific constraints attached to each tasks. The pan&tilt unit must be warmed up ten seconds before it can move. During a move action (of the rover), the camera must be pointed at a specific angle in order to provide the best perception of the environment. Thus the move action and the move_ptu action are mutually exclusive, however the pan&tilt unit can be warmed up during the "end of the move". It allows us to start a move_ptu action before the end of the move that precedes it without "stopping" the move itself. Yet, to do so, we need 3DC+ to correctly produce and execute this plan. Without this, IxTeT produce a plan which may shorten the duration of the move to its lower limit and we will most likely get a temporal failure. You can see on Fig.7 that in the top plan the end of two move action is overlapped by a move_ptu action. Unfortunately at this stage, the IxTeT Plan Viewer used to produce these screen dumps does not show the wait introduced by the 3DC+ algorithm. The plan on the bottom part of the picture has been produced by over constraining the move_ptu action to take place strictly after the move action. This plan is clearly safe but less efficient and flexible than the previous one.

The plan execution is controlled by both executives as

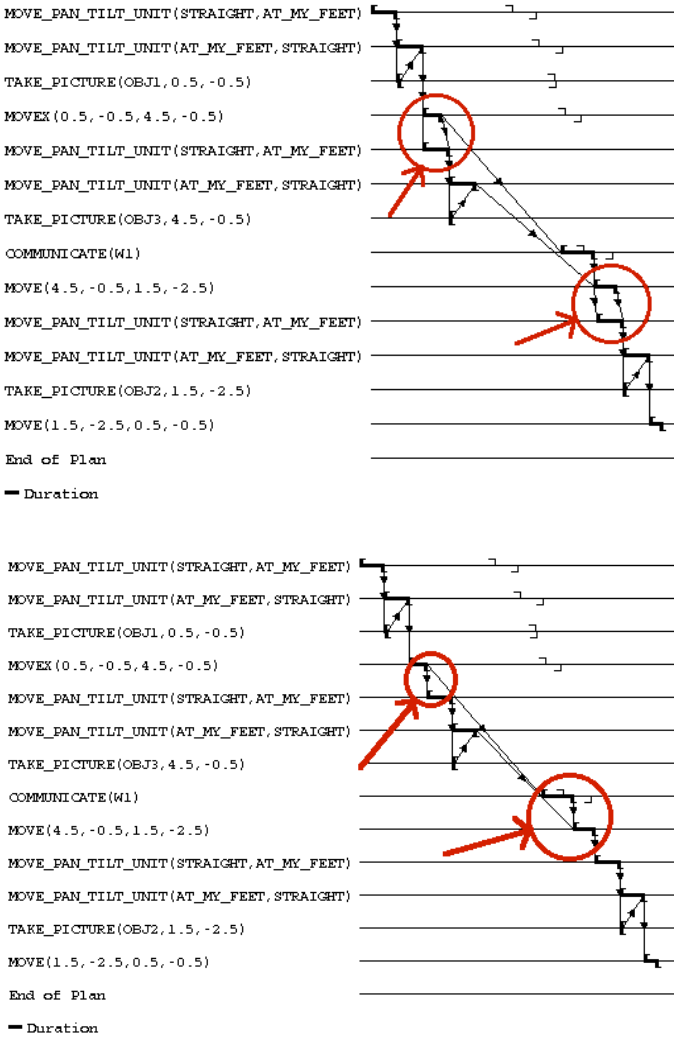


Figure 7: Initial plan produced with the use of 3DC+ (top) and without (bottom) (note the flexibility left, the dependencies and the parallelism).

follows. IxTeT decides when to start or stop an action in the plan and handles plan adaptations. OpenPRS expands the action into commands to the functional level[6], monitors its execution and can recover from specific failures. It finally reports to IxTeT upon the action completion.

This mission (the corresponding initial plan with 3DC+ is shown in Fig. 7) has been executed by Dala under IxTeT control (with $\mu = 1s$ and total cycle duration= $2s$). The initial plan with 3DC+ was produced in $7.1s$, and the plan without 3DC+ was produced in $4s$. Each resulting run is different.

Figure 9 shows the duration of each phase of the cy-

[6]For the download_images and communicate actions, specific procedures simulate the visibility windows and the gradual download of images.

Figure 8: The robot Dala.

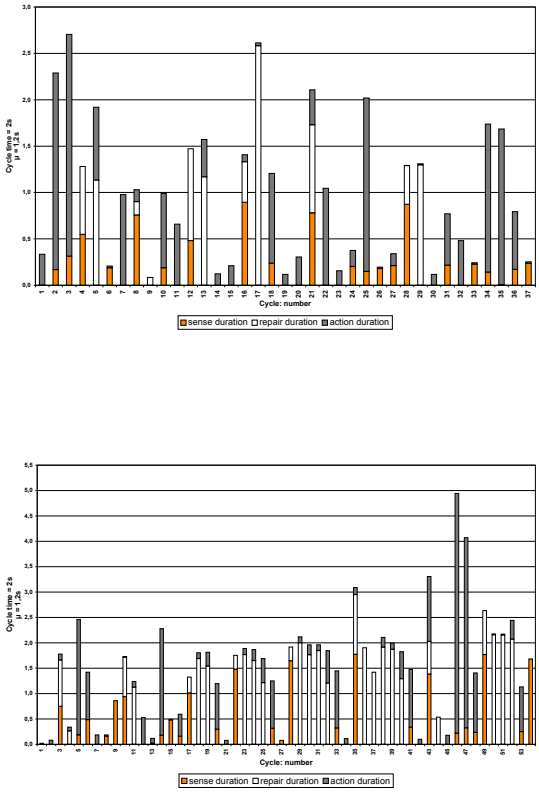


Figure 9: Cycle duration of plans with 3DC+ (top) and without 3DC+ (bottom).

cle for two different runs (one with 3DC+ and another without). The two runs are different because the real execution lead to more frequent failures of the move action in the second run. Yet, we see that using 3DC+ during execution does not increase execution time too much.

Discussion and Prospectives

If one looks at the current state of the art, few high level planning systems have been integrated onboard real robots while running complex navigation software. Many architectures (such as Claraty (Estlin *et al.* 2002)) provide a "decisional" level for such components , but little has been done as far as deploying them entirely on real systems. The main reason is probably that despite the availability of good planning systems, few of them integrate the proper plan repair and replanning mechanisms. Still, the ROGUE system (Haigh & Veloso 1998), for instance, performs planning for asynchronous goals and execution monitoring enhanced with learning capabilities. In (Beetz 2000), the authors propose a different approach where the plans themselves specify the adaptation processes as subplans. In any case, very few approaches explicitly handle time and address the issue of temporal execution. The CASPER system (Chien *et al.* 2000) (part of Claraty) performs continuous planning interleaved with execution. State and temporal data are regularly updated and potential future conflicts are incrementally resolved using iterative repair techniques. However this approach does not handle conflicts which appear within the replanning time interval. Other approaches such as IDEA (Finzi, Ingrand, & Muscettola 2004) are more radical and provide an architecture which seamlessly integrates temporal planning and execution control: each component can be seen as an agent running a reactive planner, and sharing with the others parts of a global temporal model specifying the "behavior" as well as the communication between agents.

We have presented in this paper the IxTeT system which combines a temporal lifted POCL planner with a temporal executive to integrate deliberative planning, execution monitoring and replanning while respecting real-time constraints. This approach cannot account for all the possible execution failures in all their generality. Nevertheless, in many situations where some temporal and resource flexibility has been left,one can expect the presented repair techniques to greatly improve the overall performance of the system by:

- reducing the number of complete replannings,
- improving the system reactivity to unexpected events,
- taking into account new goals on the fly,
- managing the changes in the resources capacity,
- managing the uncertainty in the model description (actions duration, resources consumption/production).

Moreover, by implementing 3DC+ in IxTeT, we have a better handling of temporal controllability, and pro-

duce plans which are more robust at execution time, without a major degradation in performance.

We have conducted a number of field experiments. Although preliminary, the current results are quite promising. First, we show that planning with time and resource combined with execution control, plan repair and replanning can be used on real world problem. Second it shows that such an approach can be deployed on current hardware along with the "state of the art" navigation software (stereo vision, terrain mapping, path planning, visual odometry, etc).

Yet, IxTeT effectiveness can be increased by improving replanning strategies (rejected goals selection, state update requests).

Despite the obvious application of systems such as IxTeT to exploration probes and rovers, one can easily see the possibilities it opens for service robotics (with the added value of human robot interactions and problem joint resolutions) and fields robotics, where planning and execution control problems are also present.

References

Ai-Chang, M.; Bresina, J.; Charest, L.; Jónsson, A.; Hsu, J.; Kanefsky, B.; Maldague, P.; Morris, P.; Rajan, K.; and Yglesias, J. 2003. Mapgen: Mixed initiative planning and scheduling for the mars 03 mer mission. In *Proceedings of iSAIRAS*.

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. 1998. An architecture for autonomy. *International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming* 17(4):315–337.

Beetz, M. 2000. Runtime plan adaptation in structured reactive controllers. In *Proceedings of the Fourth ICAA*.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *AAAI*.

Estlin, T.; Fisher, F.; Gaines, D.; Chouinard, C.; Schaffer, S.; and Nesnas, I. 2002. Continuous Planning and Execution for an Autonomous Rover. In *Proc. Third International NASA Workshop on Planning and Scheduling for Space*.

Finzi, A.; Ingrand, F.; and Muscettola, N. 2004. Model-based executive control through reactive planning for autonomous rovers. In *IROS 2004 (IEEE/RSJ International Conference on Intelligent Robots and Systems)*.

Garcia, F., and Laborie, P. 1995. Hierarchisation of the search space in temporal planning. In *EWP*.

Goldberg, S.; Maimone, M.; and Matthies, L. 2002. Stereo vision and rover navigation software for planetary exploration. In *Proc. IEEE Aerospace Conference*.

Haigh, K. Z., and Veloso, M. M. 1998. Planning, execution and learning in a robotic agent. In *AIPS*.

Knoblock, C. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *IJCAI*.

Lacroix, S.; Mallet, A.; Bonnafous, D.; Bauzil, G.; Fleury, S.; Herrb, M.; and Chatila, R. 2003. Autonomous rover navigation on unknown terrains, functions and integration. *IJRR*.

Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In *AAAI 2004, July 25-29*.

Lemai, S. 2004. *IxTeT-eXeC: planning, plan repair and execution control with time and resource management*. Ph.D. Dissertation, LAAS-CNRS and Institut National Polytechnique de Toulouse, France.

Py, F., and Ingrand, F. 2004. Dependable execution control for autonomous robots. In *IROS 2004 (IEEE/RSJ International Conference on Intelligent Robots and Systems)*.

Rajan, K. 2004. Invited talk: Mapgen. In *IWPSS 2004, 4th International Workshop on Planning and Scheduling for Space, June 23 - 25*.

Trinquart, R., and Ghallab, M. 2001. An extended functional representation in temporal planning : towards continuous change. In *ECP*.

Vidal, T.; Morris, P.; and Muscettola, N. 2001. Dynamic Control of Plans With Temporal Uncertainty. In *IJCAI*, 494–502.

Making Robot Learning Controllable: A Case Study in Robot Navigation

Alexandra Kirsch, Michael Schweitzer, Michael Beetz

Abstract

In many applications the performance of learned robot controllers drags behind those of the respective hand-coded ones. In our view, this situation is caused not mainly by deficiencies of the learning algorithms but rather by an insufficient embedding of learning in robot control programs. This paper presents a case study in which ROLL, a robot control language that allows for explicit representations of learning problems, is applied to learning robot navigation tasks. The case study shows that ROLL's constructs for specifying learning problems (1) make aspects of autonomous robot learning explicit and controllable; (2) have an enormous impact on the performance of the learned controllers and therefore encourage the engineering of high performance learners; (3) make the learning processes repeatable and allow for writing bootstrapping robot controllers. Taken together the approach constitutes an important step towards *engineering* controllers of autonomous learning robots.

Introduction

Implementing competent autonomous robot control systems that can accomplish large spectra of dynamically changing and interacting tasks is very difficult. The realization and maintenance during their development requires the control systems to be equipped with, and make ample use of autonomous learning mechanisms. Unfortunately, the performance of learned routines typically drags substantially behind the performance of handcoded ones, at least if the control tasks are complex, interact, and are dynamically changing.

In our view, this situation is caused not primarily by deficiencies of learning algorithms but rather by an insufficient embedding of learning into robot control. For a robot to learn effectively and successfully it does not suffice to merely apply the right learning algorithm. Rather, the robot must also be able to recognize the experiences relevant for learning, to actively acquire specific experiences to accelerate learning and to select informative experiences and throw away misleading ones. We know from data mining applications that the realizations of these tasks have an tremendous impact on the performance of learning and data mining applications.

Beetz et al. [2004] have proposed ROLL (Robot Learning Language, formerly called $\text{RPL}_{\text{LEARN}}$), an extension to the robot control and plan language RPL, that allows programmers to specify such mechanisms declaratively and modularily as part of the control program. ROLL introduces experiences, distributions and abstractions thereof, learning tasks, and learned routines as first class objects into the language. It also provides transparent and modular specification mechanisms for them. Using ROLL, a programmer can specify an experience class relevant for a given learning task by adding a perception mechanism for it, a routine for performing physical actions to gather such experiences, a critic that decides whether or not an experience is informative. Using the last specification, a robot can recognize an experience in which it collided with an object, which is not informative for learning the dynamics of the robot.

In this paper we evaluate some of the claims made by Beetz et al. [2004] by applying the extended language to learning an example class of navigation tasks for autonomous robot soccer. In the context of this case study, we will discuss whether changing the parameters and the mechanisms that are made explicit in ROLL have substantial impact on the performance of learned routines. We will also investigate whether the changes can be made modularily and transparently and whether the language extensions encourage the *engineering* of autonomously learning controllers through a seamless integration of programming and learning.

The case study demonstrates the huge potential of control languages that support learning for the realization of more competent controllers that are easier to develop and maintain.

In the remainder of this paper we proceed as follows. Section briefly introduces some of the language constructs provided by ROLL that are used for our case study. In section we present a case study demonstrating the language ROLL. Thereafter we compare several navigation routines that were implemented with ROLL. Finally we discuss related work and conclude with section .

The Robot Learning Language ROLL

Before we start with our case study, let us first describe the computational model for the interpretation of ROLL controllers and then the key language constructs provided by ROLL for the embedding of learning mechanisms.

The Interpretation Model of ROLL

The extended robot control language ROLL assumes that ROLL controllers are executed by an interpretation model that has the structure and components depicted in figure 1. The main parts of the system are the *performance element* that controls the robot, the *critic* that executes the learning task specific perception mechanisms, the *learning element* that reasons about and modifies the performance element in order to improve its behavior. To do so, the learning element uses a database of experiences and a library of learning algorithms as its resources. Finally, the computational model includes a *problem generator* that allows the robot to acquire relevant experiences actively. In the remainder of this section we will briefly sketch the functionality of the individual components of the interpretation model.

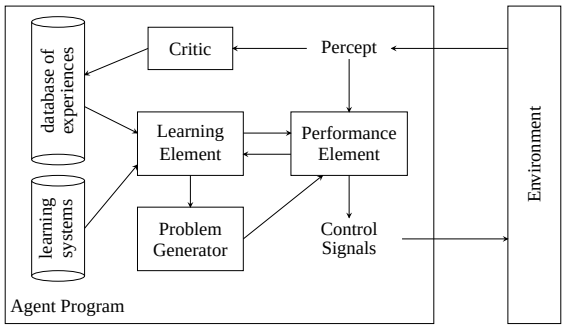


Figure 1: Overview of a learning agent after (Russell & Norvig 1995)

The *performance element* realizes the mapping from percepts into the actions that should be performed next. It contains code pieces, called *control tasks* that might not yet be executable or optimized. These are the code pieces to be learned. Thus the ROLL interpreter might have to interpret a control task that has not yet been learned, using the ROLL specification of the learning problem. In this case, it automatically activates the learning process including the collection of the necessary experiences, and continues with the interpretation after the learning process has generated an executable code piece for the control task.

The *critic* is best thought of as a learning task specific abstract sensor that transforms raw sensor data into information relevant for the learning element. To do so the critic monitors the collection of experiences and abstracts them into a feature representation that facilitates learning. The critic also generates feedback signals or rewards that assess the robot's performance during an episode. Finally, the episodes are stored and maintained in a relational database system coupled with a datamining toolset as resources for learning. The current ROLL version uses MySQL[1] as its database system and Weka[2] for data mining.

The *learning element* uses experiences made by the robot in order to learn the routine for the given control task. To do so, the learning element selects a subset of experiences from the episode database and transforms these experiences into input data for the learning algorithm to be applied. The learning element also specifies the appropriate parameterization of the learning mechanism, the bias, to perform the learning task effectively. Finally, the learning element specifies how the result of the learning process is to be transformed into a piece of code that can be executed by the performance element.

The *problem generator* is called with an experience class and returns a control routine that, when executed, will generate an experience of the respective class. The new parameterizations are generated as specified in the distribution of parameterizations of the experience class.

Learning-specific Constructs of ROLL

In order to write a ROLL controller, a programmer has to specify control tasks that are to be learned, experiences that are needed to learn a routine for the tasks, abstractions of experiences that are better correlated with the concepts to be learned, and learning algorithms, their parameterization, conversions of experiences into input data of the algorithm and the transformations of the algorithm output into pieces of the control program. For each of these aspects ROLL provides modular and transparent means for their specification.

To specify the experiences for a learning task we must code how the experiences are to be recognized, how they can be actively acquired by performing control routines, and what the distribution of experiences should be. The performance of learned routines often improves as the distribution of experiences matches the expected distribution of control tasks which they are learned for. To facilitate learning the programmer can also define suitable abstractions or "feature languages". The experiences are stored in an episode database automatically.

experience class ⟨*name*⟩
  with feature language ⟨*feature language*⟩
    abstraction ⟨*abstraction*⟩
    distribution ⟨*distribution*⟩
    methods ⟨*detect-method*⟩, ⟨*collect-method*⟩

A learning problem consists of two parts: the experiences and a learning element. The experiences are extracted from a database. This gives the programmer the freedom to choose from the gathered experiences only those that are most suited for the particular problem. For this purpose we use an abstract language that was designed for data cleaning (Galhardas *et al.* 2001). This language is an extension of SQL and provides, among others, constructs for matching, clustering, and merging of data. Thus a set of experiences of an experience class can be used for different learning problems. The learning element contains the choice of a learning algorithms and its parameterization for the learning problem.

learning problem ⟨*name*⟩
  experiences ⟨*experience set*⟩
  learning element ⟨*learning element*⟩

Given a set of experiences a robot learning problem is essentially the application of an appropriately parameterized learning algorithm, the transformation of the abstracted experiences into the input format of the learning algorithm,

[1] http://www.mysql.com/

[2] http://sourceforge.net/projects/weka/

and the generation of code that is executable within the controller and solves the control task from the output of the algorithm.

Navigation with Bézier Curves: A Case Study

In the domain of robot soccer, the navigation is a fundamental issue. We consider mobile robots with a simple differential drive which we can control with an an abstract interface that allows for the drive control in terms of a desired rotational velocity v and translational velocity ω of the robot. Steering differential drives for complex navigation tasks with high performance is very difficult. As described and justified by experimental results in (Beetz *et al.* 2004) we perform learning tasks in a simulator with the robot dynamics learned from the real physical robots.

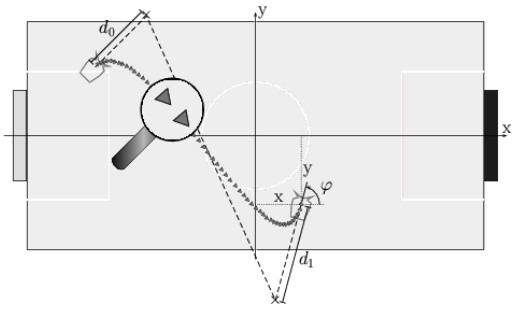


Figure 2: The navigation problem using Bézier curves

In robot soccer it often does not suffice to reach a position, but positions must be reached in orientations that facilitate subsequent actions. Thus we consider navigation tasks that are specified by the current robot position and orientation $\langle x, y, \varphi \rangle$ and the intended one $\langle x_g, y_g, \varphi_g \rangle$ and use cubic Bézier curves for specifying the trajectories to be followed (figure 2). With $t \in [0, 1]$ a cubic Bézier curve is defined as $B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^(t-1)P_2 + t^3 P_3$.

The P_i are called control points. For the navigation problem P_0 is the starting position and P_3 is the goal position. With the orientation given at P_0, P_1 is determined by the distance to P_0, for P_3 and P_2 respectively. So for us a Bézier curve is determined by the two distances d_0 and d_1.

```
control routine navigation (⟨x_g, y_g, φ_g⟩)
  trajectory := calculate-bezier-curve(⟨x, y, φ⟩, ⟨x_g, y_g, φ_g⟩)
  do
    trajectory-point := pop(trajectory)
    do
      get-next-command(⟨x, y, φ⟩, trajectory-point)
    until at-point(trajectory-point)
  until at-point(⟨x_g, y_g, φ_g⟩)
```

Figure 3: Overview of our navigation routine.

We decompose the navigation problem into two subproblems as shown in figure 3: (1) Finding a suitable Bézier curve. We call this the high-level navigation problem. (2) Navigating from a point on the curve to the next trajectory point. This we refer to as the low-level navigation problem.

To keep the case study simple, we abstract away from other parameters that influence the performance of the navigation routine such as the density of trajectory points on the Bézier curve. They have small impact on performance and we keep them constant in our experiments.

Finding a Trajectory

In our case study we have investigated three different approaches for the implementation of the function calculate-bezier-curve: a fully programmed one that performs an exhaustive search through all possible solutions; a heuristic one, and a learned one.

The fully programmed function produces very good results, but is too slow to be used at execution time. The heuristic solution calculates d_0 and d_1 independently, based on the angle deviation of each point with respect to the line of sight between the two points. A third solution is to learn the function calculate-bezier-curve by experience. Unfortunately, there is a strong dependency between the parameters d_0 and d_1. Therefore we split the problem into two learning tasks: one to determine only d_0, the other one to determine d_1 as a function of d_0.

Low-Level Navigation

In this section, we explain the solution of the low-level navigation problem in more detail. We demonstrate the explicit specification of a learning problem and show different alternatives for solving the navigation task.

The low-level navigation is simpler than the original navigation problem in that the target points are rather close, so that the goal angle can be omitted. The orientation of the overall navigation task is achieved by following the Bézier curve.

(1) Parameterization of the Experience Abstraction. The choice of abstractions determines how concisely situations and tasks can be represented and how strongly the characterizations of situations and tasks correlate with the concepts to be learned. The abstraction has a strong effect on the performance of the learning process.

Two possible feature languages with their respective abstractions are shown in table 1. The first one describes the distance between the start and goal point and the angle of the start point relative to the line of sight. In the second possibility an arc is drawn between the start and goal point with the orientation vector at the start point as a tangent. Here the abstraction is described in terms of the radius r_c of this arc.

(2) Parameterization of the Experience Distribution. It is often useful to specify more than one distribution for obtaining a broader variety of experiences. Therefore we define the relevant parameters for the distribution first.

```
distribution parameters nav distribution
    ⟨x, y, φ⟩_start: constant ⟨-5.0, -2.5, 0.0⟩
    φ_end:           constant 90.0
    rotation:        range ⟨1.0, 180.0⟩
    translation:     range ⟨0.0, 1.0⟩
```

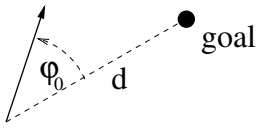
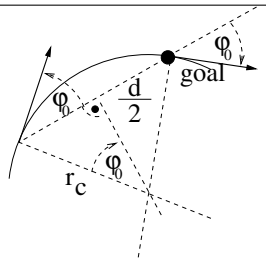
features 1	features 2
	
$d \leftarrow \sqrt{(x_g - x)^2 + (y_g - y)^2}$ $\varphi_0 \leftarrow \left\lvert \varphi - \arctan\left(\frac{x_g - x}{y_g - y}\right)\right\rvert$	$r_c \leftarrow \frac{\sqrt{(x_g-x)^2+(y_g-y)^2}}{2 \sin \varphi_0}$
$d \times \varphi_0 \rightarrow v \times \omega$ (AB 1)	$r_c \rightarrow v \times \omega$ (AB 2)
abstraction 1	abstraction 2

Table 1: Different parameterizations for feature language and abstraction.

Now different distributions can be defined by setting the values of the non-constant parameters. The values of a parameter can be obtained systematically, randomly or by a list of fixed values. If not stated otherwise, the parameters are assumed to be independent, although distributions over combinations of parameters can be defined as well.

distribution medium curves **of type** nav distribution
rotation: **systematic range** $\langle 20.0, 60.0 \rangle$ **step** 1.0
translation: **systematic range** $\langle 0.2, 1.0 \rangle$ **step** 0.05

Instead of defining experience distributions, it is possible that the robot use experiences aquired during its operation. In robot soccer, we can use the experiences made during games.

(3) Methods for Recognizing Experiences. To collect experiences the robot has to recognize them and detect failures during their acquisition. In our example an experience starts at a certain point and terminates when the robot has reached a certain turning angle. Furthermore we check if the robot has gone out of the field or exhausted the given time resources.

The methods for gathering experiences are usually straightforward, so one doesn't have to experiment with them the way one does with the other parameters like experience abstraction or distribution.

(4) Parameterization of the Experience Extraction. As described in section , a learning problem is defined by a set of experiences and a learning element. The experiences are extracted from an episode database.

Table 2 shows the specifications of two possible sets of experiences. The first one is trivial and uses all the available experiences. The second one is more sophisticated in that it selects only fast examples. The table match-nav is obtained by applying a matching operator as described in (Galhardas *et al.* 2001) on the stored examples, so that similar routes are grouped together.

CREATE MATCHING match-nav
FROM nav-exp ne1, nav-exp ne2
LET distance = pathSimilarity(ne1.id, ne2.id)
WHERE distance $<$ maxDist(ne1.id, ne2.id, δ)
INTO match-exp

ES 1	**def-experience-set** all-experiences SELECT id FROM nav-exp
ES 2	**def-experience-set** fast-experiences SELECT DISTINCT time,id FROM (SELECT time,id,id1 FROM 'nav-exp' ne JOIN 'match-exp' me ON ne.id=me.id2) t1 JOIN (SELECT MIN(time) mt,id1 FROM 'nav-exp' ne JOIN 'match-exp' me ON ne.id=me.id2 GROUP BY id1) t2 ON t1.time=t2.mt AND t1.id1=t2.id1;

Table 2: Different parameterizations for experience set.

(5) Parameterization of the learning element. Now having chosen the experiences that are to be used for learning, we only have to describe the parameters of the learning element. One parameterization could be

learning element nav learning element
use system SNNS
with parameters
hidden units: 5
cycles: 50
learning function: Rprop

(6) Parameterization of the amount of programming. Often learning alone is not enough to solve complex problems. For instance, our learning approach to the low-level navigation has one fundamental drawback. It is hard to decide whether a navigation routine is better than another. There can be cases when a routine is fast, but inaccurate. Depending on the situation the robot must have access to navigation routines with different qualities. Any learned low level navigation routine can only be optimized under one criterion.

Instead of learning completely different routines, we can reformulate the learning problem by inverting the abstraction:

abstraction nav abstraction
$v \times \omega \rightarrow r_c$ (AB 3)

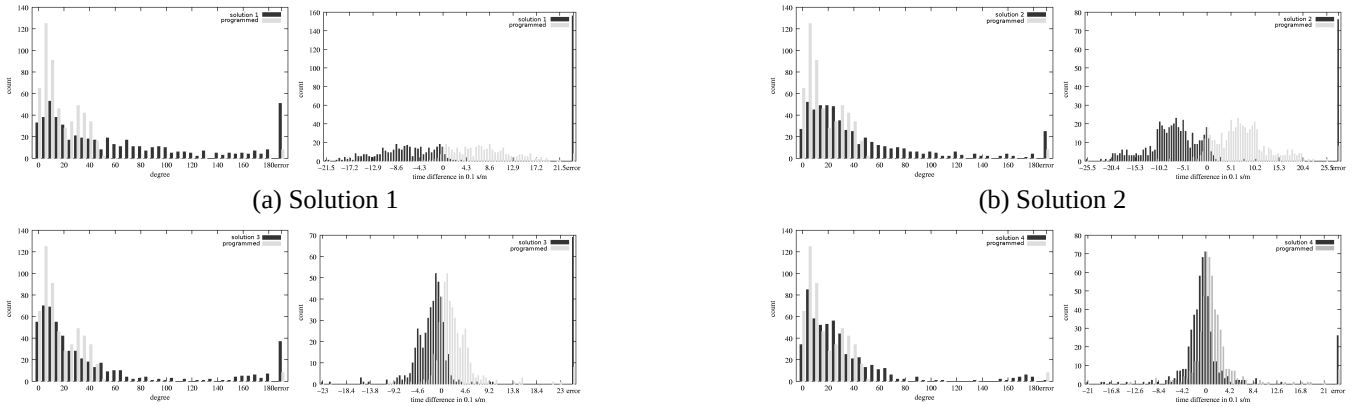
Here we know our translational and rotational velocities and are interested in the arc the robot will go with these parameters. The learned function can now be used for a search algorithm. We optimized the function so that the robot goes as fast as possible while rotating as little as possible. But it would be easy to write functions with different criteria.

Experimental Results

In this section we present experimental results that were obtained by combining the different parameters explained in the previous section. The following table gives an overview of our solutions.

solution	1	2	3	4
abstraction	AB 1	AB 1	AB 1	AB 3
experience set	ES 1	ES 1	ES 2	ES 2
programming	none	none	little	yes
high-level	heuristic	learned	learned	learned

Experiments. In our experiments we gave the robot several navigation tasks where it had to reach a point with a certain orientation. Only runs that reached the point within a given radius were considered successful. The successful



(a) Solution 1 (b) Solution 2

(c) Solution 3 (d) Solution 4

Figure 4: Comparison of different parameterizations. In each case the left diagram shows the accuracy of the goal angle, the right one the time difference compared to a completely programmed routine, which is drawn in light grey.

runs were then categorized along two lines: the accuracy the desired orientation was achieved with and the time needed.

In section we described a fully programmed routine that provides good solutions, but is intractable for real-time applications. Since in our simulation the computation time for finding a solution can be disregarded, we consider this routine as the best possible solution and therefore compare our (partly) learned solutions to this reference routine.

The diagrams for the accuracy in figure 4 give the angle deviation at the goal point. At the rightmost side the cases are denoted when the routine didn't reach the point at all.

In the time diagrams the time difference of two routines is shown. When comparing two identical routines, the diagram shows two bars of equal height at the origin. The faster a routine, the more bars of its color are on the right hand side. In the time comparison, runs are considered successful only when the accuracy is better than $45°$.

Results. In the first trial we used the simple heuristic approach for calculating the Bézier curve. For the low-level navigation we used abstraction AB1 and all experiences without filtering. Figure 4(a) shows the performance of this configuration. This routines does very poorly. It is practically always slower than our reference routine and it often differs from the desired goal angle by more than $90°$.

The second solution only differs from the first in the calculation of the Bézier curve. This time the parameters are learned from experiences. The result of this configuration is shown in figure 4(b). We see a slight improvement in accuracy, although it is still far from satisfactory. Similarly the programmed method still runs faster in almost every case, although the difference is smaller now.

In the third experiment we only used the fastest examples and a little programming was added, so that the robot turns at the beginning of the trajectory when the turning angle towards the goal point was near $180°$. This time the effect is more notable. Now most of the points are reached with an acceptable angle deviation. Furthermore the time statistics have shifted significantly. It is sometimes faster or not much slower than the reference routine.

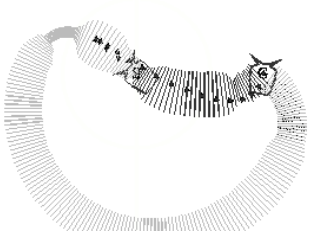


Figure 5: Trajectories of solution 2 (light gray) and 3.

The performance gain can also be seen when comparing the navigated trajectories. Figure 5 shows a navigation task performed by solution 2 and 3. Whereas the former can't follow the desired Bézier curve at all and takes a long way round, the latter approach follows the trajectory perfectly.

Finally we implemented a combined approach of programming and learning as described in section . As shown in figure 4(d), the performance almost reaches the level of the programmed one.

Discussion

Let us now discuss some of the issues of tightly embedding learning into autonomous robot control. (1) Section shows that aspects of autonomous robot learning can be specified in ROLL explicitly and transparently. Section shows that the specification of these aspects has a large impact on the performance of learned routines. In most learning robot controllers these aspects are adressed mainly implicitly or not at all. Turning them into pieces of code of the control program substantially improves the engineering methodolgy for learning robot conbtrollers. (2) The specifications listed in section are complete. Taken together they specify the complete learning process including experience acquisition, feature abstraction, monitoring of experience collection, parameterization of the learning algorithm. The learning problem specifications are therefore completely executable by the ROLL interpreter. (3) ROLL allows for the specification

and simultaneous experimentation with different variants of the learning problem. It thereby encourages and simplifies the experimentation and the empirical comparison of different variants. (4) We have also seen the seamless transition between, and a mixing of, learning and programming. Programmed code pieces can be simply substituted by specifications of learning problems that are also executable in the respective context.

In our view, these are critical functionalities that robot control languages must provide in order to allow us programmers to implement learning robot controllers for complex and dynamically changing tasks that can compete in terms of performance with their hand-coded counterparts. We believe that such programming language functionality is necessary to further promote the application of autonomous learning mechanisms in robot control.

Related Work

We are not aware of any work where aspects of learning problems are systematically changed and compared on the scale of our work. Empirical evaluation is an important issue in robotics. CLIP/CLASP (Anderson *et al.* 1995) is a macro extension of LISP, which supports the collection of experimental data and its empirical analysis. Other interesting approaches for the comparison of components in robot control systems are found in the work of Guttman and Fox [1998; 2002]. However, they only treat very selected and restricted aspects of robot control.

We use the language ROLL, because it provides most of the concepts we are interested in. There are several other programming language we have considered for this purpose, but that didn't quite satisfy our requirements. Thrun (Thrun 2000) has proposed CES, a C++ software library that provides probabilistic inference mechanisms and function approximators. Unlike our approach a main objective of CES is the compact implementation of robot controllers. Programmable Reinforcement Learning Agents (Andre & Russell 2001) is a language that combines reinforcement learning with constructs from programming languages such as loops, parameterization, aborts, interrupts, and memory variables. This leads to a full expressive programming language, which allows designers to elegantly integrate actions that are constrained using prior knowledge with actions that have to be learned. None of these projects addresses the problem of better learning by acquiring and selecting the data used for learning.

Conclusion

Proper embedding and parameterization of learning mechanisms is a necessary precondition for successful robot learning. In this paper we have performed a case study that has supported this point. We have used ROLL, an extension of the robot control language RPL that allows for the explicit and transparent specification of learning problems, their embedding into robot control, and the parameterization of the learning mechanisms. Using ROLL we could make aspects of learning explicit that are typically neglected or only implicitly modeled in robot control. The parameters that we have controlled using ROLL include state space transformations, reformulations of the learning problems, filtering experiences, and reasoning about the performance of learning mechanisms. In our experiments we could enhance the learning performance significantly through adequate choice of the parameter settings.

We have also seen that the explicit specification of learning problems has additional benefits. The declarativity of ROLL's control structures has substantially improved the readability of the program and made the solutions to learning problems understandable. Thus learning problems can be carried over to similar problems or other robot platforms with minimal modifications. Parameterizations of learning problems can be compared easily, which leads to a faster development of high quality solutions.

We have further seen that a smooth interlinkage of classical programming and learning algorithms yields solutions that can neither be achieved by learning nor programming alone. With an integration of learning into programming, robot controllers can be developped more quickly and more robustly.

References

Anderson, S.; Hart, D.; Westbrook, J.; and Cohen, P. 1995. A toolbox for analyzing programs. *International Journal of Artificial Intelligence Tools* 4(1):257–279.

Andre, D., and Russell, S. 2001. Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems*, 1019–1025. Cambridge, MA: MIT Press.

Beetz, M.; Schmitt, T.; Hanek, R.; Buck, S.; Stulp, F.; Schröter, D.; and Radig, B. 2004. The agilo robot soccer team experience-based learning and probabilistic reasoning in autonomous robot control. *Autonomous Robots*.

Beetz, M.; Kirsch, A.; and Müller, A. 2004. RPL-LEARN: Extending an autonomous robot control language to perform experience-based learning. In *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*.

Galhardas, H.; Florescu, D.; Shasha, D.; Simon, E.; and Saita, C.-A. 2001. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of the 27th VLDB Conference*.

Gutmann, J.-S.; Burgard, W.; Fox, D.; and Konolige, K. 1998. An experimental comparison of localization methods. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Gutmann, J.-S. Fox, D. 2002. An experimental comparison of localization methods continued. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall.

Thrun, S. 2000. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. San Francisco, CA: IEEE.

Evaporating tasks during execution of dynamically controllable networks

Russell Knight
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, CA
russell.knight@jpl.nasa.gov

Abstract

We present an extended execution algorithm for executing plans represented using simple temporal networks with uncertainty. We presume that the network to be executed is dynamically controllable. Our extension allows for skipping tasks that can be shrunk to zero duration if subsequent tasks are ready to start execution. The asymptotic time complexity of the technique scales as a polynomial of the number of timepoints that could execute simultaneously.

Introduction

One advantage of formulating plans conservatively is that we can know a-priori that the execution will (or probably will) succeed. One disadvantage of this approach is that we waste time and resources. In practice, we often skip the execution of tasks that lead to a goal if the tasks are deemed to be superfluous. We present a technique for modeling "skippable" tasks and skipping these tasks during execution.

Simple temporal networks (STNs) [2] provide a rich framework to connect inter-related tasks, execute the tasks, and monitor the execution. Unfortunately, STNs presume that the constructors and executors of these task networks have control over the duration of the tasks being executed, which is not always the case. Simple temporal networks with uncertainty (STNUs) [12] increase the representational capability of STNs by including a labeling of those intervals that are uncertain (contingent). Previous work by [7] has shown that we can know that an STNU is executable if our executor only changes those durations that are not uncertain (free) for future tasks, i.e., the STNU is *dynamically controllable*. The execution strategy of [7] presumed that we wouldn't skip tasks—the technique we present here allows us to skip tasks by shrinking their associated durations to 0. This is implemented in the Mission Data System [3] software framework as part of the timepoint firing algorithm.

Importance

If an execution agent can incorporate runtime feedback, this can be used to optimize plan execution in two ways: 1) reduction of resource utilization and 2) reduction of make-span. We can avoid using resources during execution, thus redundancy can be built into a plan but also can be ignored if it is not required. Additionally, we can reduce the make-span of an executed plan with respect to the original plan. Both resource usage and make-span are useful metrics for plan quality, thus our technique provides for on-line plan optimization.

Preliminaries

Simple Temporal Networks with Uncertainty

A simple temporal network (STN) can be represented as a directed, edge-labeled graph $G = (N, E)$ with real-valued, edge-label functions l and u. The nodes of the graph represent *timepoints*. A timepoint refers to a specific, yet possibly unspecified, moment in time. The edges of the graph, along with the l and u labels, represent *temporal constraints* between timepoints. More specifically, the l and u values bound the duration allowed between two timepoints. (Figure 1 shows two timepoints and a temporal constraint pictorially.)

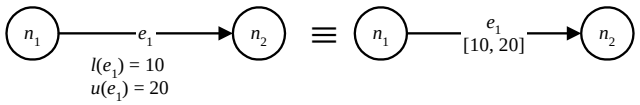


Figure 1 – Timepoint and temporal constraint representation

An *execution* of an STN is a real-valued, node labeling T of G that adheres to the temporal constraints. Thus, for all edges $e = (n_1, n_2) \in E$, $l(e) \leq T(n_2) - T(n_1) \leq u(e)$ implies that T is a valid execution. Checking for STN executability is computable in polynomial time [2]. (This is equivalent to checking consistency of the STN.) But, this assumes that we can know (and control) with certainty the duration of each interval represented by the temporal constraints before execution, which is not the case in many real domains. Thus we need to consider uncertainty for specific intervals.

A simple temporal network with uncertainty (STNU) is an STN with the addition of set $C \subseteq E$. Edges contained in C are temporal intervals that are determined by "nature" (but still adhere to the l and u constraints). We refer to such intervals as *contingent* (as Figure 2), otherwise the

intervals are referred to as being *free* (as Figure 3). Any execution strategy must accommodate the contingent edges. Thus, *T* for an STNU is partially defined by nature, and partially defined by us. There are a number of ways to characterize an STNU with respect to execution. An STNU is *strongly controllable* if we can devise a fixed valued *T* that accommodates all possible assignments by nature to *T*. (We learn nature's assignment after making our assignment.) An STNU is *weakly controllable* if for all possible assignments of *T* by nature an assignment of *T* by us is possible. (We learn nature's assignment before making our assignment.) An STNU is *dynamically controllable* if for all possible assignments of *T* by nature at execution time we can assign (or reassign) *T* for intervals not yet executed. (We learn nature's assignment during execution, and likewise make our assignment during execution.) Strong controllability [12] and dynamic controllability [7] are decidable in polynomial time, while the task of deciding weak controllability is co-NP complete [5].∉∈

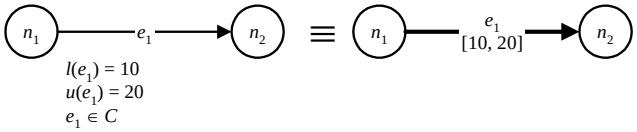


Figure 2 Contingent edge representation

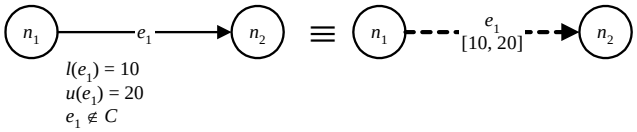


Figure 3 Free edge representation

This paper focuses on the execution of STNUs that are dynamically controllable. With this comes the requirement for *waits*—periods that are determined by nature and the result of computing dynamic controllability. *Waits* cause delays in normally free intervals to ensure the accommodation of the contingent intervals, thus a wait might make a free interval partially contingent. We return to the issue of waits later.

Concerning notation, we use *E* as the edge set of $G = (N, E)$ of the STNU representing the plan being discussed. Also, we presume that for all edges $e \in E$, $l(e) \geq 0$. Any unlabeled edge *e* in our figures is assumed to have $l(e) = 0$ and $u(e) = \infty$.

Plan Representation

A plan is a network of tasks and temporal constraints. A task consists of a temporal constraint $e = (n_1, n_2)$ in the STNU, the timepoints n_1 and n_2 associated with *e*, and the associated values represented by the functions $l(e)$, $u(e)$, and set *C*. We refer to n_1 as the *start-time* of the task, and n_2 as the *end-time* of the task. For simplicity, when referring to a task we will only refer to its associated edge in the STNU. This characterizes the temporal extent of the task but not its preconditions, in-conditions, or effects. To provide these, we include the additional condition-valued, edge-label functions *precond*, *incond*, and *effect*. While conditions have a rich history of semantics and representation, for our purposes a condition is a predicate *holds*(*x*) where *x* is a condition. *holds* considers in the current execution context whether or not a condition holds. If there exists a temporal constraint $e \in E$ that is not related to any task, *holds*(*precond*(*e*)), *holds*(*incond*(*e*)), and *holds*(*effect*(*e*)) are trivially true.

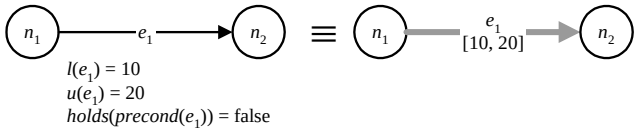


Figure 4 Task with unmet preconditions

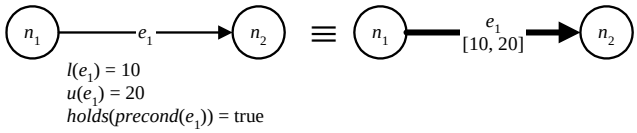


Figure 5 Task with met preconditions

Task Truncatability

To represent that a task is potentially *truncatable*, we employ the use of the edge-predicate function *truncatable*. Note that *truncatable*(*e*) implies that if *e* is being executed, then we can truncate its execution as long as doing so does not force the execution of a task for which its preconditions do not hold. We assume that all free intervals can be truncated. *truncatable*(*e*) being true for some contingent $e \in E$ (where $e \in C$) implies that although the completion time of *e* is under control of nature, its only purpose in the plan is to achieve the preconditions of the next step, and it may be truncated if these preconditions are already met. Thus *truncatable* provides us the semantic information we need to perform opportunistic execution.

Henceforth we will concern ourselves only with *truncatable* edges. Any edge $e \in E$ that is not *truncatable* is contingent, and only nature can determine the interval associated with it. We have no opportunities for skipping it and all possible intervals must be accommodated.

Approach

Execution Strategy

Our overall strategy for execution is to execute timepoints as early as possible while accommodating the contingent intervals. We intend to show how to skip tasks during execution, but first we describe the naïve execution

strategy. The naïve execution for task $e = (n_1, n_2)$ proceeds thusly:

(1) Wait until the start-time (n_1) of the task may be legally assigned to the current time (henceforth referred to as *now*).

(2) Wait until *holds*(*precond*(*e*)) is true.

(3) Assign $T(n_1)$ to *now*, i.e., execute all tasks that n_1 is the starting timepoint.

(4) Assume that *holds*(*incond*(*e*)) continues to be true until *exec*(n_2) is assigned (otherwise an error in execution has occurred and should be handled as an exception).

(5) Wait until the end-time (n_2) of the task may be legally assigned to *now*.

(6) Wait until all edges $x = (n_2, n) \in E$, *holds*(*precond*(*x*)) is true (all tasks for which the end-time of *e* is the start-time have satisfied preconditions).

(7) Assign $T(n_2)$ to *now*.

(8) Assert *effect*(*e*). (Note: effects are not handled during execution as execution is concerned only with preconditions; thus the function *effect* is not pertinent to our discussion and will be dropped.)

(9) Returning to the topic of *waits*, it is clear to see that *waits* can be implemented as part of the *precond* function. Since the system waits for preconditions to hold, each interval associated with the wait must be labeled as being contingent. It should be noted that, when combined, the *precond* and *holds* functions work as a monitor of the state of the world.

Because this framework is built around the dynamic controllability of the STNU, it will execute properly if no free interval is actually contingent and all interval bounds are obeyed by nature. We will assume that this is the case; otherwise some form of plan recovery would be required. Plan recovery is outside the scope of this work.

Simple disjunctive execution

To handle certain types of disjunctions of tasks, we take advantage of an ambiguity in the semantics of time with respect to the preconditions, in-conditions, and effects of tasks. Under normal circumstances, the minimum duration of any task is some real value $\varepsilon > 0$. But, what does it mean semantically when a task is of zero duration? Nothing in the STNU requires tasks to be of greater than zero duration. For our work, we say that a task (or temporal constraint or interval) of zero duration is *skipped*. Thus, if we have a series of possibly zero-duration tasks, all of which are free, then, under certain criteria, we can skip them all. The remainder of this paper describes: 1) under which criteria tasks can be truncated or skipped, and 2) how to tractably execute a plan with skipping.

Skipping Criteria

In general, we assume that we can skip tasks as long as, in the end, the following conditions hold: 1) we introduce no violations of the temporal constraints in the STNU, 2) all newly executing tasks have their preconditions fulfilled, and 3) all currently executing tasks have their in-conditions fulfilled.

We need to identify threats to skipping. Knowing whether or not a temporal constraint violation would occur during execution is handled using the algorithms of [7]. If a task is started with its preconditions fulfilled, it is assumed that its in-conditions will hold during its execution, (otherwise this is an execution-time error that would need to be handled as an exception.) Thus, the remaining threat to skipping is the possibility of a newly executing task not having its preconditions fulfilled.

If a timepoint n_3 comes after or is simultaneous to another timepoint n_1, then the assignment of *exec*(n_3) to *now* implies the assignment of *exec*(n_1) to *now*, thus forcing the execution of n_1. If n_1 has some outgoing edge $e = (n_1, n_2) \in E$ where *holds*(*precond*(*e*)) is false, then executing n_3 causes *e* to fail. Of course, we might be able to skip *e*, but then we would have to check the out-degree of n_2, etc., until we reached the end of our skipping opportunities. Thus, a threat to the execution of n_3 is caused by any edge $e = (n_1, n_2) \in E$ where the execution of n_3 implies the execution of n_1, n_2 cannot be executed due to temporal constraints, and *holds*(*precond*(*e*)) is false. If no such edge exists, then no threats exist, and n_3 can be executed.

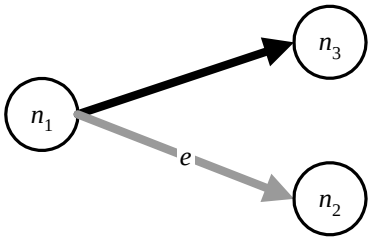


Figure 6 -- Timepoint n_1 threatens n_3

Threat propagation

We can see from the previous discussion that the source of threats is the frontier of the skippable tasks. The frontier of skippable tasks are those constraints or tasks whose start-times could be assigned to now, but whose end-times cannot be due to temporal constraints. Any task that is skippable is not the source of a threat per se—only tasks that are not skippable and for which their preconditions do not hold. It is also important to note that only tasks that are candidates to begin execution are sources of threats. Figure

7 gives an example of the frontier. Task (n_1, n_2) is executing; Tasks (n_2, n_3), (n_2, n_4), (n_3, n_5), (n_4, n_5), and (n_4, n_6) are skippable (they could execute now), and tasks (n_5, n_7) and (n_6, n_7) are at the frontier. (Note: remember that unlabeled edges are ordering constraints.)

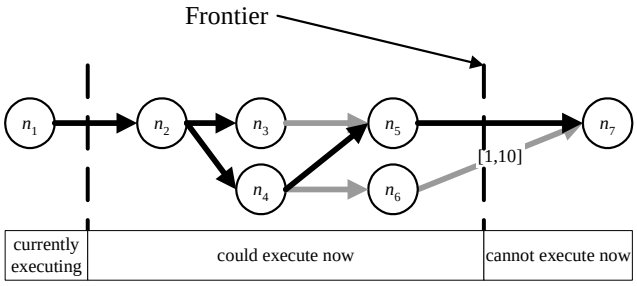


Figure 7 The execution frontier

Theorem 1: The only sources of threats are tasks at the frontier of the collection of skippable tasks.

Proof is by contradiction. Assume the contrary— all tasks at the frontier could execute but there exists a threat to execution in the set of tasks that could execute now. Therefore, there is a threat in the collection of skippable tasks, but all tasks e for which (**a**) the start-time is executable according to the STNU constraints and (**b**) the end-time is not, (**c**) *holds*(*precond*(e)) is true. Then, there must exist an e such that (**d**) its start-time is executable, (**e**) its end-time is executable, (**f**) *holds*(*precond*(e)) is false. We know **d** and **e** because this characterizes the rest of the executable tasks excluded by **a** and **b**. We know **f** because we presume a threat, and by **c** we know it is not at the frontier. But, if the end-time of a task is executable, it can be skipped, and the value of *holds*(*precond*(e)) is no source of a threat, but by our assumption it cannot be skipped. The only reason a task cannot be skipped is if it causes the execution of a start-time of a task e for which *holds*(*precond*(e)) is false. The entirety of the frontier can be executed (by **c**), thus any chain of skippable tasks leads to an executable task, leading us to a contradiction. □

Theorem 2: Threats propagate backward over a task e only when *holds*(*precond*(e)) is false.

Proof: We exhaustively list the alternatives. Consider the tasks $e_1 = (n_1, n_2)$ and $e_2 = (n_2, n_3)$. e_2 is a threat, thus *holds*(*precond*(e_2)) is false, and n_3 is not executable according to the constraints of the STNU.

Case 1: *holds*(*precond*(e_1)) is true and threats propagate backward. This is false because if *holds*(*precond*(e_1)) and n_1 is executable, then we can avoid the threat by executing n_1 and not executing n_2. Thus, threats do not propagate backward over tasks for which the preconditions hold.

Case 2: *holds*(*precond*(e_1)) is false and threats propagate backward. This is true because if we execute n_1 we are in error because *holds*(*precond*(e_1)) is false, and if we skip e_1 by executing n_2 we are in error because *holds*(*precond*(e_2)) is false. □

Figure 8 gives an example of the threat of (n_6, n_7) propagating backward to threaten n_4, but does not threaten n_2.

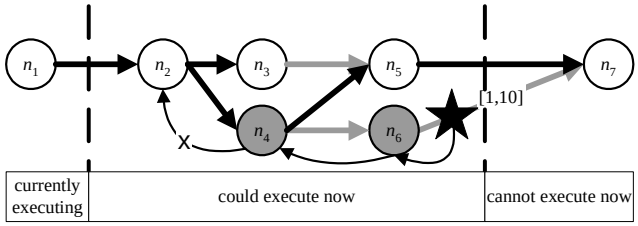


Figure 8 Threats propagate backwards

Theorem 3: Threats propagate forward over all tasks.

Proof is by induction on the number of interceding tasks between a threat and any subsequent executable tasks. Clearly, as our previous example illustrates, the base case of a task $e_1 = (n_1, n_2)$ that begins executing causes the task $e_{\text{threat}} = (n_1, n_{\text{threat}})$ to execute. Since e_{threat} is a threat (*holds*(*precond*(e)) is false and n_{threat} cannot be executed), but shares a start-time with e_1, executing e_1 would lead to the execution of e_{threat}, thus e_1 is also a threat. Inductively, consider an additional $e_m = (n_m, n_{m+1})$. The base case reveals itself at each previous e_x to e_m, leading to each e_x being identified as a threat, until $e_{m-1} = (n_{m-1}, n_m)$ is identified as a threat. The induction closes with e_m being identified as a threat following the same argument as for e_1. It should be noted that since all interceding tasks are skipped except for e_m, the preconditions of the interceding tasks have no effect. The preconditions of e_m also are not important, because even if *holds*(*precond*(e_m)) is true, executing n_m leads to the execution of n_1. □

Figure 9 shows threats propagating forward over (n_4, n_5) even though *holds*(*precond*((n_4, n_5)) is true. Note that propagation would continue to n_3 according to Theorem 2.

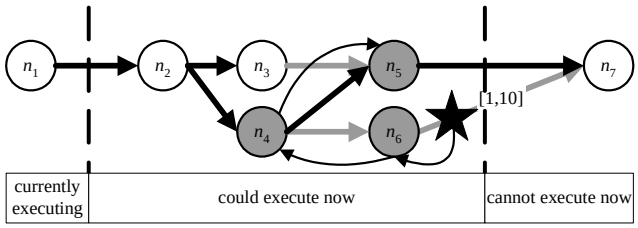


Figure 9 Forward propagation of threats

Threat Propagation Algorithm

Let us call the set of timepoints that can be executed now according to the constraints of the STNU *Candidates* $\subseteq N$. For all currently executing tasks $e = (n_1, n_2)$ such that $e \in C$ and *truncatable*(e) is also true and nature could assign $T(n_2)$ to *now*, we include n_2 in *Candidates*. Let us call the

set of edges that lie on the frontier of the candidates *PossibleThreats*. More formally, *PossibleThreats* $\subseteq E$ such that for each $e = (n_1, n_2) \in$ *PossibleThreats*, $n_1 \in$ *Candidates* and $n_2 \notin$ *Candidates*. Let *Threats* $\subseteq E$ be the set of edges that are known to be threats. We initialize *Threats* with every $e \in$ *PossibleThreats* such that *holds*(*precond*(*e*)) is false (Theorem 1).

We now propagate threats backward across each edge $e \in$ *Candidates* such that *holds*(*precond*(*e*)) is false (Theorem 2), adding each timepoint to *Threats*. We also propagate threats forward across each edge $e \in$ *Candidates* regardless of its preconditions (Theorem 3). This can be accomplished in time that is linear in the number of edges in *Candidates* using a simple reachability algorithm on a transformed graph. After propagation, we have the entire set of threats. We execute all timepoints in *Candidates* that are not start-times for edges in *Threats*.

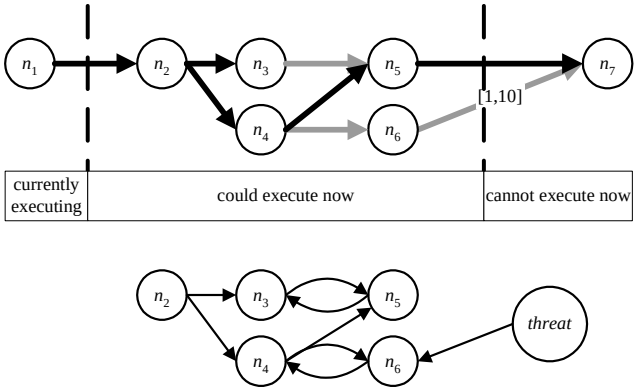


Figure 10 -- Reachability graph transformation

The reachability graph $G_R = (N_R, E_R)$ is a directed graph. *threat* is a node not in *N* from which threats originate. It is from this node that we will compute reachability. N_R=*Candidates* $\cup$ {threat}. For all $e = (n_1, n_2) \in$ *Threats* (as calculated above before propagation), add an edge $x = (n_1,$ *threat*) to E_R. For all $e = (n_1, n_2) \in E$ such that $n_1 \in$ *Candidates* and $n_2 \in$ *Candidates*, add *e* to E_R. If *holds*(*precond*(*e*)) is false, add (n_2, n_1) to E_R. Compute reachability from *threat*, which results in a set of nodes *X* that are reachable from the source node, *threat*. *Threats* is equivalent to *X*. The time complexity of computing reachability is $O(|N|\lg|N|+|E|)$, using Dijkstra's algorithm with a Fibonacci heap [1]. The space complexity is at most $O(|N|)$.

Examples

This system is deployed in a real-world plan execution system. The problems solved by its design are very much a function of the types of problems that needed solving according the to requirements of domain modelers. Some useful examples of meta-structures used by modelers include series preconditional satisfaction and parallel preconditional satisfaction.

Series preconditional satisfaction (SPS) is the idea that several tasks in series are used to achieve a goal, but if a precondition for any task later than the current executing task becomes satisfied, it is appropriate to skip all intervening tasks and execute the latest task that causes no threat. A specific example of this strategy is the "winnowing down" of a state requirement through a series of related tasks. For example, we might want to point a camera on a spacecraft towards a planet. This might require, in general, a series of operations that refine the pointing of the spacecraft. (See Figure 11.) But, if we are already opportunistically pointing at the planet, we might wish to skip steps to continue to our goal, (as in Figure 12).

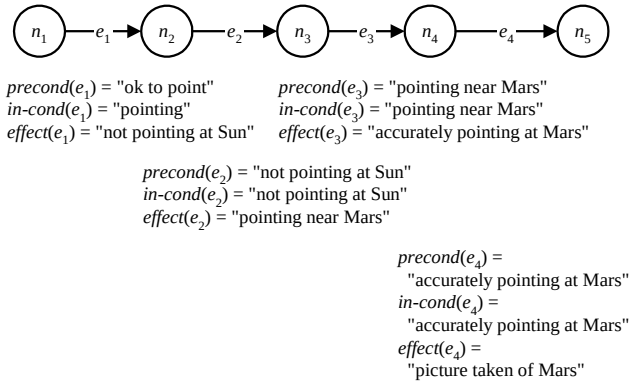


Figure 11 SPS example

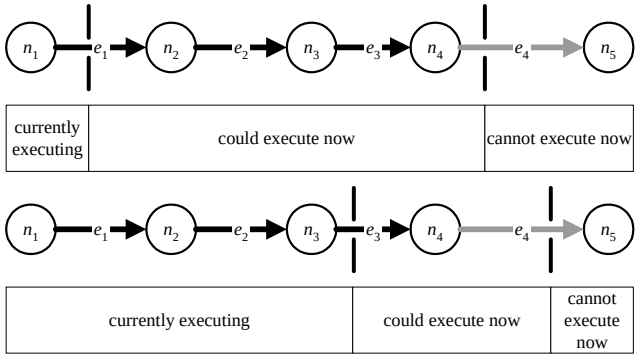


Figure 12 Execution of n_2 and n_3

Additionally, SPS can be augmented to include the idea that at least one of the members of the series must be executed by inserting a temporal constraint from the beginning of the series to the end of the series of $\in$ minimum duration, (as in Figure 13.)

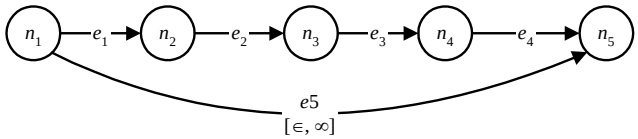


Figure 13 At least one task must execute

Parallel preconditional satisfaction (PPS) is the idea that several tasks in parallel are used to provide redundancy in hopes of satisfying a precondition. Once the precondition is met, the tasks might be truncated or skipped, depending upon whether or not the preconditions of each are satisfied. This allows one task that is ready to start to begin attempting to achieve the goal precondition while other tasks with the same general goal wait for their own preconditions to start. For example, we might want to ensure that the solar panels of a spacecraft are pointed to the sun. We have a number of ways of doing this, and we need only one way to succeed and thus will truncate any other executing tasks and skip pending tasks to continue on with charging the spacecraft. (See Figure 14 and Figure 15.)

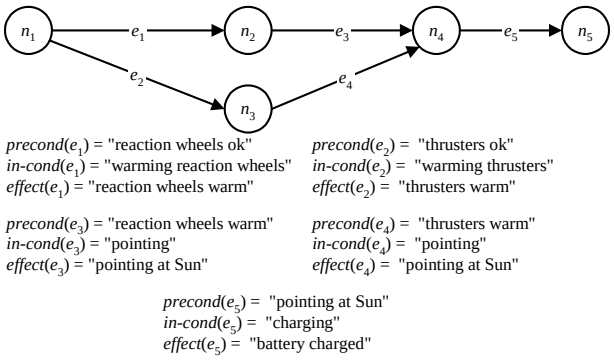


Figure 14 PPS example

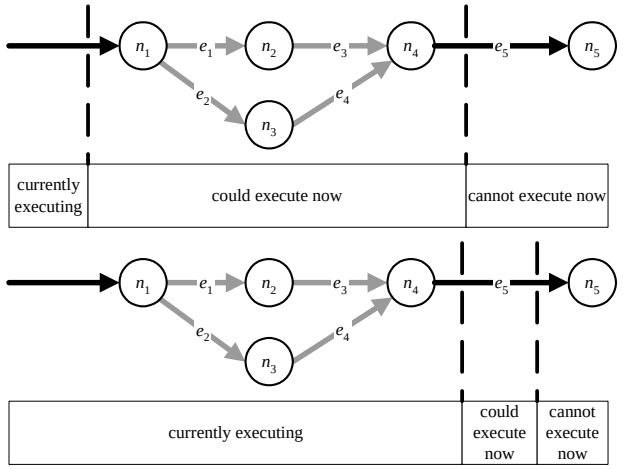


Figure 15 Execution example

Previous Work

The seminal work on STNs is [2]. Previous work in STN and STNU execution includes the work of [6] where efficient algorithms for managing propagation were given and the notion of network dispatchability was introduced. [12] and [5] introduced the notions of various types of controllability for STNUs and proved that determining strong controllability for an STNU is in P while determining weak controllability is co-NP complete.

[7] delivered the surprisingly tractable algorithm for determining dynamic controllability and executing a dynamically controllable STNU. [11] is extending this work to planning with shareable resources by handling some aspects of task sequencing at execution time (on-line).

[9] provides a description of conditional simple temporal networks (CSTNs). Even though we provide a weak form of conditional execution, our plans are not conditional in that all tasks are possible in a single context, meaning that we would have only one context label in a CSTN. Also, our work focuses on the execution of such plans as opposed to the verifiable construction of such plans.

Conclusions

We have described a tractable technique for reasoning about certain disjunctive conditions while executing a plan in metric time. We have presented a plan representation and an algorithm that, when combined with other existing algorithms, provides safe execution. We have provided as examples some useful STNU topologies from real-world examples. Our framework is deployed as part of the Mission Data System [3].

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] Corman, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms*. MIT Press, 1996, pg. 430

[2] Dechter, R., Meiri I., and Pearl J., "Temporal Constraint Networks," *Artificial Intelligence*, 49, 1991, pp 61-95.

[3] Knight, R., Chien, S., Starbird, T., Gostelow, K., and Keller, R. "Integrating Model-based Artificial Intelligence Planning with Procedureal Elaboration for Onboard Spacecraft Autonomy." *SpaceOps 2000*, Toulouse, France, June 2000.

[4] Laborie, P., Ghallab, M., "Planning with Sharable Resource Constraints," Proceedings IJCAI-95, 1643-1649

[5] Morris, P., and Muscettola, N., "Managing temporal uncertainty through waypoint controllability." In T. Dean, editor, *Proceedings of the 16th International Joint Conference on A.I. (IJCAI-99*), pages 1253–1258, Stockholm (Sweden), 1999. Morgan Kaufmann.

[6] Morris, P., Muscettola, N., and Tsamardinos, I., "Reformulating temporal plans for efficient execution." *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-98*), Trento (Italy), 1998.

[7] Morris, P., Muscettola, N., and Vidal, T., "Dynamic control of plans with temporal uncertainty." *International Joint Conference on A.I. (IJCAI-01*), Seattle (WA, USA), 2001.

[8] Muscettola, N., Nayak, P., Pell, B., and Williams, B., "Remote Agent: To Boldly Go Where No AI System Has Gone Before," Artificial Intelligence 103(1-2):5-48, August 1998.

[9] Tsamardinos, I., Pollack, M., and Horty F., "Merging plans with quantitative temporal constraints, temporally exetended actions, and conditional branches," *Artificial Intelligence Planning Systems*, 2000.

[10] Vidal, T., "Controllability characterization and checking in contingent temporal constraint networks." In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR2000*), Breckenridge (Co, USA), 2000. Morgan Kaufmann, San Francisco, CA.

[11] Vidal, T. and Bidot, J., "Dynamic Sequencing of Tasks in Simple Temporal Networks with Uncertainty." In the Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP2001), Paphos, Cyprus, 2001. Springer-Verlag.

[12] Vidal, T. and Fargier, H., "Handling contingency in temporal constraint networks: from consistency to controllabilities." *Journal of Experimental & Theoretical Artificial Intelligenc*e, 11:23–45, 1999.

Unified Planning and Execution for Autonomous Software Repair

Richard Levinson

Attention Control Systems, Inc.
650 Castro Street, Suite 120, PMB 197
Mountain View, CA 94041
rich@brainaid.com

Abstract

This paper addresses the need for more flexible autonomous systems that detect and correct software failures at runtime. We present new methods for integrated planning and execution that enable runtime verification and repair for software failures, and explore the related issue of integrated procedural and declarative action representations.

We present a library for embedding declarative planning methods within procedural C++ code. The library provides an interface to supervisory processes, which monitor software execution and provide last-resort error recovery after pre-programmed error handlers fail. The library provides an interface to search and temporal constraint engines maintained by the meta-processes. The planner and controller use the same procedural representation in order to share context (computational state) between planning and execution.

Motivation

Limited Autonomy. Today's autonomous systems provide more coverage for hardware failures than software failures. If they cannot represent and reason about software failures, they are doomed to blind spots and will have limited autonomy. There are several reasons why software failure cannot be avoided including: limited time and information at design time, limited time and resources for running test cases, changing operating conditions and changing mission requirements. Our goal is to include more software within the scope of recovery and develop autonomous systems that repair their own software.

Limited Architectures. Currently, most system software is outside the scope of plan-based recovery because it is not written in the planner's modeling language. To increase model scope, the planner and controller must share computational state information about failure and repair contexts. This is challenging because integrated planning and execution traditionally involves translating between planners that use declarative languages and controllers that use procedural languages. Major problems caused by this *language barrier* include:

Redundant & Low Fidelity Models. There is a need to develop and maintain a declarative model of the execution system using a planner's modeling language. The planner's model is redundant with the execution "model" (i.e. the software). The planner's model of software is low fidelity since much of the computational state is hidden in a black box model of software. The redundant planning model may not match the actual execution software, and it is difficult to maintain the model and the code in parallel.

Information loss is nearly guaranteed when translating between the controller's procedural language and planner's declarative language, which are typically developed independently and optimized for different purposes. This information loss reduces the planner's understanding about execution context, and vice versa.

System complexity is increased because of the need to translate between the two languages. There is an increased need to develop ad hoc translators between the two languages, and a lot of the integration effort is devoted towards accommodating specific language differences.

Architecture Overview

Our approach to increasing the scope of failure recovery and removing the language barrier is called the **Program Planning and Execution Language (Propel)**. Propel provides a library for integration between C++ and supervisory processes that detect and correct software failures. The library includes new methods for integrating search and temporal constraints within C++ applications.

We also present new methods for integrated planning and execution with improved computational context switching and sharing capabilities compared with existing methods. These new methods allow us to increase the scope of error handling to include the system's software infrastructure. Additionally, the planner and controller's action representations are unified so more context information can be preserved during transitions between execution and planning.

Figure 1 shows the three different process levels within Propel: The *Application*, *Supervisor*, and *Executive* levels. The Application level contains all of the domain-specific processes. The Executive and Supervisor levels are meta-processes (they monitor and manipulate the application processes) in order to detect and correct software failures.

The **Application level** executes C++ code extended with the Propel Library Interface to meta-level supervisory processes. The code contains declarative statements that interface to search and temporal constraint engines.

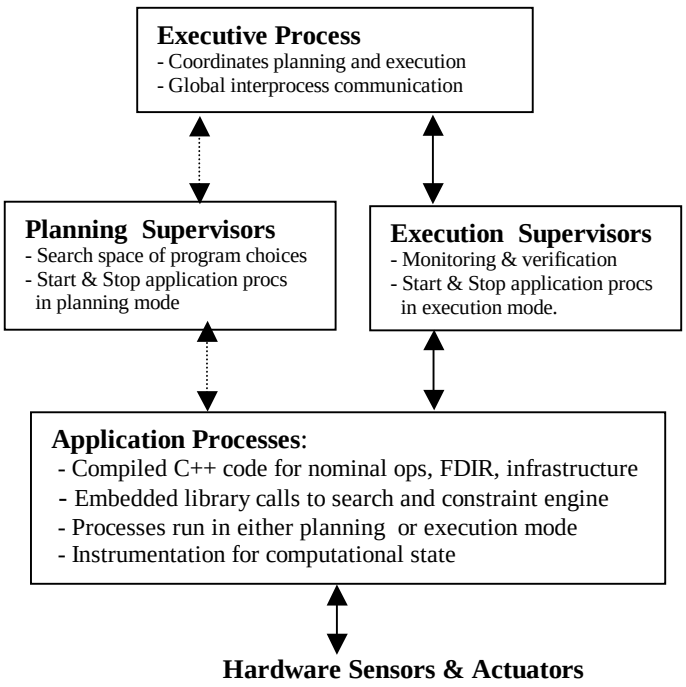


Figure 1: Propel's three levels of processes: *Application, Supervisor, and Executive.*

Choice points embedded in the code describe alternative operations and resources, and define a space of program variations. The Application level has information only about its own execution environment, like any other C++ process. The application level sends status messages, computational state information, and queries to the meta-level supervisors, and are started and stopped by the supervisors.

The **Supervisor level** processes perform runtime monitoring,, verification, and recovery of application level processes. They manage search spaces of application level processes. One planning supervisor and one execution supervisor is created for each parallel process in the application. The planner provides a last-resort failure handler after all application-level fault protection has failed. The term *"planner*" refers to a Planning Supervisor and the term "*controller"* to refer to an Execution Supervisor.

The Planners and controllers start, monitor, and stop the same compiled application-level code in the same environment. The difference is that the planners run the application level code with a _PLANNING_ flag true but the controllers run the code with that flag false. Application code accesses that flag to determine if it is running in planning (simulation) or execution mode.

Application code may execute different branches depending on the status of this flag. Subroutines which send out physical actuator commands may simulate the commands when the planning flag is true instead of sending out actual actuator commands. The only procedures that require simulation are those that interact directly with hardware actuators because we want to block the actuator commands during simulation, however, any procedure can be simulated in order to skip details. All of this requires runtime simulation, which may involve models of various levels of fidelity. The simulation branches may also include choice points and probabilities so that high probability variant outcomes may be simulated. Application processes in planning and execution mode communicate via rules, called Situated Control Rules (SCRs) [Drummond 1989], that provide *choice point advice* about what to do at the choice points embedded in the code.

The **Executive level** is a single process that manages transitions between planning and execution and manages databases that enable inter-process communication and synchronization. This involves sending messages to the supervisor processes. The executive also handles requests for information from the application level.

Three levels but not 3T: Although Propel has three levels, it is significantly different than the 3T system [Bonasso, et., al., 1997]. In 3T, each level encodes domain-specific activities, but in Propel all of the domain-specific activities are encoded only at the application level. Hierarchy within the application level is accomplished through standard C++ programming methodology. Propel's other two layers (Supervisor and Executive) are meta-processes that monitor and manipulate the application level.

Example

Figure 2 shows the application level source code for an example that we will use throughout the paper. The example includes two parallel processes: *Camera* and *Wheels*. The wheels iteratively move to a target while the Camera takes pictures until Camera detects that the wheels have stopped moving, at which time it takes a close-up picture. When the Wheels process arrives at the target, it adds the fact "Not Moving" to the database, which signals to the waiting Camera process that it can proceed to take the close-up. Line 58 shows use of the _PLANNING_ flag to determine if a command should be simulated. Only the wheels portion is described here due to limited space. See the online version of this paper for further explanation of how this failure is handled (Levinson 2005).

Search Space

To recover from software failures, Propel searches through a space of program variations defined choice embedded in the code. The application-level interface to the search engine includes choice points, fail statements, heuristics and meta methods. The choices identify alternative subroutine calls and assignment statements.

Figure 3 shows the Process tree, where each node is a Unix Process. The tree illustrates the three process levels

```
 1 // Filename: rover.cpp
 2 #include "propel.h"
  ---------------------------------------------
 3 void initializePropel ()
 4 {declare_Process(Wheels);
 5  declare_Process(Camera);
 6  declare_Goal(Move_1, "Move ?d");
 7  declare_Goal(Move_2, "Move ?d");
11  declare_Heuristic(preferFast);
13  declare_Heuristic(preferShort);
16  declare_TP("Not_Moving [1 30]");}
  ---------------------------------------------
17  void Wheels ()
18   {start_Task([5 10] Wheels {1 80} [6 100]);
19    gotoLoc(getTarget());
20    add_Fact("Not Moving");
21    execute_TP  ("Not_Moving", 1, 30);
22    if (!wait_Task ("[0 20] waitForCamera {0 10}
          constraint after Camera End [0 5]"))
23      fail ("Wheels Timeout");}
  ---------------------------------------------
24 void gotoLoc(Location* loc)
25  {start_Task(gotoLoc {0 30});
26   int d, directions[] = {N, S, E, W};
27   int step = 0;
28   _Var(step);
29   while (!atLocation(loc))
30     {d = (int) choose_item(directions, 4,
                "preferShort(?loc)", _Var(loc))
31      choose_task("Move ? :preferFast()", _Var(d));
32      require(step++ < MAX_STEPS);}}
  ---------------------------------------------
48  void Move_1 (void* direction)
49   {Direction dir = *(int*)direction;
50    if (dir == E) agent = NULL; //Fault Injection
51    int x = agent->x;
52    int y = agent->y;
53    switch(dir)
54      {case N: y++; break;
55       case S: y--; break;
56       case E: x++; break;
57       case W: x--; break;}
58    if (_PLANNING_) Simulate_Command("GoTo", x, y);
59    else Execute_Command("GoTo", x, y);}
```

Figure 2: **Application-level source code** for the Wheels process with embedded library calls (in bold text) to search and constraint engines. (The parallel Camera process is not shown due to space limitations)

shown in Figure 1. The root node in Figure 3 is the Executive process, the Supervisor processes are the second level, and Application level processes are everything below the second level. Subtrees below the Supervisor processes are search trees, where nodes correspond to application level processes and arcs represent branches at the embedded choice points. The application level processes execute local heuristics to sort the choices using information available at the application level. Global heuristics are used by the SupervisorFailure method for backtracking.

Choice points. Choice points identify alternative operations and resources. Lines 30 and 31 show examples. There are three types of choice points: *choose_item* is a non-deterministic assignment statement, *choose_task* is a non-deterministic subroutine call, and *choose_fact* is a non-deterministic database query. They are non-deterministic because executing them will have different outcomes based on heuristics and planner results.

When a choice point is executed by an application level process, that process calls fork() to create a child process that continues with the selected choice. The parent process remains suspended until backtracking occurs, and then the supervisor may wake up the parent to generate a new choice (fork a new child). Heuristics and other user-specified methods are used for search control.

Figure 3 shows the search space for our example. The nodes at the application level represent computational continuations resulting from forks at choice points Each choice point defines a set of disjunctive branches in the search tree. One branch is created for each choice that is tried. Only choices that are actually tried cause new processes to be created.

`void* choose_item()` - This statement will choose an element from a list of integers or pointers and functions as a non-deterministic assignment statement. The choices are sorted by the specified heuristic function. The <var> are passed into the heuristic. The return value is type void* and points to the object pointer selected from the list.

The example on line 30 says: choose a direction from the array containing all directions (N,S,E,W are C "enums" that map to integers). The heuristic "preferShortest" is used to sort the choices in order of minimum "manhattan distance", and the heuristic takes a target location as input. Heuristics are declared at initialization (lines 10-13).

`choose_task()` - This statement will choose a subroutine that matches the given goal pattern, and functions as a non-deterministic subroutine call. It calls one of several methods that all achieve the same goal pattern. The example shown on line 30 will choose a task that matches the pattern "move ?" where ? is a variable bound to the move direction. This matches the goal declarations from lines 6 and 7, which says that either Move_1 or Move_2 could be called to achieve this goal.

The "?" is a place holder for the goal pattern variables There must be one goal pattern variable provided for each ? in the pattern. This is similar to the way *`printf("%d %d", i, n)`* requires a var for each %. The goal pattern variables are passed by reference so execution of a task that matches the pattern can change the value of the vars. The heuristic is named on the right of the colon, so the example in line 31 says use heuristic "preferFastest".

`choose_fact()-` This statement will choose one fact from the database that matches the given fact pattern, and functions as a non-deterministic database query. It queries

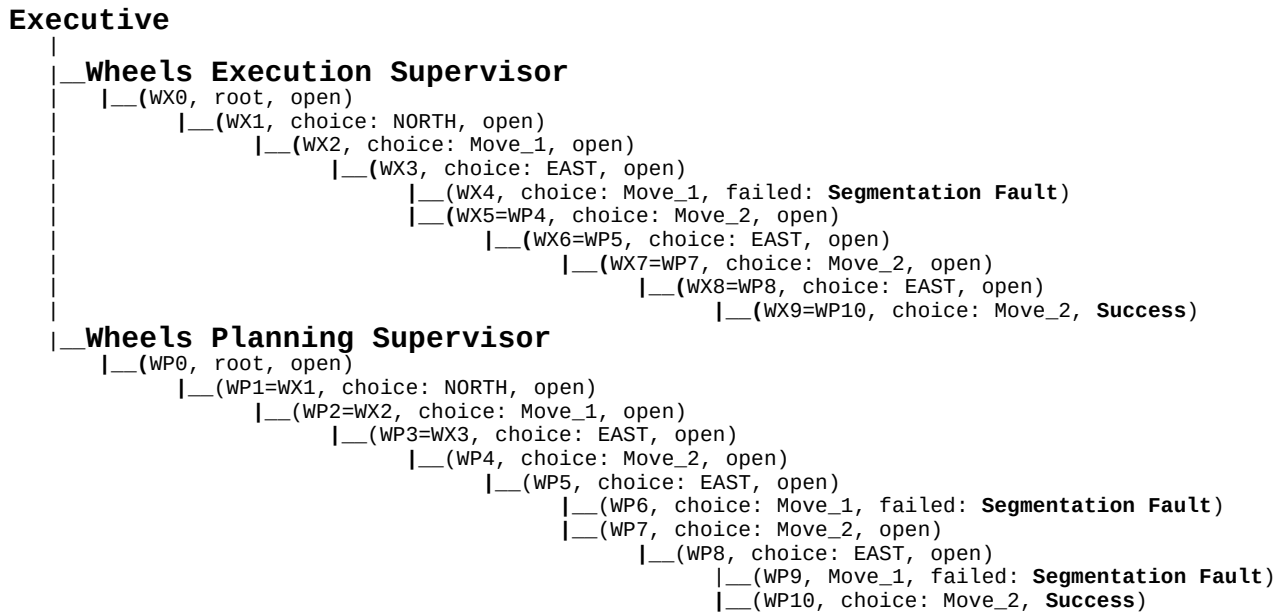


Figure 3: Process Tree and Search Space. The levels of the tree correspond to three levels shown in figure 1. Each node is a process with unique PID. The top level process is the Executive, Level 2 contains supervisors that monitor and manipulate the application-level processes. All nodes below the Supervisors are application-level processes running code defined in figure 2. Each subtree below a supervisor (starting with "root" nodes WX0 and WP0) is a search space with branches that are disjunctive choices.

the database for facts that match query pattern and returns one of the matching facts. The Executive maintains a database that can be used for interprocess communication between concurrent application processes (such as Wheels and Camera). The choose_fact() statement chooses bindings via unification with the DB.

Fail and Require statements. Application level code may contain fail and require statements which cause search node failures and may trigger backtracking. These statements provide runtime verification capabilities by detecting when requirements are violated. Lines 23, and 32 show examples of *fail* and *require* statements.

The **`fail()`** statement causes the current process to inform its supervisor process that it failed, and then the process is suspended. The supervisorFailure() and executiveFailure() message handlers may be designed to handle the failure by transitioning from execution to planning or by backtracking within planning space. The example in line 23 triggers failure if the wait statement in line 22 timesout.

The **`require()`** statement is similar to the C++ Assert statement except that Assert requires user intervention. `require`(condition) will test the condition, calls "fail" if the condition is false. The condition represents a runtime verification requirement. The example in line 32 triggers failure when the variable step exceeds MAX_STEPS.

Other failure types include exhausted choice points, unhandled exceptions, and temporal constraint violations. Any statement may trigger an implicit **fail** statement because any subroutine call has the potential to cause a segmentation fault or floating point exception. Any choose statement can trigger a failure when all choices have been exhausted (or no choices exist). Wait statements trigger a failure if they time out. When any of these failures occurs, the search node marked "failed", the process is suspended, and the supervisor is notified.

Failure Example: We've injected a software failure into our example to show how it is handled. In our example, an unhandled exception occurs because one of the two move operation (`Move_1`) dereferences a null pointer when it tries to move East. Line 50 sets the variable "*agent*" to NULL when the direction is East. This causes the injected failure when the pointer is dereferenced on line 51. The segmentation fault is trapped by propel, and treated as if a fail statement had been executed by the failing process.

Search Control

Propel includes several methods to control search. The first and most important is the *sparse search space*. The search space is sparse because it only has branches at choice point locations. Deterministic subroutine calls like `gotoLoc()` (line 24) are not represented in the search space. Most the application-level code can be deterministic with search triggered only at explicit choice point locations.

Situated Control Rules [Drummond 1989; Drummond et. al, 1993, Levinson 1995] are the second most important search control method. The planner and controllers communicate by exchanging condition-action rules called Situated Control Rules (SCRs). SCRs provide

```
Situated Control Rule:
IF <condition> Then <action>

<condition> = (<StackFrame>+)
<StackFrame> = (Frame: <subroutineEntryPoint>,
                PC <programCounter> <var>*)
<subroutineEntryPoint> =
"subroutineName:lineNumber"
<programCounter> = "filename:lineNumber"
<var> = (Var: <varName> <varAddress> <varValue>)
<action> = (<choice> <mode> <status>)
<mode> = Planning | Execution
<status> = success| failure | open

RULE WX4
IF ((Frame: gotoLoc:rover.cpp:24, PC rover.cpp:31
      (Var: step 0xffbec9d0 0003)
      (Var: d 0xffbec9d4 0003))
    (Frame: Wheels:rover.cpp:18, PC rover.cpp:18))
THEN Move_1 Execution failure

RULE WP10
IF ((Frame: gotoLoc:rover.cpp:24, PC rover.cpp:31
      (Var: step 0xffbec9d0 0005)
      (Var: d 0xffbec9d4 0003))
    (Frame: Wheels:rover.cpp:18, PC rover.cpp:18))
THEN Move_2 Planning success
```

Figure 4: Situated Control Rule (SCR). Each branch in the search space is described by an SCR. The <condition> is the control stack capturing the computational state when the choice is made, and <action> is the choice.

choice point advice that describes preferences at choice pointes. They describe the context and outcome of prior choices made during planning and execution. These rules are the key to context sharing between planning and execution. Figure 4 shows the grammar and examples of SCRs.

One SCR is defined for each branch in the search tree shown in Figure 3. The name of the SCR is the name of the search node in that tree. The condition (left-hand side) of the rule is the control stack when branch occurred, and the action (right-hand side) is the choice associated with that branch. An SCR says: "If the rule's *Condition* part matches the current process' control stack, and the choice outcome was not failure, then select choice (continue the process associated with choice).

In Figure 4, the control stack for Rule WX4 describes the execution context at search Node WX4 (shown in Figure 3). Rule WX4 says: The Wheels procedure was entered at line 18 of file rover.cpp (Figure 2) and then it called subroutine gotoLoc at line 24 of file rover.cpp, and the program counter (PC) says it was at line 31 when the branch occurred (at a choice point). The address and value for gotoLoc()'s local variables *step* and *d* are also recorded in the control stack. This rule will apply only when local variable *step* = 3.

This failure occurred because gotoLoc() called Move_1, which caused a segmentation fault (lines 50-51). After replanning, the executing process will look for planner advice about which choice to take. Rule WP10 says that the planner found a successful plan with choice "Move_2" when the program control stack was in the given state. The rule for node WX4 is used to tell the planner that the controller failed inside the task `gotoLoc()`. The planner uses this to avoid simulation of the "Move_1" choice until it has explored other tasks that achieve the same goal.

SCRs are collected when transitioning between planning and execution and vice versa. Collecting SCRs is similar to classical goal regression and form a partial policy. When a search success or failure occurs, SCRs are collected for the path from the given node to the root node. For example, when the first Wheels excution failure occurs at node WX4 in the tree above, SCRs are collected that describe the control stack, choice, and outcome for each node between WX4 and the root WX0.

The rules are passed from the controller to the planner so that the planner understands what choices the controller took. When the planner finds a workaround, similar rules will be collected from the planner's search tree, and used to tell the controller which choices led to a successful plan. When an application level process executes a choice point, SCRs are combined with heuristics to sort the choices.

Heuristic functions: Users can define *local heuristics* as preference functions that are called to sort choices locally at choice points. The heuristcs can use "less-than" predicates and the built-in function "SortChoices" to reorder the choices at the choice point. For example the following heuristic sorts choices into increasing values. Users can also define *global heuristics* by modifying the search function, **`BestNode`**(), which may control search using global information not available at the application level. BestNode() is called by the supervisors to determine which application level node (process) should be explored (continued) next. Computational state information about application-level processes and the global database can also be used to implement different search strategies.

The default implementation of BestNode is similar to Reaction-First Search (RFS) [Drummond et. al., 1993]. The search is biased to first explore the controller's default behaviors (reflexes defined by heuristics) to see if the controller's "reactions" will work in the current situation. When the default reaction is inappropriate, the planner generates advice rules to override the default reactions.

Supervisor and Executive Interface

The application level interface to meta processes includes the methods described below. We describe the default behaviors for these methods but users can write their own versions to customize the coordination of planning and execution.

The **SupervisorFailure**() method is called when a Supervisor receives a *SupervisorFailure* message from the application level. In planning mode it triggers backtracking, otherwise it suspends execution and then informs the Executive by sending an *ExecutiveFailure* message. The **SupervisorSuccess**() method is called when the Supervisor receives a *SupervisorSuccess* message from the application level. It typically informs the Executive by sending an *ExecutiveSuccess* message.

The **ExecutiveStartup**() method is the Executive level startup handler called at startup time to initialize concurrent processes and propel structures, including the initial temporal constraint network. This method may start execution before planning or vice versa, or it may run them concurrently, depending on application.

This **ExecutiveFailure**() method is called by the Executive when it receives an *ExecutiveFailure* message from a Supervisor. The Executive knows whether it was a planning failure or an execution failure. In planning mode, this event indicates an exhausted search space.

The **ExecutiveSuccess**() method is called when the Executive receives an *ExecutiveSuccess* message from a Supervisor. This happens when application level process has an empty control stack during execution, or in planning mode when the subroutine which called the failed subroutine is popped off the stack.

These meta-level handlers allow users to specify different *executive strategies* including: a) predictive detection which simulates a program prior to execution, b) reactive detection where you only call the planner after an execution failure occurs, c) anytime planning which stops planning at anytime and collects SCRs for the partial plan, d) batch planning which plans until complete solution is found, and e) incremental replanning which only fixes the current execution failure before returning to execution. Another executive strategy could have the planner generate SCRs for failure contingencies to "robustify" the SCR policy.

Search Process Walkthrough

To illustrate how this works together we will walk through the steps that produced the results shown in Figure 3. To save space, only the Wheels process is described here.

First, the initialization routine shown in Figure 2 is executed. This causes the executive and supervisors to be created along with application-level root nodes WX0, WP0. The root nodes are suspended after creation and the `executiveStartup()` method is executed, which in our case tells the Supervisors to start executing the Wheels process by activating root nodes WX0.

The Wheels execution proceeds through the choice points in the gotoLoc() routine. A branch in the tree is created for each choice point executed. Wheels proceed and generate the nodes WX1, WX2, WX3, and WX4. The node WX4 represents a process that threw a segmentation fault when it executed Move_1 in the East direction. When WX4 calls `fail`(), the Wheels Execution Supervisor is informed, and calls its `SupervisorFailure()` method which may do different things depending on whether it is a planning or execution failure. In this case the Supervisor informs the Executive of the failure and passes the SCR's that describe the failure context.

The Executive's `executiveFailure()` method is called and the Executive informs the Wheels Planning Supervisor to start executing the wheels procedure (in planning mode), and it passes the execution SCRs to the planning Supervisor where they are used to guide the search process down the same path as execution. This guidance can be seen in nodes WP1-WP3, where the nodes are labeled with the name of the execution SCR that was used by the planner. For example WP1=WX1 means that the choice taken by the planner at node WP1 is based on the SCR that defines the branch for execution process WX1. This rule will only apply the first time line 31 is executed because the step variable = 1.

The planner follows the execution SCRs through node WP3 (notice that WP1-WP3 have equals signs). WP4 does not follow the execution choice because that is the choice that led to failure in WX4. This rule, which identifies a choice that led to failure, is a *rule of avoidance*, and treated differently than other rules. SCRs that identify failure choices will cause that choice to be preferred last. It will be chosen only after all other choices failed. The planner delays the execution choice (**`Move_1`**) to the end of the search space since it is known to have failed. The planner chooses the next option, which is **`Move_2`** (node WP4), which does not fail when moving east. The remaining planner nodes (WP4-WP10) are not guided by SCRs because execution never got that far. The planner chooses the failing **`Move_1`** two more times in simulation (nodes WP6 and WP9) before reaching success (empty control stack) in at WP10.

The successful application process WP10 then informs the Wheels Planning Supervisor about the node success and the Supervisor informs the Executive, which executes its `ExecSuccess()` method, and collects SCRs for the path from node WP10 to node WP1. The Executive then informs the Wheels Execution Supervisor to continue execution using the planner's SCRs as choice point advice.

The Wheels Execution Supervisor resumes execution by expanding node WX3 as determined by `BestNode()`. Since WX4 failed, its parent WX3 is the chronological backtracking choice, but it has been suspended since it spawned child WX4. After being reactivated by the Supervisor, Node WX3 generates a new child WX5 based on the SCR from the planner's WP4 node. This instructs the application code to choose `Move_2` instead of `Move_1`. The remainder of the Execution processes (nodes WX5 - WX9) are guided by the planner's SCRs as shown by the equals signs next to the node names.

Temporal Constraint Engine

Propel uses a Simple Temporal Network (STN) [Dechter et al., 1991] to monitor and control C++ program execution based on a declarative temporal model. As the propel application executes, progress is shadowed by the STN. The STN can be partially declared before the C++ applications are started but the network will also be dynamically generated as the C++ code executes.

As execution proceeds, timepoint values are constrained by actual execution times. When a subroutine is called, the STN is checked to see if it is ahead or behind

schedule. If it enters the subroutine early, then it waits. If it is late, then fail() is called and the Supervisor is notified. Static STN statements declare "background" parts of the temporal network that are defined before any tasks start execution, and dynamic statements extend the network dynamically and conditionally during task execution. The following declarative statements can be embedded in C++ to modify the temporal constraint network.

`start_Task()` - Lines 18 and 25 show examples of the start_Task statement, which declares temporal constraints on a C++ function. It creates two timepoints in the temporal network. One timepoint represents the start time of the function and another represents its end. The Task_Start on line 18 declares that Wheels has the following temporal constraints: Start Time in range[5 10], End Time in range [6 100], duration range {1 80}.

`wait_Task()` – wait for a fact to be added to the database, or for temporal constraints to be satisfied (w/ timeout). Lines 22 and 23 show how the wait_Task statement and the `fail` statement can be combined. Line 22 says that the program should start waiting between time 0 and 20 and wait for a maximum of 20 seconds and wait for between 0 and 5 seconds after the Camera program ends. If the maximum duration of 20 seconds is reached before the camera program ends, then `fail`() is called.

`declare_TP()` adds a new Timepoint to the temporal network when it is executed within a C++ function. Other processes may share constraints with TP. Line 16 illustrates the `declare_TP` statement. When this line is executed, a time point is created with lower bound of 1 and an upper bound of 30, so the timepoint must be executed sometime between time 1 and time 30. Other processes can establish constraints to this node using by referring to its label, which is "Not_Moving" in this example. The new timepoint is not actually executed (collapsed to a singleton) until either start_task() or execute_TP() is executed which refers to the same timepoint label.

`execute_TP()` is the same as declare_TP except it also executes the timepoint. This means that the value of the time point is collapsed to a singleton (the current time) and that time is propagated through the temporal network. Line 21 shows an example of execute_TP().

Related Work

Propel 1 [Levinson, 1995; Levinson 1994]. This paper presents Propel 2, which is very different from Propel 1 because it uses compiled C++ as its action representation. In order to accommodate the fact that the application code is compiled, the Propel 2 architecture shown Figure 1 differs significantly from Propel 1's architecture.

Propel 1 had only one planner and one controller, which interpreted the same LISP action representation using their own instruction fetch-execute cycles. In Propel 2, multiple planners and controllers start and stop compiled application code and they don't have instruction fetch-execute cycles. Propel 2 extends Propel 1 by adding temporal constraints. It also extends SCRs to represent the state of compiled C++ code and to include rules of avoidance. Since Propel 2 enables planning to be embedded in C++, it is better suited for use in deployed autonomy applications.

ERE [Drummond, et. al, 1993]- Propel is a direct extension of ERE (the Entropy Reduction Engine). Propel incorporates several ERE features for integrated planning and execution including Reaction First Search and SCRs. Since ERE was never used to model software actions, a key difference is that Propel uses a C++ action representation compared to ERE's STRIPS-like action representation.

IDEA [Muscettola et. al, 2000] The IDEA system is a unified planning and control system like Propel. However, IDEA executes the *planner's* language while Propel plans with the *controller's* language. IDEA's controller executes plans by interpreting the planner's *declarative* language. IDEA models software as black boxes and does not distinguish between a hardware or software black box. It can detect unexpected (software) inputs, outputs, and timing, but has a minimal model of the logic and computational state details relating the inputs to outputs.

KIRK/RMPL – [Kim, et. Al, 2001] William's KIRK/RMPL system also provides a unified approach to planning and execution. It differs from Propel because it compiles procedural constructs into a declarative model which is then interpreted by during execution. KIRK is similar to IDEA this way, but differs from IDEA by using an explicit (declarative) model of control behavior. RMPL can represent control flow constructs such as loops and conditionals, which are compiled into a declarative model used for planning and then interpreted during execution.

Future Work

Propel 2 is currently in the working prototype stage. We have identified many open research issues including:

Backtracking issues such as using model-based diagnosis to provide dependency directed backtracking. We also must address issues such as deciding which concurrent processes must be planned together, and simulation with metric time (backtracking and warping forward).

Executive Strategies for managing transitions between planning and execution. This includes proactive planning, concurrent, and interleaved planning, anytime planning, and planning after a failure occurs. This involves definition of the planner termination test which decides when the planner has "gotten around" the current failure so that execution may continue.

Software Sensors and Actuators. We currently insert macros to instrument the code by hand. Future work may use a separate preprocessing phase to automatically instrument the code, and also OS level instrumentation of computational state. We'd like to use OS-level actuators that may provide lightweight alternatives to fork(). We also need a better way to capture the control stack information used by SCRs.

Runtime Simulators are needed so the application code can run in _PLANNING_ mode (see Line 58). The simulators are needed only for physical actions and may provide different levels of abstraction and/or fidelity. Users can plug in application-specific simulators or use Propel's built-in database to keep track of simulated or executed state properties. Planning and execution have their own copies of the database.

Performance - The search nodes are currently implemented as computational continuations (created by the UNIX fork() command). Future work will involve using lower-overhead alternatives to fork(). Also could use branch and bound to limit the number of processes that remain open for backtracking.

Evaluation Plan

We will perform experiments to test our hypothesis that unified planning and execution with a procedural representation can significantly increase failure recovery scope and decrease cost. We will inject software failures into complex software and measure the coverage of existing recovery systems compared to our approach. We will measure the costs for human vs. autonomous recovery, and performance costs of the new methods.

Conclusion

Propel is an architecture and a language. The architecture provides integrated planning and execution modules that monitor and manipulate application-level processes written in the Propel language. The language is a library of methods for embedding search and temporal constraint information into C++, creating a "superset" of C++. This library provides an interface from the application-level processes to supervisory meta-processes (the planning and execution modules) which monitor the application level processes in order to detect and correct software failures.

Propel provides unified planning and execution modules that share a procedural action representation. Motivation for using a procedural representation includes the following goals:

- Include all software within the planner's model in order to increase the scope of failure recovery to include infrastructure software failures.
- Represent complex procedures including loops, conditionals, local variables, and multiprocessing.
- Reduce the need to develop and maintain different models for the planner and execution system.
- Reduce risk of loss of information in translation between execution and planning (and vice versa).

Propel is designed to increase the scope of the planner's model to include software failure detection and recovery, and to close the gap between the declarative action model used by a planner and the procedural languages used to develop real-world software. The representation is intended to be expressive enough to embed search and temporal constraints into system and infrastructure software in order to recover from software failures and increase system autonomy.

References

Bonasso, R, Firby R., Gat, E., Kortenkamp, D., Miller, D., and Slack, M. Experiences with an Architecture for Intelligent, Reactive Agents, in *Journal of Experimental and Theoretical Artificial Intelligence* , January, 1997.

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

Drummond. M. Situated Control Rules. 1989. *Proceedings of Knowledge Representation 1999* (KR'89).

Drummond, M., Bresina, J., Swanson, K., Levinson, R. 1993. Reaction-First Search: Incremental Planning with Guaranteed Performance Improvement. *Proceedings. of IJCAI-93*. Chambrey, France.

Kim P., Williams B., Abramson M., 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. IJCAI '01. AAAI, Menlo Park, CA.

Levinson, R. 1995. A General Programming Language for Unified Planning and Control. *Artificial Intelligence*, Vol. 76. See http://www.brainaid.com/papers/propel_aij95.pdf.

Levinson R. 2005. Unified Planning and Execution for Autonomous Software Repair (10-pg version of this paper) http://www.brainaid.com/papers/propel_ijcai05.pdf.

Muscettola, N., Dorais, G., Fry, C., Levinson, R., Plaunt, C. 2000. A Unified Approach to Model-Based Planning and Execution. *Proc. of IAS '2000*. Venice, Italy.

Efficiently Solving Hybrid Logic/Optimization Problems

Through Generalized Conflict Learning

Hui Li and Brian Williams
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{huili, williams}@mit.edu

Abstract

An increasing range of problems in Artificial Intelligence and Computer Science, such as autonomous vehicle control and planning with resources, are formulated through a combination of logical, algebraic and cost constraints. Their solution requires a hybrid mixture of logical decision techniques and mathematical optimization. Using Disjunctive Programming (DP) to formulate these problems, we present a novel algorithm, called DP Conflict-Directed Branch and Bound (B&B), that efficiently solves DP problems through a powerful three-fold method. First, during the search process, *generalized conflict learning* learns qualitative descriptions (*conflicts)* for regions of the state space that are infeasible or sub-optimal. Second, *forward conflict-directed search* uses these qualitative descriptions to heuristically guide the forward step of the search, by moving away from regions of state space corresponding to known *conflicts*. Finally, *induced unit clause relaxation* automatically forms a strong relaxed problem from a subset of the unit clauses that are implied by the original problem. Our experiments on model-based temporal plan execution for cooperative vehicles demonstrate an order of magnitude speed-up over Mixed Integer Programming B&B.

1 Introduction

An increasing range of problems in Artificial Intelligence and Computer Science involve finding optimal solutions to problems that involve a rich combination of logical and algebraic constraints, and require a hybrid coupling of logical decision techniques with mathematical optimization to solve. Examples include planning with resources, autonomous vehicle control and verification of timed and hybrid systems. Focusing on the area of autonomous vehicle control, deep space explorers must choose between tasks and temporal orderings, while optimizing flight trajectories for fuel usage. On Earth, search and rescue units must construct and compare different vehicle trajectories around dangerous areas, such as a fire, on the approach to a trapped individual.

Each of these tasks involves designing an optimal state trajectory, based on a continuous dynamic model. At some point, they must satisfy additional logical constraints, such as mission tasks, task orderings and obstacle avoidance.

These Hybrid Logic/Optimization Problems (HLOPs) can be formulated in three ways: first, by introducing integer variables and corresponding constraints to Linear Programming (LP), known as Mixed Integer Programming (MIP) as in [Schouwenaars et al, 2001, Vossen et al, 1999, Kautz and Walser, 1999]; second, by augmenting LP with propositional variables so that the propositional variables can be used to "trigger" linear constraints, known as Mixed Logic Linear Programming (MLLP)[1] in [Hooker and Osorio, 1999] and LCNF[2] in [Wolfman and Weld, 1999]; third, by extending LP with disjunctions, without adding any variables or constraints, called Disjunctive Programming (DP) as in [Balas, 1979]. This paper builds upon the last option, DP, which combines the expressive power of propositional logic with that of LP, without the overhead of additional variables or constraints.

In this paper we introduce a novel algorithm for efficiently solving Disjunctive Programs called Conflict-Directed Branch & Bound (DP-CD-BB). It extends the Branch & Bound (B&B) algorithm by using logical inference to do relaxation, by abstracting the qualitative descriptions of the source of discovered infeasibility and sub-optimality as *conflicts* to guide the forward search, so as to prune the state space. Our experiments, comparing DP-CD-BB against Mixed Integer Programming B&B (MIP-BB), demonstrate a significant performance gain on model-based temporal plan execution for cooperative vehicles.

DP-CD-BB builds upon the Conflict-Directed Clausal LP Branch & Bound method in [Krishnan, 2004], which uses the same formulation as DP and learns infeasible states as conflicts to guide the search. The concept of conflict learning from sub-optimality draws from Activity Analysis in [Williams and Cagan, 1994], which reasons using qualitative abstractions of sub-optimal subspaces in order to guide the numerical methods away from subspaces with the same abstractions.

2 Problem Formulation

We use disjunctive programs to effectively capture both the continuous dynamics and control decisions present in

[1]MLLP is a generalization from MIP, but its main feature is the introduction of propositional variables.
[2]Optimization is not involved.

hybrid logic/optimization problems. Figure 1 depicts a simple example of an HLOP, introduced in [Schouwenaars et al, 2001]. Eq. (1) describes its DP formulation. In particular, this is an instance of a spatial reasoning problem, in which a vehicle has to go from point A to C, without hitting the obstacle B, while minimizing the fuel use.

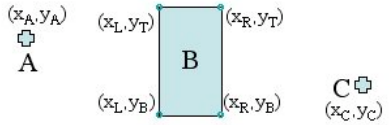


Figure 1. A simple example of an HLOP

$$\begin{aligned} &\text{Minimize} && f(\mathbf{x}) \\ &\text{Subject to} && \mathbf{g(x) \le 0} \qquad (1) \\ &\text{and} && x_i \le x_L \ \mathbf{V}\ x_i \ge x_R \ \mathbf{V}\ y_i \le y_B \ \mathbf{V}\ y_i \ge y_T\ , \\ & && \forall\ i = 1, \ldots, n \end{aligned}$$

In Eq. (1), **V** denotes logical *or*, **x** is a vector of decision variables that includes, at each time step i (=1,…,n), the position, velocity and acceleration of the vehicle; f(**x**) is a linear cost function in terms of fuel use; **g(x) ≤ 0** is a conjunction of linear constraints on vehicle dynamics, and the last constraint keeps the vehicle outside obstacle B at each time step i. In general, DP takes the form shown in Eq. (2):

$$\begin{aligned} &\text{Minimize} && f(\mathbf{x}) \\ &\text{Subject to} && \mathbf{\Lambda}_{i=1,\ldots,n} \left(\mathbf{V}_{j=1,\ldots,mi}\ C_j(\mathbf{x}) \le 0 \right) \qquad (2) \end{aligned}$$

where **x** is a vector of decision variables, f(**x**) is a linear cost function, and the constraints are a conjunction of n *clauses*, each of which (*clause* i) is a disjunction of m_i linear inequalities, $C_j(\mathbf{x}) \le 0$. DP reduces to a standard LP in the special case when $m_i=1$, $\forall i=1,\ldots,n$. In comparison with MIP, DP adds no overhead variables or constraints to represent logical decisions.

3 The DP-CD-BB Algorithm

The DP-CD-BB algorithm has four key features: First, Generalized Conflict Learning, which learns qualitative abstractions (*conflicts)* comprised of constraint sets that produce either infeasibility or sub-optimality; Second, Forward Conflict-Directed Search, which heuristically guides the *forward step* of the search away from regions of state space corresponding to known *conflicts*; Third, Induced Unit Clause Relaxation, which forms a relaxed problem from a subset of the unit clauses that are induced from the original problem; Fourth, Search Order: Best-first Search (BFS) versus Depth-first Search (DFS). In the following subsections, we develop these key features in detail, including examples and pseudo code.

DP-CD-BB builds upon B&B, which is frequently used by MIP, to solve problems involving both discrete and continuous variables. Instead of exploring the entire feasible set of a constrained problem, B&B uses bounds on the optimal cost, in order to avoid exploring subsets of the feasible set that it can prove are sub-optimal, that is, subsets whose optimal solution is not better than the *incumbent*, which is the best solution found so far. Pseudo code for B&B is shown in Figure 2.

GenericBB (original problem F)

```
1   incumbent U = +∞;
2   select a sub-problem Fi;
3   if (Fi is infeasible)
4      delete it;
5   else
6      compute the lower bound lb(Fi);
7   if (lb(Fi)≥ U)
8      delete Fi;
9   else if (the solution to Fi satisfies all the constraints of
    F)
10     U ← lb(Fi);
11  else
12     break Fi into sub-problems;
```

Figure 2. Pseudo code for generic Branch & Bound

The search tree of B&B for MIP branches by assigning values to the integer variables. In our case, B&B for DP branches by splitting clauses. At each node in the search tree, a relaxed LP is solved. *p'* is a relaxed LP of an optimization problem *p,* if the feasible region of *p'* contains the feasible region of *p*, and they have the same objective function. Therefore if *p'* is infeasible, then *p* is infeasible; if *p'* is solved with an optimal value *v*, the optimal value of *p* is guaranteed to be worse than *v*. B&B uses relaxed problems to obtain lower bounds of the original problem (assuming minimization).

3.1 Generalized Conflict Learning

Underlying the power of B&B is its ability to prune subsets of the search tree that correspond to relaxed sub-problems that B&B identifies as inconsistent or sub-optimal, as seen in line 4 and 8 in Figure 2.

In the related field of discrete constraint satisfaction, conflict-directed methods, such as dependency-directed backtracking [Stallman and Sussman, 1977], backjumping [Gaschnig, 1978], conflict-directed backjumping [Prosser, 1993] and dynamic backtracking [Ginsberg, 1993], dramatically improve the performance of backtrack (BT) search, by learning the source of each inconsistency discovered, and by using this generalization, called a *conflict*, to prune additional sub-trees that the conflict identifies as inconsistent.

To apply conflict learning to B&B, we note that B&B prunes subtrees corresponding to relaxed sub-problems that are sub-optimal and infeasible. Hence two opportunities exist for learning and generalized pruning. We exploit these opportunities by introducing the concept of *generalized conflict learning*, which extracts a qualitative

description from each pruned (fathomed) sub-problem that is infeasible or sub-optimal. This avoids exploring sub-problems with the same description in the future. Moreover, it is valuable to have the qualitative description as compact as possible, because the smaller the *conflict* is the larger the subspace to be pruned.

Each conflict can be of two types: (1) an irreducible set of constraints that is learned from infeasibility, or (2) an irreducible set of constraints that is learned from sub-optimality. A set of constraints is *irreducible* if removing any one of the constraints from the set resolves the infeasibility or sub-optimality. Note that there can be more than one irreducible sets (possibly with different cardinalities) involved in one infeasibility or sub-optimality, and a type-1 or type-2 conflict is not guaranteed to have the *minimal* cardinality. Hence the name *irreducible* instead of *minimal*. An *infeasibility conflict* (type 1) is an irreducible subset of the inconsistent constraints of an infeasible sub-problem. An example is the constraint set {a,c,d} in Figure 3(b). The sub-problem in Figure 3(a) is infeasible, but its constraint set is not an *infeasibility conflict*, as a proper subset of it, as in Figure 3(b), remains inconsistent. An *active* constraint of a feasible problem *S*, is a constraint that takes equality at *S*'s optimal solution x*. A *sub-optimality conflict* (type 2) is an irreducible subset of the active constraints of a feasible sub-problem whose optimal solution is not better than the *incumbent*. An example is the constraint set {c} in Figure 4(b). All the constraints are active in Figure 4(a), but the set {a,b,c,d} is not a *sub-optimality conflict*, as it can be reduced to Figure 4(b) without affecting the optimal solution x*.

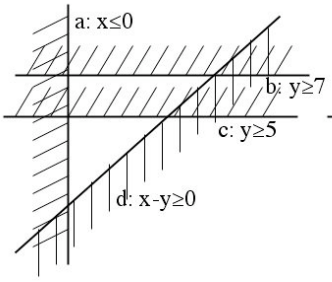
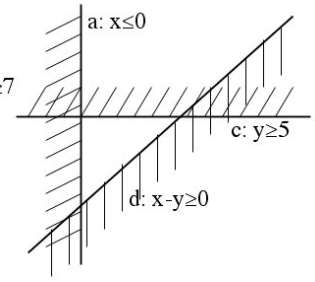


Figure 3(a). An infeasible sub-problem: constraint set {a,b,c,d} is not consistent.



(b) After removing constraint b, it is still infeasible.

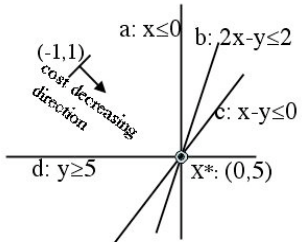
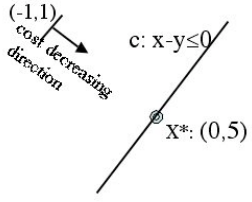


Figure 4(a).The optimal solution is X*. Constraints a, b, c and d are all active.



(b). After removing constraints a, b and d, X* stays the same.

To perform *generalized conflict learning* efficiently, the dual method of LP is used to extract a sub-problem's irreducible set. For infeasibility, this function is provided by the commercial software CPLEX: *getIIS()*, and its principle is explained in [Wolfman and Weld, 1999]. For sub-optimality, we introduce a novel approach based on the LP dual method. According to Complementary Slackness [Bertsimas and Tsitsiklis, 1997] from linear optimization theory or equivalently Kuhn-Tucker conditions for the linear case [Williams and Cagan, 1994], the non-zero terms of the optimal dual vector correspond to the irreducible set of active constraints at the optimal solution of the LP. Thus we use the dual optimal solution that is provided by the CPLEX function, *getDuals()*, to identify the *sub-optimality conflict.* The pseudo code is shown in Figure 5. After they are extracted, the conflicts are stored in a conflict database, *confDB,* with a timestamp that marks when they are discovered.

```
ExtractSubConf(LP problem p)
1.  solve p using CPLEX;
2.  if (p solved with an optimal solution) {
3.     dual ← getDuals();
4.     for (int  i=0; i<dual.length; i++)
5.        if (dual[i] !=0)
6.           subConf.add(constraint[i]); //constraint[i] is the
    corresponding constraint in p.
7.     return subConf;
8.  }else
9.     return null;
```

Figure 5. The function to extract sub-optimality conflicts

3.2 Forward Conflict-directed Search

The *forward conflict-directed search* heuristically guides the forward step of the search away from regions of the feasible space that are ruled out by known conflicts. Traditionally, conflicts are used in the backward step, such as dependency-directed backtracking [Stallman and Sussman, 1977], backjumping [Gaschnig, 1978], conflict-directed backjumping [Prosser, 1993], dynamic backtracking [Ginsberg, 1993] and LPSAT [Wolfman and Weld, 1999]. These backtrack search methods use conflicts to select backtrack points. In contrast, we use conflicts in forward search, to move away from known "bad" states. We generalize this idea to guiding B&B away from regions of state space that the known conflicts indicate are infeasible or sub-optimal. Our experiment results show that *forward conflict-directed search* significantly outperforms backtrack search on a range of cooperative vehicle plan execution problems.

The implementation , as seen in the pseudo code in Figure 6, includes three steps. 1. Conflict retrieval from *confDB*: only conflicts that are discovered after the creation time of the node to be expanded,

nodeToExp.timestamp, are retrieved, because conflicts discovered before are resolved by the creation of this node or its parents. Note that a node in the search tree represents a DP problem, which is a partially assigned problem from the original DP problem. 2. Negation: the types of the linear constraints in each conflict are reversed (e.g. ≤ becomes ≥) and the relation between the constraints in each conflict becomes logical *or*, so that a conflict is turned into a clause, called a *conflict clause*. Recall that a conflict represents the region where no feasible solution or only sub-optimal solutions exist. Therefore a *conflict clause* denotes the regions where an optimal solution can lie. 3. Clause addition: conflict clauses, *confClauses*, are added to the clause set of the node to be expanded, *nodeToExp.clauseSet*. In this way, *nodeToExp* is updated.

```
ForwardCDSearch(confDB, nodeToExp)
1.  if (confDB(nodeToExp.timestamp) != null) {
2.      currConfs ← confDB(nodeToExp.timestamp);
3.      for (int  i=0; i<currConfs.length; i++)
4.          for (int j=0; j<currConfs[i].length; j++)
5.              confClauses[i].add(¬ currConfs[i][j]);
6.      nodeToExp.clauseSet.add(conClauses);
7.  }
8.  GenericBB(nodeToExp);
```

Figure 6. Pseudo code for forward conflict-directed search

3.3 Induced Unit Clause Relaxation

Relaxation is an essential tool for quickly characterizing a problem when the original problem is hard to solve directly; it provides bounds on feasibility and the optimal value of a problem, which are commonly used to prune the search space. Previous research [Hooker, 2002] typically solves Disjunctive Programs by reformulating them as Mixed Integer Programs, in which binary integer variables are used to encode the disjunctive constraints. A relaxed problem for a MIP consists of the continuous relaxation of the integer constraints.

An alternative way of creating a relaxed LP is to operate on the DP encoding directly, by removing all non-unit clauses from the DP (a unit clause is one that contains a single constraint). Prior work argues for the reformulation of DP as MIP relaxation, with the rationale that it allows the solver to use continuous relaxation on the (binary) integer variables, in contrast to ignoring the non-unit clauses. However, this benefit is at the cost of adding integer variables and constraints, which can significantly increase the dimensionality of the search problem. This cost is not incurred by the DP relaxation.

Our approach starts with the direct DP relaxation, hence drawing from its strength in terms of a smaller state space. We overcome the weakness of standard DP relaxation (loss of non-unit clauses) by adding to the relaxation additional unit clauses that are logically entailed by the original DP. In the experiment section we compare our *induced unit clause relaxation* with the MIP relaxation and show a profound improvement on a range of cooperative vehicle plan execution problems.

The strongest relaxed problem is constructed when all entailed unit clauses are added to the relaxed problem; however, finding all of them is NP hard. A relaxation is valuable only to the extent that it saves computation time. Hence we choose the middle ground of finding all unit clauses that can be quickly induced. From propositional theories, unit clauses can be induced quickly through unit propagation; we generalize this approach to DP.

To implement *induced unit clause relaxation*, as seen in pseudo code in Figure 7 and the example in Figure 8, three steps are included. 1. Each unique constraint in the clause set of the DP problem *p* is assigned a unique propositional symbol. 2. (Incremental) Unit Propagation, as presented in ITMS [Nayak and Williams, 1997], is used to induce unit clauses. 3. The relaxed LP problem is formed with all the unit clauses induced from *p* and the objective function of *p*.

```
UnitClauseRelax(DP problem p)
1.  symbClauseSet ← p.clauseSet; //assigning a unique
    propositional symbol to each unique linear inequality.
2.  unitClauseSet ← UnitPropagation(symbClauseSet);
3.  relaxLP.constraintSet ← unitClauseSet.convert();
    //converting symbols to linear inequalities to form the
    relaxed LP
4.  relaxLP.objective ← p.objective;
5.  return relaxLP;
```

Figure 7. Pseudo code for induced unit clause relaxation

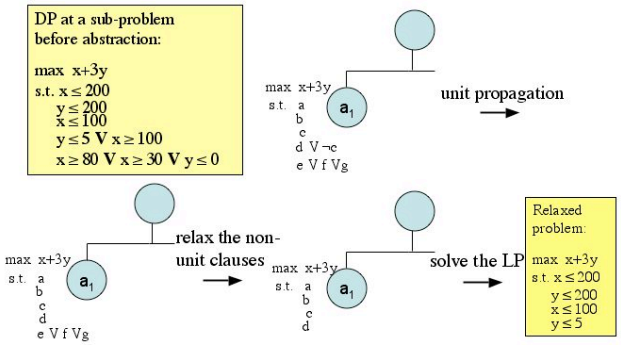


Figure 8. An example of induced unit clause relaxation

3.4. Search Order: Best-first versus Depth-first

For B&B it is known empirically that, in the average case, Best-first Search (BFS) performs better than Depth-first Search (DFS) in time efficiency. This is because BFS expands the search tree in the order of always exploring the most promising node, thus allows larger portions of the search tree to be pruned. However, BFS can take dramatically more memory space than DFS. Nevertheless, with conflict learning and forward conflict-directed search

the queue of the BFS search tree is significantly reduced. Our experimental results show that BFS can take memory space similar to DFS.

An additional issue for DP-CL-BB is that the concept of sub-optimality is rooted in maintaining an incumbent. Hence it can be applied to DFS but not to BFS (which does not have an incumbent). To evaluate these tradeoffs, our experiments in the next section compare the use of BFS and conflict learning from infeasibility only, with DFS and conflict learning from both infeasibility and from sub-optimality.

4 Experimental Performance Analysis

This section provides experimental results of the DP-CD-BB solver, compared with the benchmark MIP-BB, on a range of problems. We also compare the effect of several parameters, in particular, BFS versus DFS, infeasibility conflict learning versus sub-optimality conflict learning and forward search versus backtrack search. While each algorithmic variant terminates with the same optimal solution, a major result is that DP-CD-BB achieves an order of magnitude speed-up over MIP-BB. In addition, the difference in performance increases as the problem enlarges.

As the bulk of the computational effort expended by these algorithms is devoted to solving relaxed LP problems, the total number and average size of these LPs are representative of the total computational effort involved in solving the HLOPs. Note that extracting infeasibility conflicts and sub-optimality conflicts can be achieved as by-products of solving the LPs, and therefore does not incur any additional LP to be solved. We use the total number of relaxed LPs solved and the average LP size as our LP solver and hardware independent measures of computation time.

As explained in the first section and verified by our experiments, the MIP encoding enlarges the HLOP by adding overhead integer variables and constraints. Therefore, the average size of its LPs is larger than that of the LPs solved for the DP encoding. Experiments also show that the average sizes of relaxed LPs, solved by all the methods that use the DP encoding, are similar to each other. The data table is not listed due to space limit of the paper. To measure memory space use, maximum queue size is used.

We programmed MIP-BB, DP-CD-BB and its variations in Java. All used the commercial software CPLEX as the LP solver. Test problems were generated using a model-based temporal planner, performing multi-vehicle search and rescue missions. This planner takes as input a temporally flexible state plan, which specifies the goals of a mission, and a continuous model of vehicle dynamics, and encodes them in DP. The DP-CD-BB solver generates an optimal vehicle control sequence that achieves the constraints in the temporal plan. For each Clause/Variable set, 15 problems were generated and the average was recorded in the tables. These planning problems are tightly constrained and hence often "hard" problems, as studied in [Mitchell et al, 1992].

In Table 1, the number of relaxed LPs solved for each approach is recorded. The second row shows that MIP-BB solves more LPs than any other approach. The difference increases dramatically as the problem grows larger. The next three rows are dedicated to DP with BFS. The addition of infeasibility conflict learning (Inf) significantly outperforms without conflict learning (w.o. CL). The method using conflict-directed backtrack search (BT), which uses infeasibility conflicts to check consistency of a relaxed LP before solving it, performs dramatically worse than the method using forward conflict-directed search (Inf). The last five rows represent the variations of DP with DFS. Within these five rows, the method that solves the least relaxed LPs is the method with both infeasibility and sub-optimality conflict learning (Sub+Inf). The worst case is w.o. CL.

Consider BFS versus DFS, using only infeasibility conflict learning, BFS performs better than DFS, but the performance of DFS with Sub+Inf is close to BFS with Inf. For very large problems, DFS with Sub+Inf performs better. Under the same situation, either w.o. CL or with Inf or with BT, BFS solves less relaxed LPs than DFS, as explained before in section 3.4. As the only difference between DP with DFS without conflict learning and MIP-BB is in the formulation and relaxation methods, the significant improvement of the former over the latter verifies the statement in section 3.3.

For all tests, our DP-CD-BB algorithm, using either DP with BFS and infeasibility conflict learning, or DP with DFS and infeasibility plus sub-optimality conflict learning, performs the best and has a profound improvement over MIP-BB on large problems.

Clause / Variable		80 / 36	700/ 144	1492/ 300	2336/ 480
MIP-BB		31.5	2009	4890	8133
DP BFS	w.o. CL	24.3	735.6	1569	2651
	Inf	19.2	67.3	96.3	130.2
	BT	23.1	396.7	887.8	1406
DP DFS	w.o. CL	28.0	2014	3023	4662
	Inf	22.5	106.0	225.4	370.5
	BT	25.9	596.9	1260	1994
	Sub +Inf	22.1	76.4	84.4	102.9
	Sub	25.8	127.6	363.7	715.0

Table 1. Comparison on the number of relaxed LPs solved

In Table 2, all approaches have similar maximum queue sizes, except BFS w.o. CL and BFS with BT. As discussed in section 3.4, BFS generally takes more memory space than DFS, but when forward conflict-directed search is used, the search space is reduced and the corresponding queue is shortened. Note that although the methods using BT are conflict-directed, the queue size of the one with BFS is not largely reduced.

Clause / Variable		80 / 36	700/ 144	1492/ 300	2336/ 480
MIP-BB		8.4	30.8	46.2	58.7
DP BFS	w.o. CL	19.1	161.1	296.8	419.0
	Inf	6.4	18.3	38.4	52.5
	BT	15.6	101.7	205.1	327.8
DP DFS	w.o. CL	6.1	18.7	25.1	30.3
	Inf	6.5	21.4	45.0	57.3
	BT	6.1	18.4	23.5	28.1
	Sub +Inf	6.5	21.4	33.0	40.9
	Sub	6.5	21.6	38.7	47.0

Table 2. Comparison on the maximum queue size

5. Conclusion

Hybrid Logic/Optimization Problems can be encoded effectively using Disjunctive Programming (DP). This paper presented a novel algorithm, DP Conflict-Directed Branch and Bound, that efficiently solves DP problems through a powerful three-fold method, featuring *generalized conflict learning, forward conflict-directed search* and *induced unit clause relaxation*. The key feature of the approach is that infeasible or sub-optimal subsets of state space are reasoned using qualitative descriptions (*conflicts*), in order to heuristically guide the forward step of the search, by moving away from regions of state space corresponding to known *conflicts*. Our experiments on model-based temporal plan execution for cooperative vehicles demonstrated an order of magnitude speed-up over Mixed Integer Programming Branch and Bound.

References

[Balas, 1979] E. Balas. Disjunctive programming, *Annals of Discrete Mathematics* **5** 3-51.

[Bertsimas and Tsitsiklis, 1997] D.Bertsimas and J.N.Tsitsiklis. Introduction to Linear Optimization.

[Bitner and Reingold, 1975] J. Bitner and E. Reingold. Backtrack Programming Techniques, *Communications of the Association for Computing Machinery 18(11).*

[Gaschnig, 1978] J. Gaschnig Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisficing Assignment Problems. *The 2nd Canadian Conference on AI 268-277.*

[Ginsberg, 1993] M. Ginsberg, Dynamic backtracking, *Journal of Artificial Intelligence Research* ***1*** *25–46.*

[Hooker and Osorio, 1999] J.N.Hooker and M.A.Osorio. Mixed Logical/Linear Programming. *Discrete Applied Mathematics* ***96-97*** *395-442.*

[Hooker, 2002] J.N.Hooker, Logic, optimization and constraint programming, *INFORMS J. on Computing 14 295-321.*

[Krishnan, 2004] R. Krishnan. Solving Hybrid Decision-Control Problems Through Conflict-Directed Branch & Bound. *M.Eng. Thesis. MIT.*

[Mitchell et al, 1992] D. Mitchell, B. Selman, H. Levesque. Hard and easy distributions of SAT problems. *AAAI.*

[Nayak and Williams, 1997] P. Nayak, B. Williams. Fast Context Switching in Real-time Propositional Reasoning, *AAAI.*

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem, *Computational Intelligence 3, 268–299.*

[Ragno, 2002] R. Ragno. Solving Optimal Satisfiability Problems Through Clause-Directed A*. *M.Eng. Thesis. MIT.*

[Schouwenaars et al, 2001] T. Schouwenaars, B. De Moor, E. Feron, J. How. Mixed integer programming for multi-vehicle path planning. *European Control Conference.*

[Stallman and Sussman, 1977] R. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* ***9*** *135–196.*

[Vossen et al, 1999] T. Vossen, M. Ball, A. Lotem, D. Nau. On the use of integer programming models in AI planning. *IJCAI.*

[Williams and Cagan, 1994] B. Williams and J. Cagan. Activity Analysis: The Qualitative Analysis of Stationary Points for Optimal Reasoning. *AAAI.*

[Wolfman and Weld, 1999] S. Wolfman and D. Weld. The LPSAT engine & its application to resource planning. *IJCAI.*

A Fast Incremental Dynamic Controllability Algorithm

John L. Stedl
Massachusetts Institute of Technology
32 Vassar Street
Cambridge, MA 02139
stedl@mit.edu

Brian C. Williams
Massachusetts Institute of Technology
32 Vassar Street
Cambridge, MA 02139
williams@mit.edu

Abstract

In most real-world planning and scheduling problems, the plan will contain activities of uncertain duration whose precise duration is observed rather than controlled by the agent. In many cases, in order to satisfy the temporal constraints imposed of the plan, the agent must dynamically adapt the schedule of the plan in response to these uncertain observations. Previous work has introduced a polynomial-time *dynamic controllability (DC)* algorithm, which reformulates the temporal constraints of the plan into a form amenable for dynamic execution.

In this paper we introduce a novel, fast incremental DC algorithm that (1) is significantly faster than previous approaches in reformulating partially controllable plans for dynamic execution and (2) efficiently maintains the dispatchability of the plan when the subset of the constraints change. This new Fast DC algorithm has been experimentally shown to run in $O(N^3)$ time when reformulating unprocessed plans and in O(N) time when incrementally maintaining the dispatchability of the plan.

Introduction

In most real-world planning and scheduling problems, the timing of some of the events will be controlled by the agent; while others will be controlled by nature. For example, a Mars rover is capable of controlling when it starts driving to a rock; however, its precise arrival time is determined by environmental factors.

For plans that contain activities of uncertain duration, it is insufficient to merely guarantee that there exists a feasible schedule. Instead, the agent must ensure there is a strategy to consistently schedule the controllable events for all possible outcomes of the uncertain durations. The problem of determining if a viable execution strategy exists was first formally addressed by [Vidal 1996, Vidal and Fargier 1999]. This work has identified three primary levels of *controllability*: Strong, Dynamic and Weak. Controllability refers to the ability to "control" the consistency of the schedule, despite the uncertainty in the plan. In this paper we are concerned with *dynamic controllability*, in which the uncertain durations are observed at execution time. Informally, a plan is dynamically controllable if there is a successful execution strategy that assigns execution times to the controllable events, which only depends on past outcomes and satisfies the timing constraints in the plan for all possible execution times of uncontrollable events.

Beyond merely checking if the plan is dynamically controllable, we are interested in process of reformulating the plan into a dispatchable form, such that it can be efficiently scheduled at execution time. In addition, we are interested in addressing the problem of quickly reformulating the dispatchable plan in response to either changing timing requirements. This ability becomes particularly important when dealing with plans for highly agile systems, such as unmanned aerial vehicles, where there may not be enough time to completely re-plan if some of the constraints change.

[Morris 2001] introduced a polynomial time dynamic controllability (DC) algorithm to reformulate a partially controllable plan into a dispatchable plan. In this paper we improve upon this DC algorithm in two key ways. First we introduce a fast incremental dynamic controllability algorithm (Fast-DC) that enables the agents to quickly maintain the dispatchability of the plan when only some of the constraints change. Second, we show how to efficiently apply this incremental DC algorithm in the startup case in order to reformulate unprocessed plans into a dispatchable form. In the startup case our algorithm is faster than previous approaches.

[Morris 2001] showed that the issue of converting a partially controllable plan into a dispatchable form is reduced to repeatedly applying a set of constraint propagations. In this paper we show how to efficiently apply these constraint propagations.

In order to develop our incremental DC algorithm we introduce and exploit a property called *pseudo-dispatchability*, which enables an efficient, recursive constraint propagation scheme, called *dispatchability-back-propagation (DBP)*. The sub-term "back-propagation" refers to that property that the constraints only need to be propagated toward the start of the plan. When reformulating unprocessed plans, this purely recursive approach removes the need to perform repeated calls to an $O(N^3)$ All-Pairs Shortest-Path (APSP) algorithm, as required by the DC algorithm introduced by [Morris 2001].

In the startup case we introduce two other innovations. First our new dynamic controllability algorithm removes redundant constraints before performing constraint propagation, which significantly reduces the number of propagations required. Second, we intelligently initiate the DBPs such that the algorithm continuously reduces the size of the problem.

First we review background on Simple Temporal Networks with Uncertainty (STNU) and present a simple

example. Then we describe how to perform DBP on STNs. Next we extend this DBP framework to STNUs. Next we introduce the Incremental DC algorithm to handle the case when only one constraint changes. Next we show how to efficiently apply this incremental DC algorithm for both unprocessed plans and when multiple constraints change. Finally, we present some experimental results of our Incremental DC algorithm.

Background

A Simple Temporal Network with Uncertainty [Vidal and Fargier 1999] is an extension of a STN [Decher 1991] that distinguishes between controllable and uncontrollable events. A STNU is a directed graph, consisting of a set of nodes, representing *timepoints*, and a set of edges, called links, constraining the duration between the timepoints. The links fall into two categories: *contingent links* and *requirement links*. A contingent link models an uncontrollable process whose uncertain duration, ω, may last any duration between the specified lower and upper bounds. A requirement link simply specifies a constraint on the duration between two timepoints. All contingent links terminate on a *contingent timepoint* whose timing is controlled by nature. All other timepoints are called requirement timepoints and are controlled by the agent.

Definition (STNU [Vidal 1999])*: A STNU is a 5-tuple < N, E, l, u, C >, where N is a set of timepoints, E is a set of edges and l : E → Â È {-¥} and u : E → Â È {+¥} are functions mapping the edges to lower and upper bound temporal constraints. The STNU also contains C, which is a subset of the edges that specify the contingent links, the others being requirement links. We assume 0 < l(e) < u(e) for each contingent link.*

To support efficient inference, a STNU is mapped to an equivalent distance graph [Dechter 1991], called a Distance Graph with Uncertainty (DGU), where each link of the STNU, containing both lower and upper bounds, is converted into a pair DGU edges, containing only an upper bound constraint. In the DGU, the distinction between a contingent and a requirement edge is maintained. For example consider the triangular STNU and associated DGU shown in Figure 1.

A STNU is consistent only if its associated DGU contains no negative cycles [Dechter 1991]. This can be efficiently checked by applying the Bellman-Ford SSSP algorithm [CLR 1990] on the DGU. However consistency does not imply dynamic controllability.

In order for the STNU to be dynamically controllable, each uncontrollable duration, ω_i, must be free to finish any time between $[l_i,u_i]$, as specified by the contingent link, C_i. The set of all implicit constraints contained in the STNU can be made explicit by computing the All-Pairs Shortest-Path (APSP) graph of the DGU via the Floyd-Warshall algorithm [CLR 1990].

If temporal constraints (requirement and contingent) of the plan imply strictly tighter bounds on an uncontrollable duration, then that uncontrollable duration is considered

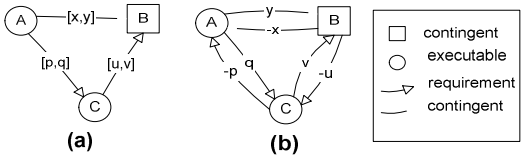


Figure 1 (a) Triangular STNU and (b) DGU

squeezed [Morris 2001] and the plan is not dynamically controllable. In this case there exists a *situation* [Vidal 1999] where the outcome of the uncontrollable duration may result in an inconsistency. A STNU is *pseudo-controllable* [Morris 2001] if it is both temporally consistent and none of its uncontrollable durations are squeezed.

In this paper we are interested preparing the STNU for dynamic execution in which a dispatcher [Morris 2001] uses the associated DGU to schedule timepoints at execution time. In this case, even if a STNU is pseudo-controllable, the uncontrollable durations may be squeezed at execution time [Morris 2001].

The dynamic controllability (DC) reformulation algorithm introduced by [Morris 2001] adds additional constraints to the plan, in order to enable the dispatcher to consistently schedule the plan at execution time without squeezing the uncontrollable durations. In our Fast-DC algorithm we apply these tightenings efficiently.

Incremental STN Dispatchability Maintenance

The speed of our Incremental Fast-DC algorithm depends on a technique called *dispatchability-back-propagation (DBP)*. In this section we introduce the DBP rules for STNs. In the next section we extend these rules for STNUs.

In order to address real time scheduling issues, [Muscettola 1998b] showed that any consistent STN can be converted into an equivalent *dispatchable* distance graph, which can be dynamically scheduled using a locally propagating dispatching algorithm [Muscettola 1998a].

The dispatching algorithm schedules and executes the timepoints at the same time. The dispatcher works by maintaining a feasible execution widow, $W_x \in [lb_x,ub_x]$, for each timepoint X. When the dispatcher executes a timepoint A, upper-bound updates are propagated via all outgoing, non-negative edges AB and lower-bound updates are propagated along via all incoming negative edges, CA. The dispatching algorithm is free to schedule timepoint X anytime within that timepoints execution window, as long as the timepoint is enabled. A timepoint is enabled if all timepoints that must precede have been executed.

For a dispatchable graph, the dispatcher is able to guarantee that it can make a consistent assignment to all future timepoints, as long as each scheduling decision is consistent with the past. Therefore, in order to maintain the dispatchability of the graph when a constraint is modified, we only need to make sure that the change is consistent with the past; the dispatcher will ensure that this constraint change is consistent with the future at execution time.

We call the process of ensuring that a change is consistent with the past, *Dispatchability Back-Propagation (DBP).*

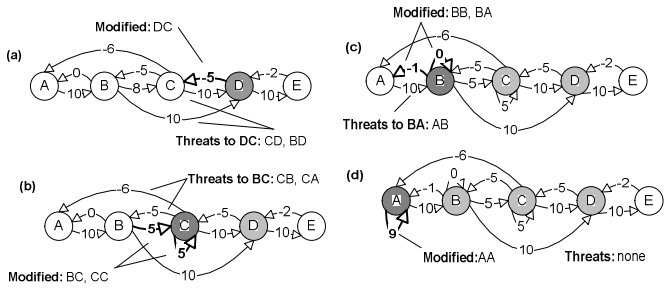
Lemma (STN-DBP) *Given a dispatchable STN with associated distance graph G,*
(i) Any tightening (or addition) of an edge AB with d(AB) = y, where y>0 and A?B, for all edges BC with d(BC)= u, where u <= 0, we can deduce a new constraint AC with d(AC) = y + u.
(ii) Any tightening (or addition) of an edge AB with d(AB)= x, where x <= 0 and A?B, for all edges CB with d(CB)= v, where v >= 0, we can deduce a new constraint CA with d(CA) = x+v.

Proof: (i) During execution, a negative edge AB propagates an upper bound to B of $ub_B = T(A) + d(AB)$. A non-negative edge CB propagates a lower bound to B of $lb_B = T(C) - d(BC)$. At execution time, changing AB will be consistent if $ub_B >= lb_B$ for any C, or $T(A) + d(AB) >= T(C) - d(BC)$, which implies $T(A) - T(C) < d(AB) + d(BC)$. Adding an edge CA of d(AB) + d(BC) to G encodes this constraint. Similar reasoning applies for case (ii) when a negative edge changes.ÿ

Recursively applying rules i and ii, when an edge changes in a dispatchable distance graph, will either expose a direct inconsistency or result in a dispatchable graph. This back-propagation technique only requires a subset of the edges to be resolved with the change, instead of all the edges which would happen if we were to re-compute the APSP every time an edge changed.

For example consider the series of STN back propagations required when the edge DC, shown in Figure 2a, is changed in the originally dispatchable graph. This change is back-propagated through the possible threats CB, and BD. The modified edges, BC and CC, resulting from this back-propagation is shown in Figure 2-b. The self loop CC in consistent and has no threats; however, the edge BC must be back-propagated through its threats, CB and CA. The results of this back-propagation modifies edges BB and BA, as shown in Figure 2-c. BA is threatened by AB. Back-propagating, resulting in the new dispatchable graph shown in Figure 2-d. In this example, only 5 propagation were required. Applying the Floyd-Warshall APSP algorithm would have required 125 propagations.

In order to apply DBP to distance graphs with uncertainty (DGUs), we introduce the idea of *pseudo-dispatchability*. If we ignore the distinction between contingent and requirement edges in the DGU, then the DGU is effectively converted into distance graph (DG). If this associated DG is dispatchable, then we say the DGU *pseudo-dispatchable*. If a DGU is both pseudo-dispatchable and pseudo-controllable, then its dispatchability is only threatened possible tightens to the uncontrollable durations at execution time. Furthermore, the pseudo-dispatchability of the DGU is maintained by recursively appling the STN-DBP rules.



We also introduce the term *pseudo-minimal dispatchable graph* (PMDG), which is DGU that is both pseudo-dispatchable and contains the fewest number of edges. The PMDG can be computed by applying either the "slow" STN reformulation algorithm introduced by [Muscettola 1998] or the "fast" STN reformulation algorithm introduced by [Tsarmrdinos 1998], to the DGU.

DBP Rules for STNUs

In this section we unify reduction and regression rules introduced by [Morris 2001] with the STN dispatchability back propagations (DBP) rules described in the previous section to form the DBP rules for STNUs. These rules form the core of our Incremental DC algorithm.

In this section we describe when the reduction and regression rules introduced by [Morris 2001] need to be applied. Consider the triangular STNU and associated DGU shown in Figure 1. Assume that the STNU is both pseudo-controllable and in an All-Pairs form.

Precede Case: u > 0:

The precede reduction prevents the propagations from either CB or BC, from squeezing the contingent link AB.

Definition (Precede Reduction [Morris 2001]) If *u > 0, tighten AC to x-u, and edge CA to v-y.*

Unordered Case: v ³ 0 and u £ 0:

The unordered reduction prevents propagations through edge CB from squeezing the contingent link AB, when C executes first, yet allows B to propagate an upper bound through BC, when B executes first.

A conditional edge[1], introduced by [Morris 2001], must be added to the DGU in this case. We call a DGU containing a set of conditional constraints, a Conditional Distance Graph with Uncertainty (CDGU). A conditional edge CA of <B,-t> specifies that A must wait at least *t* time units after A executes or until B executes, which ever is sooner. Note that the conditional edge is similar to a negative requirement edge.

Definition (Unordered Reduction [Morris 2001]) If *v ³ 0 and u = 0, apply a conditional constraint CA of <B, v-y>.*

In some cases the conditional edge is unconditional. The unconditional unordered reduction describes when to convert the conditional edge into a requirement edge.

Definition (Unconditional Unordered Reduction [Morris 2001]) *Given a STNU with contingent link AB Î [x,y] and the associated CDGU with a conditional constraint CA of <B,-t>, if x > t, then convert the conditional constraint CA into a requirement CA with distance –x.*

In order to prevent a conditional constraint from being violated at execution time, it must be regressed through the CDGU.

Lemma (Regression [Morris 2001]): *Given a conditional constraint CA of <B,t>, where -t is less than or equal to the upper bound of contingent link AB. Then (in a schedule resulting from a dynamic strategy):*

i.) If there is a requirement edge DC with distance w, where w ³ 0 and D ¹ B, we can deduce a conditional constraint DA of <w+t, B>.
ii.) If t < 0 and if there is a contingent link DC with bounds [x,y] and B ¹ C, then we can deduce a conditional constraint DA of <x+t, B>.

In order to maintain the dispatchability of the CDGU when a constraint changes, we only need iteratively apply all rules (STN-DBP, regression, and reductions) that pertain to that constraint. Table 1 summarizes the DBP rules used in our Incremental Fast-DC algorithm. This unified set of rules enables each type of propagation to be interleaved. We call the edges the change must be propagated through, threats. In the next section we describe the process of iteratively applying the DBP for STNUs to resolve all possible threats to the dispatchability.

```
function BACK-PROPAGATE(G,u,v)
1    if IS-POS-LOOP( u, v ) return TRUE
2    if IS-NEG-LOOP( u, v ) return FALSE
3    for each threat (x,y) to edge (u,v)
4       apply DBP rules to derive a new candidate edge (p,q)
5       if (p,q) is conditional
6             if dominated by a Req. edge (p,q) return TRUE
7             convert (p,q) to Req. edge as required by UUR
8       end if
9       resolve the edge (p,q) with G
10      if G is modified
11         if G is squeezed return FALSE
11         if ¬BACK-PROPAGATE(G,p,q) return FALSE
12      end
13   end for
14   return TRUE
```

Figure 3 Pseudo-Code for Back-Propagate

Incremental Dynamic Controllability

In this section we use the DBP rules in order to define an incremental algorithm for maintaining the dispatchability of a plan when one constraint changes. In the next section we extend this algorithm to reformulate unprocessed plans into a dispatchable form.

The function BACK-PROPAGATE, shown in Figure 3, maintains the dispatchability of a CDGU, G, when an edge (u,v) changes. The function BACK-PROPAGATE recursively applies the DBP rules shown in Table 1, until either it detects a direct inconsistency or until no more propagations are required.

The algorithm first checks if the edge (u,v) is a loop, (i.e. starts and ends on the same timepoint). If it is a positive loop, no more propagations are required and the algorithm returns true. If the edge is a negative loop, then an inconsistency is detected and the algorithm returns false.

Next the algorithm resolves all possible threats to (u,v) by applying the DBP rules in order to generate a candidate update edge (p,q). Two special conditions are considered if the candidate is a conditional edge. First, if the conditional edge is dominated by an existing requirement constraint,

If This Changes:	Must Back-Propagated Through (Threats)	Updates	Rule:
[-] Req. edge BA	1. any [+] Req. edge CB	[+/-] Req. edge CA	STN(ii)
	2. any Ctg Link CB	[+/-] Req. edge CA	PR
[+] Req. edge AB	1. any [-] Req. edge BC	[+/-] Req. edge AC	STN(i)
	2. any Ctg. Link CB *	[+/-] Cond. edge AC**	PR/UR
	3. any [-] Cond. edge BC of <-t,D>, where D ≠ A	[+/-] Cond. edge AC**	REG(i)
[-] Cond. edge BA of <-t,D>	1. any [+] Req. edge CB, where C ≠ D	[+/-] Cond. edge CA**	REG(i)
	2. any Ctg. Link CB , where B ≠ D	[+/-] Cond. edge CA**	REG(ii)

Table 1 STNU-DBP Rules

* same for both precede or unordered cases , ** convert any conditional edges into requirement edges as required by the UUR.
STN: STN-DBP, UR: Unordered Reduction, UUR: Unconditional Unordered Reduction, Ctg: contingent, Req.: requirement
PR: Precede Reduction, REG: regression

[1] [Morris 2001] used the term wait constraint.

then the algorithm returns true. Second, the algorithm converts the conditional edge into a requirement edge as required by the unconditional unordered reduction.

Next the algorithm resolves the candidate edge (p,q) with G by tightening or adding the corresponding edge as necessary. If this resolution modifies a constraint in G (i.e. is not dominated by an existing edge (p,q)), the algorithm checks if this tightening squeezes an uncontrollable duration, then recursively calls BACK-PROPAGATE to resolve the change.

After recursively resolving all threats, the algorithm returns true.

The function BACK-PROPAGATE is an Incremental DC maintenance algorithm. In the next section we present the Incremental DC Reformulation algorithm.

Fast-Dynamic-Controllability

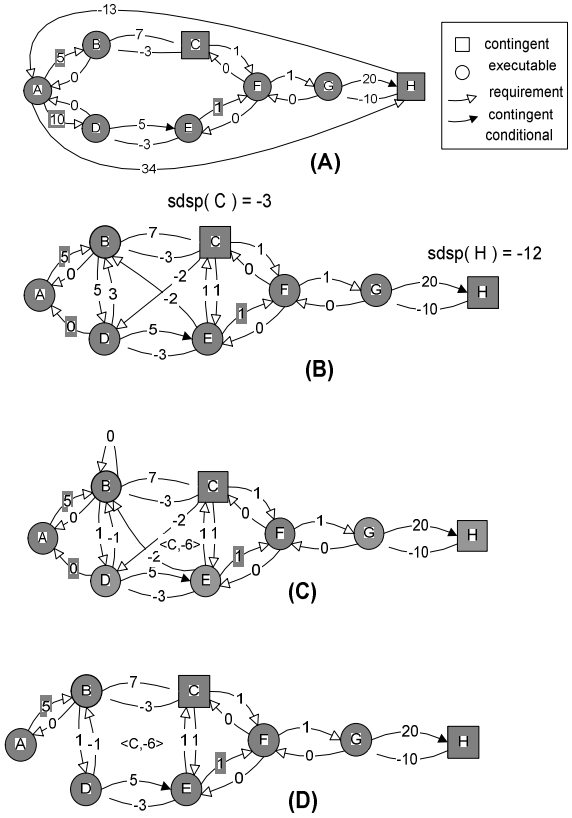
In this section we describe our Incremental-DC Reformulation algorithm (Fast-DC) which builds upon the Incremental Maintenance DC algorithm presented in the last section. We will use the example presented in Figure 4A to describe this algorithm.

The pseudo-code for the Fast-DC algorithm is shown Figure 5. If the STNU is dynamically controllable, then the Fast-DC returns a minimal dispatchable CDGU, otherwise it returns NIL.

First the Fast-DC algorithm converts the STNU into a CDGU, then computes the pseudo minimal dispatchable graph (PMDG) using the "slow" STN Reformulation Algorithm introduced by [Muscettola 1998]. If an inconsistency was detected the algorithm return NIL. The minimal pseudo-dispatchable graph for our example is shown in Figure 7-B. This PMDG is both pseudo-dispatchable and contains the fewest number of edges.

The CDGU is only dynamically controllable if it pseudo-controllable [Morris 2001]. Lines 3 checks if the contingent edges were squeezed during the process of converting the CDGU into a minimal pseudo-dispatchable graph. In our example, all contingent edges remain unchanged; therefore, the CDGU is pseudo-controllable.

Recall that our goal is to reformulate the graph to ensure that the plan can be dynamically executed. This reformulation is done by multiple calls the function BACK-PROPAGATE. The BACK-PROPAGATE function needs to be applied to any edge that may squeeze a uncontrollable duration. Each initial call of BACK-PROPAGATE causes a series of other edge updates. However, they will only update edges closer to the start of the plan. In order to reduce the amount of redundant work, we first initiate the back-propagations near the end of the plan and work the start of the plan. This way we slowly build up a solution where constraints near the end of the plan no longer need to be changed. In order to organize the back-propagations, we need to create a list of contingent timepoints ordered from timepoint that are executed near the end of the plan to the timepoints that are executed near the beginning of the plan. The contingent timepoints are ordered based on their Single-Destination Shortest-Path (SDSP) distance, sdsp(x). Specifically, the contingent timepoints are ordered from




```
function FAST-DC( G )
1  G ← STNU_TO_CDGU(G )
2  if ¬COMPUTE_PMDG(G)  return NIL
3  if ¬ IS_PSEUDO_CONTROLLABLE (G)  return NIL
4  Compute Bellman_Ford_SDSP( start(G), G )
5  Q ← ordered list of Ctg. T.P. according to the SDSP distances
6  while( ¬Q.IS-EMPTY() )
7     n ← Q.POP_FRONT()
8     if ¬ BACK_PROPAGATE_INIT( G, n ) , return NIL
9  end while
10 (optional)  COMPUTE_MPDG(G)
11 return G
```

Figure 5 Pseudo-code for FAST-DC

```
function BACK-PROPAGATE-INIT(G,v)
1   for all  pos. edges (u,v) into the Ctg. timepoint v
2     if ¬BACK_PROPAGATE(G,u,v)  return FALSE
3   end for
4   for all outgoing negative edges (v,u) from the ctg timepoint v
5     if ¬BACK_PROPAGATE(G,v,u) return  FALSE
6   end for
7   return TRUE
```

Figure 6 Pseudo-code for BACK-PROPAGATE-INIT

lowest to highest SDSP distances. The SDSP distances are computed in Line 4, and the contingent timepoints are ordered in Q in Line 5. In our example, the two contingent timepoints C and H have SDSP distances of -3 and -12

respectively. Therefore, H comes before timepoint C in the ordered list.

Next the Fast-DC algorithm initiates a series of back-propagation by calling the function BACK-PROPAGATE-INIT. This function initiates the back-propagation by applying all back-propagation rules to ensure that the uncontrollable duration associated with the contingent timepoint *v* is never squeezed during execution. Recall the contingent duration can only be squeezed by incoming positive edges or outgoing negative edges to the contingent timepoint. Lines 1-3 of this initiation function call BACK-PROPAGATE for all incoming positive edges into the contingent timepoint *v* and Lines 4-7 calls BACK-PROPAGATE for all outgoing negative edges from *v*.

Consider the series of back-propagations the Fast-DC algorithm uses to reformulate the CDGU between 7-B, 7-C. The CDGU does not contain threats that may violate contingent timepoint H, so no back-propagations are required. The contingent timepoint C, is threatened by the incoming positive edge EC. The edge EC is back-propagated through BC, resulting in a new conditional edge EB of <C,-6>. This contingent edge is then back-propagated through DE which modifies the requirement edge DB to -1. This negative requirement edge is then back-propagated through edge BD resulting in the edge BB of distance 4. This thread of back-propagation terminates here because of a positive self-loop.

The contingent timepoint C is also threatened by the outgoing negative edge CD of length -2. This negative requirement edge CD is back-propagated through BC which sets BD = 1. This positive requirement edges is then back-propagated through the negative edge DB, resulting modifying the self looping edge BB to 0. This thread of back-propagation is then terminated. The resulting dispatchable CDGU is shown Figure 4C. The back-propagation did not introduce an inconsistency; therefore, the original STNU is dynamically controllable.

The (optional) last step of the Fast-DC algorithm trims the dominated (redundant) edges from the CDGU. This is done by calling the basic STN reformulation algorithm. The resulting graph is a minimal dispatchable CDGU which can be executed by the dispatching algorithm introduced by [Morris 2001]. For example, the minimal dispatchable CDGU for the sample group plan is shown Figure 4D.

The algorithm described in this section is the incremental DC reformulation algorithm that we set out to describe.

Run Time Complexity of the FAST-DC Algorithm

In this section describes some preliminary experimental results for our Fast-DC algorithm.

The FAST-DC algorithm was run on a set of randomly generated STNUs that contained between 10 to 70 activities (20-140 timepoints) interconnected by a set of random (yet locally consistent) requirement edges. In our trials, 50% of the activities were marked uncontrollable.

Figure 8 shows the experimental run time of the Fast-DC algorithm for successful reformulations, plotted against the number of activities in the STNU. The tests were run on a 500 MHz IBM laptop with a Pentium III processor. The

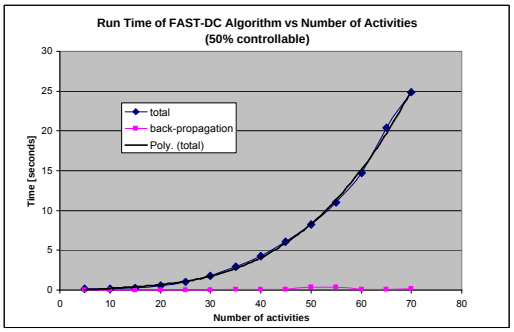


Figure 8 Experimental Run Time of Fast-DC algorithm

data labeled "total" is the total run time of the Fast-DC algorithm and the data labeled "back-propagation" represents the time the algorithm spent in the BACK-PROPAGATION function. The graph also shows a cubic regression curve fit to the total run-time.

The most interesting result is speed at which the algorithm performed the back-propagations.

Our test also shows that our Fast-DC algorithm experimentally runs in $O(N^3)$. This is not surprising if you consider the overall structure of the Fast-DC algorithm as follows. Our alogorithm is dominated by the "slow" STN reformulation algorithm done in step 1 and optionally in step 5.

1. Compute PMDG	$\Theta(N^3)$
2. Check for Pseudo-Controllability	$O(E)$
3. Run SSSP	$O(NE)$
4. Back-Propagation	polynomial
5. (optional) Re-compute PMDG	$\Theta(N^3)$

Note that the runtime of the FAST-DC algorithm can be directly improved by using the "fast" STN reformulation algorithm introduced by [Tsarmardinos 1998], which runs in $O(NE + N^2 \lg N)$ time.

In this paper we presented an Incremental DC Maintainance algorithm (BACK-PROPAGATE) and an Incremental DC Reformulation algorithm (FAST-DC).

Acknowledgements

This work has been funded by the DARPA NEST program under contract F33615-01-C-1896.

References

[CLR 1990] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Dechter 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61-95, May 1991.

[Morris 2001] P. Morris, N. Muscettola, and T, Vidal. Dynamic Control of plans with temporal uncertainty. In:

Proceedings of the 17th International Joint Conference on A.I. (IJCAI-01). Seattle (WA, USA).

[Muscettola 1998] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proc. Of Sixth Int. Conf. on Principles of Knowledge Representation and Reasoning* (KR '98), 1998.

[Tsarmrdinos 1998] I. Tsarmardinos, N. Muscettola, and P.Morris. Fast transformation of temporal plans for efficient execution. American Association for Artificial Intelligence (AAAI-98), 1998.

[Vidal 1996] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. Of 12th European Conference on Artificial Intelligence* (ECAI-96), pages 48-52, 1996.

[Vidal 1999] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistencies to controllabilities. Journal of Experimental & Theoretical Artificial Intelligence, 11:23-45, 1999

A Proposed Plan Execution Architecture for Advanced Life Support System Control

G. Biswas[1], P. Bonasso[2], S. Abdelwahed[1], E.J. Manders[1], D. Kortenkamp[2], J. Wu[1], and S. Bell[2]

[1]Dept. of EECS & ISIS, Vanderbilt University, Nashville, TN;

[2]NASA Johnson Space Center/ER2, Houston TX.

Advanced life support (ALS) systems require complex control strategies that can maintain stable system performance and balanced resources with small margins and minimal buffers. In closed-loop life support systems there are complex interactions between sub-systems such as air, water, food production, solids processing, and the crew. Recent research at NASA Johnson Space Center has led to significant insights into autonomous control of ALS systems [Leon *et al* 1997; Kortenkamp *et al* 2001; Schreckenghost *et al* 2002]. Routine control of an ALS system is well within the reach of current techniques. For example, the autonomous control system described in [Schreckenghost *et al* 1998] operated around the clock for 73 straight days during a 90 day crewed test with minimal human intervention and the autonomous control system for a recent test of an advanced water recovery system operated with minimal human intervention for over eighteen months[Bonasso *et al* 2002]. However, these control systems are not able to deal convincingly with the concurrent and interacting control of several subsystems, to coordinate the effective and efficient long term management of resources with the planning of mission activities, and to demonstrate effective recovery from significant anomalies. A solution to these issues is needed in order to demonstrate life support systems amenable to efficient long-duration missions such as the human exploration of Mars.

In this paper, we present a proposed multi-level computational architecture that integrates planning and hierarchical control schemes to develop a dynamic planning and control system that is reactive and fault-adaptive, but at the same time, is designed to manage resources for the duration of a long mission. The computational architecture adopts a novel approach to integrating components of the 3T control architecture developed at NASA JSC and Metrica [Bonasso *et al* 1997] with the hierarchical model-based multi-level control systems that have been developed at Vanderbilt University [Abdelwahed *et al* 2005]. Neither 3T nor the multi-level model-based control architecture alone present the complete solution for long-duration autonomous operations. The former lacks the dynamic models necessary to make efficient coordinated use of scarce resources and to maintain smooth transitions among controller states at finer granularity time scales, while the latter lacks domain procedural knowledge to understand the relations between mission goals and planned activities, and to allow the execution of specialized activities, such as maintenance and fault-recovery. Further, it is much harder to provide meaningful interfaces to the user through the control systems.

This integrated computational architecture combines the best of these two approaches. Our general design uses the dynamic models of model-based control architecture to inform the state-based procedural schemas during plan development and execution, as well as to carry out the dynamic control of the habitat subsystems. The 3T planner will provide overarching mission plans, while the 3T sequencer can instantiate procedures that would significantly increase the computational complexity associated with system analysis and decision making with the model-based control architecture.

The 3T planning module drives the supervisory control scheme. Given a top-level goal, such as "conduct habitat operations while supporting extravehicular activities (EVA)", the planner automatically generates a habitat plan for a given duration. The planner reasons in depth about goals, resources and sequencing constraints. It integrates mission goals with *a priori* knowledge, such as the crew schedule, EVA schedule, crop plantings and harvesting, and resource constraints. This knowledge is stored in the world model. During plan generation, the 3T planner draws from the task-resource consumption model of the Resource Manager (middle level), to take into account the dynamic effects of planning decisions. The resulting plan steps and ordering will be tailored to make the best use of scarce resources. Using the user interface capabilities of the

planner, the plan can be reviewed by mission control operators and the habitat crew before going into effect.

The middle level of our combined architecture consists of the 3T sequencer working in concert with the model-based supervisory controller. To execute the plan, the planner passes the next step in the plan for each area of the habitat to the 3T sequencer, which decomposes the plan step into RAPs that are further decomposed until the final sequences are at the level of the system controllers in the third level of the architecture, e.g., the Water Recovery System (WRS) or the crop chambers. An example sequence for the Air Revitalization System (ARS) was given in the previous section. A sequence to sustain crop growth might be 1) harvest a wheat crop, 2) harvest a soybean crop, 3) plant a soybean crop, 4) and harvest a salad crop. The selection of RAPs from the RAP library will be guided by dynamic constraints provided by the models in the model based supervisor also in the middle layer. The resulting sequences are then passed to the supervisory controller through the model information interface, which uses them as ordering constraints; e.g., the supervisor may force the ordering of a set of parallel tasks to ensure that required resources will be produced while not violating energy constraints, or it may adjust the duration of one of the steps as in the previous scenario. Using resource constraints, the supervisory controller transforms the sequence into a schedule of control specifications for the system level controllers, which then carry out the execution sequence for their respective systems (e.g., Air Revitalization (ARS) and Water Recovery (WRS)). Mission controllers and the crew have access to the state of the executing procedures via the system state information access module. This is especially needed when the crew carries out maintenance and ad hoc procedures that do not follow nominal operating schemes.

The system level controllers see each system as an input-output module, where material and energy are input to the system with the goal of producing desired states within the system and output that can be expressed in terms of material, energy, and performance quality parameters. The input-output mappings created by these controllers define utility-based multi-criterion objective functions that the lowest-level subsystem controllers employ to optimize dynamic behavior of subsystems in a way that they minimize the use of resources, while producing the necessary output. For example, given the levels of gases and the amount of energy available to the ARS during the above example sequence period, the system controller for the ARS will regulate the CO_2 and O_2 stores to maximize the CO_2 consumption to support the incineration operations.

Results of the execution from the system controllers are aggregated from the subsystem controllers in the bottom later and provided to the supervisor. In our current architecture, the subsystem controllers are designed to maintain set point control, i.e., maintain the operating region of their respective subsystems at levels and operating modes specified by the system controllers. The supervisor will update its dynamic models as well as pass the execution results to the sequencer as a set of execution states. The RAPS interpreter has the capability to determine new task sequences when faults occur in the system or in the face of unsuccessful execution of task steps. As RAPs sequences complete, the interpreter informs the planner which will update the plan and pass down the next plan step to be executed. Such an update may simply change start and stop times of steps while maintaining the original ordering. If the RAPs interpreter reports a failure of a plan step, as in the case of the faulty CDRA above, the planner may replan the mission steps, adding or omitting steps depending on the effect of the failed step on the overall mission objectives. As in plan generation, the task resource models of the supervisory controller will inform the replanning. As well, users will be able to modify the plan at their discretion as the crew did in the above scenario by requiring that the EVA take place as originally scheduled.

The principle of "cognizant failure" is still embodied in each level of the architecture. The system controllers provide robust regulation of the habitat subsystems, notifying the middle layer of any failing processes. The supervisory controller dynamically adjusts control schedules as the situation changes, informing the sequencer as to the state of tasks. The sequencer in turn serves as the mechanism to invoke alternate procedures as well as fault recovery procedures. Equally important, in light of severe failure, the sequencer will invoke "safing" procedures for the habitat subsystems, informing the planner which in turn will carry out replanning.

Additionally, the user has access to the levels of control where the aggregate of information and control stratagems is meaningful, and yet the complex details of such things as multi-criterion objectives functions remain hidden.

Scenario

We illustrate our proposed architecture through an example scenario. We begin with the assumption of a ninety-day mission plan that is scheduled in 28-day segments. Within the first 28-day period, the mission goal for the habitat might be "to conduct habitat operations while supporting an extravehicular activity (EVA) on day eighteen". An automated planning capability produces a plan of operation that includes tasks to maintain and operate the habitat, operate the water recovery system (WRS), air revitalization system (ARS) and crew quarters climate control, support

the required EVA, sustain crop growth, and ensure safe disposal of solid waste. Using resource models of the dynamics of the habitat subsystems the plan will make efficient use of power, air and water stores and habitat inventories.

Next, a reactive planning capability selects routine procedures for carrying out the first step of each part of the plan for each subsystem. For example, for the ARS:

1) Seven days of nominal operations.
2) Four days in high CO_2 consumption state to clear CO_2 reservoirs in preparation for incineration operations,
3) Four days in an extreme high CO_2 state to scrub the CO_2 resulting from incineration,
4) One day providing O_2 to tanks to be used for the upcoming 24 hour EVA on day eighteen, and
5) Resume nominal operations on day ten.

This sequence is then passed to a dynamic control execution capability that examines the existing resources for the ARS and suggests an extra day to ensure the O_2 tank level increases above a pre-determined value (say 10 kg). Since the extra day will still support the EVA on day eleven, the reactive planner makes no further changes to the ARS execution plan. The dynamic control executive issues time-ordered control specifications for all the habitat systems (WRS, ARS, Power generation, Biomass, etc.) and their corresponding subsystems commensurate with the procedures (i.e., partial plan sequences) from the reactive planner. The subsystem controllers execute the directives "optimally" taking into account the continuous dynamics of the respective subsystem for the first nine days. For example, a change detection algorithm might notice an increase in power usage in the CO_2 removal system (CDRA), but its subsystem controller is able to compensate the increase by decreasing the heater temperature a little, and also adjusting blower and pump speeds.

On day ten, however, the dynamic control executive determines that the CDRA behavior has continued to drift away from the nominal, and the system is operating sub-optimally. By now, the fault detection module has reliably established that there is a restriction in the CO_2 output line and also a leak is detected in the desiccant bed. The system controller has adjusted for this by reducing Oxygen Generation Assembly (OGA) and CO_2 Reduction System (CRS) (Sabatier) operating times, but if this trend continues, air quality in the crew chamber will start dropping below acceptable levels, or lot more energy will have to be directed toward the CDRA. With the night period approaching, this is not considered a good option (by the supervisory control predictor). This situation is reported by the supervisory controller to the RAPS (reactive planner) unit. This unit (the Sequencer) is told that it will now take five days to clear the CO_2 reservoirs. The reactive planner can make no adjustment that will compensate for the extra day and informs the planner. The planner sees the situation and determines there are options at this time such as (i) perform a CDRA repair and, (ii) drop the scheduled EVA activity.

The habitat planner considers the situation, and through its own analysis using its world model determines that a new plan that includes a two-day crew task for repair of the CDRA, which will create an O_2-restricted situation for a few days. As a result, the EVA activity is pushed back to day twenty, since one of the crew repairing the CDRA is also needed for the EVA. Furthermore, the astronauts are required to be cautious while exercising, e.g., none of the crew should exercise at the same time.

At this stage, using an interface to the planner, the habitat commander informs the planner that the EVA task cannot be slipped because it involves a communications experiment that depends on the relative orbits of the moon and the earth about the sun, a constraint unknown to the habitat planner. The planner, in further conference with the model-based resource manager, determines that if the crew completely omits their exercise period until after the EVA, the ARS can meet the incinerator and EVA requirements. The resulting habitat plan omits crew exercise from the crew plan and schedules the CDRA repair after the EVA.

When the CDRA repair takes place, the reactive planner will select an appropriate repair procedure for the crew and a set of modes for ARS and other affected subsystems, and the dynamic controller will execute these changes efficiently. For example, oxygen generation may be suspended, thus reducing the water requirement from the WRS during the repair period. As well, during the repair, the reactive planner will serve as the subsystem level interface to the dynamic controller.

When the repair is complete, the dynamic controller will verify the normal operation of the CDRA and inform the reactive planner, which in turn informs the habitat planner. The habitat planner will adjust the inventory of materials used in the repair and replan if necessary.

A key observation from this scenario is that once anomalous situations are detected, mechanisms kick in at different levels to attempt to contain and compensate for the fault, without having to sacrifice mission goals. For less critical faults of small magnitude, the subsystem controllers can compensate for the change in behavior. At the next level, the system controller may redistribute resources or, if possible reassign some tasks, to keep the system performance and output at different levels. Then the supervisory controller jumps in to determine if it can impose non-critical restrictions to avoid over draining of

resources or reduction in effort without significant loss of capabilities. If the problems persist, the reactive planner or the replanner may be invoked to determine new plans. Last, mission control or the crew may want to change some of the mission goals to avoid potential problems. In all of these situations, decisions made at the top take precedence, which imply that the lower level units, especially the lower-level controllers have to change their strategy to satisfy the new requirements.

References

Abdelwahed, S., J. Wu, G. Biswas, J. Ramirez and E. J. Manders, "Online Fault Adaptive Control for Efficient Resource Management in Advanced Life Support Systems," *Habitation: International Journal for Human Support Research*, Vol. 10, No. 2, pp. 105-116, 2005.

Bonasso, R.P., R. J. Firby, E. Gat, D. Kortenkamp, D. Miller and M. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents," *Journal of Experimental and Theoretical Artificial Intelligence*, Vol. 9, No. 1, 1997.

Bonasso, R. P., David Kortenkamp and Carroll Thronesbery, Intelligent Control of a Water Recovery System. In *AI Magazine*, Vol. 24, No. 1, Spring 2003.

Kortenkamp, D., R. Peter Bonasso and Devika Subramanian, "Distributed, Autonomous Control of Space Habitats," *IEEE Aerospace Conference*, 2001.

Leon, J., David Kortenkamp and Debra Schreckenghost, "A Planning, Scheduling and Control Architecture for Advanced Life Support Systems," *Proceedings of the NASA Workshop on Planning and Scheduling in Space*, 1997.

Schreckenghost, Debra, Mary Beth Edeen, R. Peter Bonasso, and Jon Erickson, "Intelligent Control of the Product Gas Transfer for Air Revitalization," *Proceedings of the 28th Conference on Environmental Systems*, 1998.

Schreckenghost, Debra, Carroll Thronesbery, R. Peter Bonasso, David Kortenkamp and Cheryl Martin, "Intelligent Control of Life Support for Space Missions," in IEEE *Intelligent Systems Magazine*, Vol. 17, No. 5, September/October 2002.

Robust Goal-oriented Behavior in Surprising Environments

An ICAPS 2005 Position Paper

Marshall Brinn, Mark Burstein, Robert Bobrow
BBN Technologies
{mbrinn, mburstein, rusty}@bbn.com

Abstract

Software systems are often vulnerable to failure when confronting unforeseen circumstances. Considering that real world 'open' environments are endlessly varied and dynamic, making systems robust in such environments is particularly challenging: the set of conditions cannot be enumerated and handled at design time. Our analysis of different classes of surprise suggests an architecture for achieving robust goal-oriented behavior in open environments. In particular, we content that by adding a reflective layer to standard OODA-loop control processing, recovery from a variety of unanticipated events can be achieved. We present here the elements of that architecture and some results from applying this approach to an example domain.

1. Overview

The nature of software design is that systems are built and operate with particular expectations of the environment in which they will run. Within the constraints of these expectations, the system may perform as desired. However, software systems are notoriously brittle in facing the unanticipated. The behavior may range from the inefficient to the unreasonable to outright failure. This vulnerability stems naturally from the inability on the part of the designer to anticipate every eventuality of an open environment. Despite this, what is desired, at the very least, is a system that behaves reasonably in that it can:

- Not fail the first time it encounters something unanticipated
- Not be surprised in similar circumstances thereafter
- Improve response in each subsequent encounter

Surprise results from encountering observations significantly counter to expectations. A prerequisite, then, for coping with surprise is the ability to be surprised, and thus to have expectations. Such a difference between expectation and observation may be categorized in terms of:

- *Quantitative*: The observation represents a low-probability contingency not explicitly planned for.
- *Qualitative*: The observation represents a zero-probability event, contradicting the current world model.

In either case, some inadequacy in the current model may be the source of the surprise. In the quantitative case, the environment may have changed in some way not yet reflected in models. In the qualitative case, the model may be limited in its scope by certain assumptions, precluding the representation of some events.

In seeking behaviors that are reasonable by the above criteria, we contend that systems must have some flexibility in order to adapt. They must support multiple goals to allow trade-offs of different courses of actions. They must, further, be able to reliably predict the state of the world (as a result of and independent of its actions) in order to plan and act successfully. In the face of surprise, they must be able to monitor and correct their own performance and predictions. Moreover, they must make their model assumptions explicit, to allow for identifying and recovering from the sources of surprising events by questioning these assumptions.

The architectural approach described below seeks to provide software with these attributes in order to enable them to deal reasonably in open environments.

2. Architectural Approaches

The standard cognitive control loop seeks to take actions expected achieve its goals. That is, it relies on some predictive capability to project expected outcomes of potential actions and evaluate the utility of these projections. Underlying this

control loop is another process of achieving reliable prediction. A two-tier control mechanism is suggested as illustrated in Figure 1:

- *Environmental Controller (EC)*: Work to achieve goals by acting in world based on predictions
- *Cognitive Controller (CC)*: Work on world model to improve predictions.

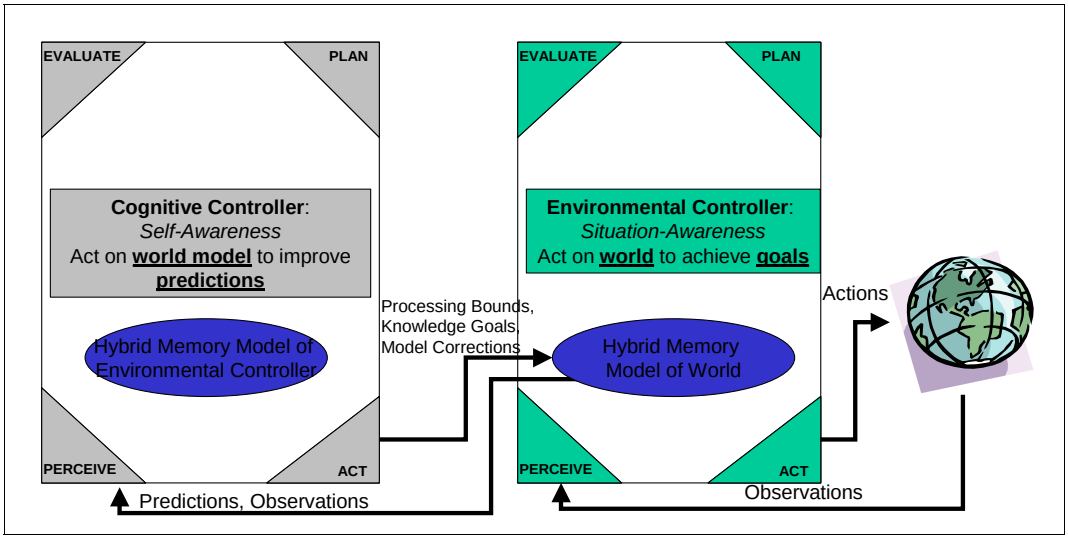


Figure 1: The Cognitive Controller (CC) provides a reflective layer on top of the Environmental Controller (EC), updating its predictive models and providing auxiliary 'knowledge discovery' goals to allow the EC to better operate in the world.

The EC represents a standard OODA loop, developing and acting on plans evaluated from a set of goals and a predictor. We have developed a generic EC that contains a genetic algorithm searching the space of plans (sets of future actions) evaluated to have the best predicted utility.

The CC represents a reflective layer on top of the EC, monitoring its predictions and observations and seeking to improve the EC's predictor accordingly. The CC structure is parallel to that of the EC, but with different goals, sensors and actions. Whereas the EC observes, plans and acts in the world, the CC observes the EC and plans and acts to improve the EC's predictive models.

It also may take control actions on the processing parameters of the EC, manipulating resource and time constraints to allow the EC to operate more effectively in the current environment. The CC is also built on a genetic algorithm; however, it searches the space of qualitative and quantitative changes to the current predictor to better match the recent and long-term observations. The CC is provided with some rough default qualitative models of the world. Initially, it seeks to fill in the quantitative details of these models by fitting to observation data. In addition, however, it seeks to expand these qualitative models by questioning their assumptions systematically to determine possible explanations for anomalous observations, be they significant deviations between observation and expectation or violations of the current qualitative model. We call the parameters by which these qualitative models may be expanded a qualitative metaphysics, containing a broad set of possible contingencies that are initially precluded. Occam's razor encourages us to try to fit the existing data into simple explanations; however, the model should be expanded (temporarily or permanently) by a particular parameter provides a better fit to the observation. For example, by default we assume all actuators and sensors are operating effectively. However, we allow the CC to search models in which the possibility that they may be broken or biased in some way to be considered.

In order for the CC to do its job, it needs adequate data flowing from the CC on actions and their observed outcomes. To that end, the CC will want to not only provide updates to the predictor, but to encourage the EC to take actions that will help the CC to improve its predictor. The relationship of the EC and the CC is one of influence rather than command, and thus it passes 'auxiliary goals' to the EC to include into its overall evaluation of prospective plans. For example, the

CC should not direct the EC to turn on a given actuator; rather it may ask the EC to view more favorably plans that exercise a given actuator, but only if it falls into the broader set of EC goals.

We have developed a generic test framework in which to develop and test robust behavior in open environments. The test framework contains a world simulator that allows for scripted changes to the world and measurements of the world as available to the EC.

3. Example Domain: Temperature Control

To illustrate our approach, we have implemented a system controlling the temperature of a room to desired levels. The components of the domain include:

- *Goals*: Comfort: Maintain the room temperature to as close to 60 degrees as possible, Economy: Minimize time of having AC/Heat running, Simplicity: Minimize the number of times we switch AC/Heat on/off
- *Sensors*: Measurements are available of temperature inside the room and outside the building
- *Controls*: Two heaters are available (one stronger than the other) and one AC

Initially, the sensors are presumed to work, but the qualitative metaphysics understand they may be noisy, biased, or broken. The actuators are presumed to work with qualitative effect that being on raises/lowers room temperature (by how much, how fast are quantitative parameters to be learned and adjusted over time).

At system startup, the CC does not have enough information about the behaviors of the actuators to make a reliable predictor and encourages the EC to sample the space of actuators. The EC obliges since it can't find any plan that has a good predicted outcome. Once some measurements are available for the actuators, a reasonable predictor is available, and the EC is able to establish classic 'saw tooth' control. Soon, the CC realizes that by including the outside temperature (originally ignored for simplicity) in its predictive model, it provides a much better correlation to observed data. The world simulator is subsequently scripted to include different surprising events, including very noisy temperature measurements (which the CC handles by increasing sampling and smoothing) and a broken heater (which the CC learns to predict will have no effect, and the EC will cease to select it in its plans).

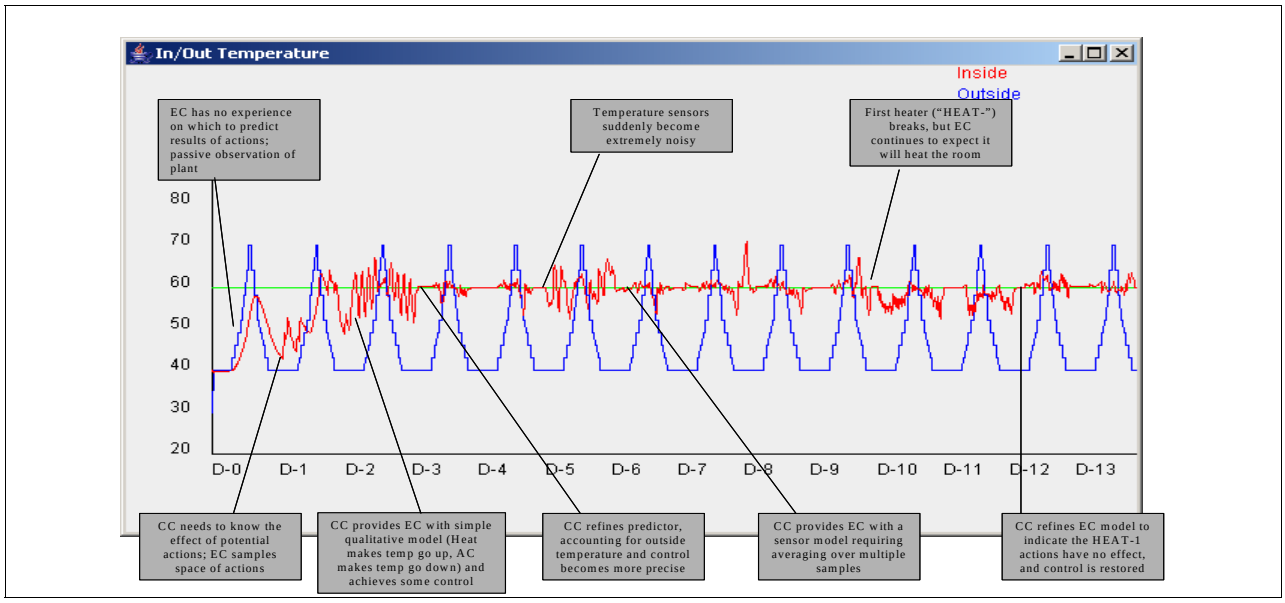


Figure 2: Results from the temperature control simulation, showing the CC providing updated predictors to allow the EC to continue to control temperature in the face of surprising events (e.g. noisy sensors and broken actuators).

Figure 2 illustrates a particular scenario in the life of this temperature control domain, including responses to these and other scenarios.

4. Conclusion

Although we are still in the early stages of exploring this paradigm, we anticipate that hybrid memory and reasoning models will provide additional degrees of robustness to different kinds of surprising, dynamic conditions. In particular, by appropriate layering of case-based, statistical and rule-based/logical models, we contend that the vulnerabilities of one approach may be covered by the strength of another. Further, we are actively investigating incorporating models of uncertainty and lack of knowledge into these predictive models. Specifically, we are seeking to formalize the notion of 'knowledge actions' as another potential action for the EC to take that will provide more information, allowing it and the CC to make better predictions. Additionally, we are investigating how the CC can work to manage the allocation of resources (particularly time) to the EC, in terms of optimal times for planning relative to unfolding execution.

These future efforts not withstanding, several aspects of our approach are suggestive towards a general approach to robust handling of surprise:

- While prediction provides a key to good control, there is a notion of 'good enough prediction' within which reasonable control and response can be achieved (if subject to subsequent fine-tuning).
- Environmental controllers should ascribe benefits to their actions not only by the utility of their predicted outcome, but the information they will provide a reflective layer to allow them to make better predictions in the future.
- The distinction between qualitative and quantitative provides a strong foundation for handling different kinds of surprises.

5. Acknowledgements

This work was performed under DARPA/IPTO contract HR0011-04-C-0078.

An Extension to PDDL: Actions with Embedded Code Calls

Okhtay Ilghami
University of Maryland at College Park
okhtay@cs.umd.edu

J. William Murdock
IBM Watson Research Center
murdockj@watson.ibm.com

Abstract

In most existing planning systems, *plan generation* and *plan execution* are two entirely separate phases. This fact has had a huge effect on the way PDDL has been developed so far: It is assumed that the two aforementioned phases are separable. In this paper, we investigate the characteristics of the domains in which this separation is impossible. We also propose extensions to PDDL, both syntactically and semantically, which will make PDDL capable of describing such domains.

Introduction

In traditional planning systems, *plan generation* and *plan execution* are separated. This simplifying separation is based on the usually unrealistic assumption of *perfect knowledge*. In reality, however, there can be two reasons for the lack of such knowledge: The state of the world may not be fully known, or actions may have unpredictable effects. In either case, the planner may need to execute some of the steps of the generated plan to observe more about the state of the world and/or the preconditions and effects of the actions in those steps. The consequence of executing actions while generating plans is that the state of the world is changing and these changes are irreversible. For example, consider a robot trying to get out of a maze of unknown configuration. The robot may initially move around to determine where the walls are, and then start planning using that information.

PDDL (Fox & Long 2001b), a purely descriptive and platform-independent language, has recently become the de facto standard for defining planning problems and evaluating plans. Perfect knowledge is assumed in the existing PDDL specification. In this paper, we propose a way to extend PDDL so that it can be used to describe domains where the planner does not necessarily have perfect knowledge, and it may execute code calls within the actions to gain knowledge that may be useful in generating later steps of the plan. In the next section, we discuss the syntactic and semantic aspects of such an extension, and how it affects the notion of a valid plan. We then conclude the paper by sections on related work and future directions.

Adding Actions with Code Calls to PDDL

In this section, we explain how our extension to PDDL is formulated, and how it affects the semantics of a domain and a plan. We propose three extensions to the PDDL syntax:

i) Two new possible requirements for a PDDL domain: `:code-call-actions` to declare that a domain contains actions with code calls, and `:code-call-durative-actions` to declare that a domain contains durative actions with code calls.

ii) A new kind of instantaneous action, denoted by the keyword `:code-call-action`: These actions look exactly like ordinary PDDL actions, except for one extra construct, denoted by `:code` in their definition. This construct consists of a predicate analogous to a function call, and a list of typed returned values. The names of these variables must start with $\#$. These variables can be used in the effects of the action, as any other variable.

iii) A new kind of durative action, denoted by the keyword `:code-call-durative-action`: These actions look exactly like PDDL durative actions, except for one extra construct, denoted by `:code` in their definition. This construct can be temporally annotated, and it consists of a predicate analogous to a function call, and a list of typed returned values. The names of these variables must start with $\#$, and they can be used in the effects of the action.

In this paper we discuss only instantaneous actions with embedded code calls. It is straightforward to generalize our discussions both syntactically and semantically to include durative actions with embedded code calls. From now on we use the term *action* to refer only to *instantaneous actions*.

Consider the k-armed bandit problem (Berry & Fristodt 1985; Kaelbling, Littman, & Moore 1996). In this problem, there are k gambling machines and a robot. When the robot pulls the arm of the ith machine, the machine pays off either a dollar with the constant probability p_i or nothing with the probability $1–p_i$. The robot is allowed to have a fixed number of pulls, h, and p_is are not known to the robot in advance. The goal is to maximize the total pay off. Since the robot does not know the p_is, it cannot simply plan and execute separately. The robot can gain information (i.e., get a better approximation of p_i) by pulling the arm of the ith machine. This is an irreversible action, since it decreases the number of allowed pulls by one. In Figure 1, we show how this problem can be defined in our proposed extension of PDDL. Action `pull` uses a code call `do-pull` to pull a lever and see what happens. This code call is defined in the line tagged `:code`. The code call returns a single numeric

```
(define (domain K-armed-bandit)
 (:requirements :fluents :adl
  :code-call-actions)
 (:types lever number)
 (:functions (pulls) (cash))
 (:code-call-action pull
  :parameters (?l - lever)
  :precondition (> (pulls) 0)
  :code ((do-pull ?l) (#pay - number))
  :effect (and (decrease (pulls) 1)
               (increase (cash) #pay))))
```

Figure 1: Encoding the K-armed bandit problem

value. This value is used to instantiate variable `#pay`. Although this code call returns only one numeric value, code calls can return several return values with arbitrary types.

In Figure 2 we show how to define the problem of a robot trying to get out of a maze. In this problem, the robot has an initial position and wants to get to a final position. It has four possible actions to move north, south, east or west. The robot cannot plan in advance and execute later since it does not know where the walls are located. The only way the robot has to figure out if there is a wall to its north is to try to move north. The return value `#ret` indicates whether the robot has moved north or has been stopped by a wall.

Actions with embedded code-calls are useful to describe actions with these three characteristics: First, the outcome of an action is not known in advance. This can happen for several different reasons, for example imperfect knowledge about the current state of the world. Since the effects of the action are not known fully in advance, it cannot be defined as a classical planning action. Second, although the outcome of the action is not known in advance, the planner may speculate that executing it may help to reach the goal (either directly or indirectly via the information gained by executing the action and then observing the results). Third, executing the action changes the state of the world.

Embedded code calls are particularly useful for sensing actions (i.e., actions that give the planner some information about the outside world). In real world, sensing actions can be performed by some external agent. For example, the planner may control a robot that can measure the temperature, navigate the outside world, etc. We define an *oracle* to be an abstraction of this external agent in our framework. Anytime the planner may wish to actually execute something, it does so by sending a message to the oracle. We assume that the oracle executes the action and returns the results of doing so back to the planner. In our proposed extension, each planning domain is coupled with an oracle O, which is responsible for processing code calls.

PDDL is a purely descriptive and implementation-independent language. Our code call syntax preserves these traits; our syntax is a general format for planners to communicate with oracles (i.e., the external agents). A code call is a general form of a function call. It is a predicate with two elements. The first element is analogous to a function call (function name followed by its arguments). The second element is a list analogous to the return values of the function

```
(define (domain robot-and-maze)
 (:requirements :fluents :adl
  :code-call-actions)
 (:types number)
 (:functions (x) (y))
 (:code-call-action north
  :parameters () :precondition ()
  :code  ((move (x) (+ (y) 1))
                (#ret - number))
  :effect (when (= #ret 1)
                (increase (y) 1)))
 (:code-call-action south
  :parameters () :precondition ()
  :code  ((move (x) (- (y) 1))
                (#ret - number))
  :effect (when (= #ret 1)
                (decrease (y) 1)))
 (:code-call-action east
  :parameters () :precondition ()
  :code  ((move (+ (x) 1) (y))
                (#ret - number))
  :effect (when (= #ret 1)
                (increase (x) 1)))
 (:code-call-action west
  :parameters () :precondition ()
  :code  ((move (- (x) 1) (y))
                (#ret - number))
  :effect (when (= #ret 1)
                (decrease (x) 1))))
```

Figure 2: Encoding the robot and maze problem

call. We assume whenever a planner decides to put an action with an embedded code call in a plan, the appropriate code call is executed, and the oracle instantiates the variables in the return value list. This instantiation is an abstraction of what happens in the real world: The planning system asks an outside agent for some information, and then continues planning using the information provided by that agent. The effects of asking the external agent to do so can be mentioned in the effects part of the action.

The details of how the actual oracle is implemented is outside the scope of PDDL. A Java implementation of the oracle may translate the code call to a member function of an object, while a Lisp implementation may translate the code call to a Lisp expression. From the point of view of someone using PDDL to define a domain, the underlying structure and implementation of the attached oracle must be transparent.

In our proposed extension, whenever an action with an embedded code call is used in a plan, the return values of the code call must be listed in the generated plan, in the same order they are mentioned in the domain definition, after the action. For example, if the robot uses the action `north` in the problem defined in Figure 2 in order to try to go north and it fails, the corresponding action listed in the plan will look like `(north)[0]`, and if the move is successful it will look like `(north)[1]` rather than simply `(north)`.

Whenever there are actions with embedded code calls listed in a planning domain, satisfaction of the precondi-

tions and achieving all the goals are not the only measures of validity of a plan. Since the planner cannot backtrack on certain actions, once it decides to invoke them using the corresponding code calls to the oracle it *must* include them in the generated plan. Therefore, the validity of a plan is defined with respect to the oracle O in the planning domain: *A valid plan must include all the actions with embedded code calls corresponding to the code calls it made to the oracle and the return values it got, in the same order, in addition to the traditional conditions for validity*.

Related Work

One approach to operating in domains where the starting state is not completely known or there is uncertainty in the effects of some actions is to interleave reasoning about what actions to execute with the actual execution. For example, reinforcement learning techniques (Watkins & Dayan 1992; Kaelbling, Littman, & Moore 1996) select actions using a simple numerical policy and incrementally learn improved policies based on the rewards obtained from performing actions. Agent centered search techniques (Korf 1990; Koenig 2001) also interleave execution and learning; unlike reinforcement learning, these techniques also perform some planning/search. Reflection using functional process models (Stroulia & Goel 1995; Murdock & Goel 2003) also interleaves reasoning and action, using a variety of planning and learning algorithms. We feel that our proposed mechanism for code calls is potentially useful for all of these methods.

There are also a variety of approaches that deal with incomplete knowledge and uncertain actions without interleaving reasoning and execution. For example, SGP (Weld, Anderson, & Smith 1998) performs contingency planning (i.e., it produces plans that include sensing actions and restrictions on which actions are performed depending on the results of those sensing actions). Similarly, CGP (Smith & Weld 1998) performs conformant planning (i.e., it produces plans that accomplish the goal regardless of the outcome of any actions). In general, approaches that handle incomplete knowledge and uncertain effects without interleaving reasoning and acting have significant disadvantages. For example, contingency plans can be very large and time-consuming to construct, and conformant plans frequently do not exist or have low quality. However, in some domains these disadvantages are outweighed by the benefits of not having to execute any actions until all planning is complete.

Code calls are not essential for planning systems that do not interleave planning and acting. However, even for those planning systems, it may be useful to integrate information needed for planning with information needed for execution. Such an integration can be useful for executing plans after planning is complete. The ability to execute plans is not important for traditional planners, but is important for many real-world applications of planning (e.g., web services, robotics, interactive agents). To the extent that PDDL can be valuable for serving as a common domain language for these sorts of systems, the addition of a mechanism for code calls seems productive.

Future Directions

This paper is meant to be a first step toward extending PDDL to handle situations in which plan generation and plan execution cannot be separated. There are still many unanswered questions and interesting topics for future research:

Although oracles are well-defined abstract entities in theory, there are issues to be addressed while implementing them in practice. Some of the questions that should be answered are: What are the effects of different programming paradigms, such as structured programming, object-oriented programming, and functional programming on the process of implementing an oracle? Do these different paradigms, in practice, affect the planning process too? What are the conceptual and practical side-effects of the assumption that there is an oracle attached to each planning domain?

Another interesting topic is the characteristics that an actual planning system needs in order to be able to handle actions with embedded code calls. Does our proposed PDDL extension have any unexpected consequences when employed in such a system? If so, are there revisions that can address these consequences?

Another question to be answered is how to enhance the framework we provided here to support other extensions proposed for PDDL. For example, are there any conceptual or practical problems in adding actions with embedded code calls to, for example, PDDL+ (Fox & Long 2001a)?

References

Berry, D. A., and Fristodt, B. 1985. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall.

Fox, M., and Long, D. 2001a. PDDL+ level 5: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects. Technical report, University of Durham, UK.

Fox, M., and Long, D. 2001b. PDDL2.1: An extension to PDDL for modelling time and metric resources. Technical report, University of Durham, UK.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. P. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Koenig, S. 2001. Agent-centered search. *Artificial Intelligence Magazine* 22(4):109–131.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2–3):189–211.

Murdock, J. W., and Goel, A. K. 2003. Localizing planning with functional process models. In *Proceedings of the 13th Int'l Conference on Automated Planning and Scheduling*.

Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 889–896. AAAI Press.

Stroulia, E., and Goel, A. K. 1995. Functional representation and reasoning in reflective systems. *Journal of Applied Intelligence* 9(1):101–124.

Watkins, C. J. C. H., and Dayan, P. 1992. Technical note: Q-learning. *Machine Learning* 8(3).

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 897–904. AAAI Press.

Optimized Execution of Action Chains through Subgoal Refinement*

Freek Stulp and Michael Beetz
Intelligent Autonomous Systems Group, Technische Univerität München
Boltzmannstrasse 3, D-85747 Munich, Germany
{stulp,beetz}@in.tum.de

Abstract

In this paper we propose a novel computation model for the execution of abstract action chains. In this computation model a robot first learns situation-specific performance models of abstract actions. It then uses these models to automatically specialize the abstract actions for their execution in a given action chain. This specialization results in refined chains that are optimized for performance. As a side effect this behavior optimization also appears to produce action chains with seamless transitions between actions.

Introduction

Many plan-based autonomous robot controllers generate chains of abstract actions in order to achieve complex, dynamically changing, and possibly interacting goals. To allow for plan-based control, the plan generation mechanisms are equipped with libraries of actions and causal models of these actions, specifying what it can achieve, and under which circumstances. By specifying these actions abstractly, they apply to a broad range of situations, reducing the search space for planning.

The advantages of this abstraction, however, come at a cost. Because planning systems consider actions as black boxes with performance independent of the prior and subsequent steps, the system cannot tailor the actions to the contexts of their execution. This often yields suboptimal behavior with abrupt transitions between actions, causing suboptimal performance. The resulting motion patterns are so characteristic for robots that people trying to imitate robotic behavior will do so by making abrupt movements between actions.

Let us illustrate these points using the autonomous robot soccer scenario depicted in Figure 1. To solve this task, the planner issues a three step plan, also shown in the figure. If the robot naively executes the first action (sub-figure 1b), it might arrive at the ball with the goal at its back, an unfortunate position from which to start dribbling towards the goal. The problem is that in the abstract view of the planner, being at the ball is considered sufficient for dribbling the ball and the dynamical state of the robot arriving at the ball is considered to be irrelevant for the dribbling action. What we would like the robot to do instead is to go to the ball *in order* to dribble it towards the goal afterwards. The robot should, as depicted in the sub-figure 1c, perform the first action sub-optimally in order to achieve a much better position for executing the second plan step.

*The work described in this paper was partially funded by the Deutsche Forschungsgemeinschaft in the SPP-1125.

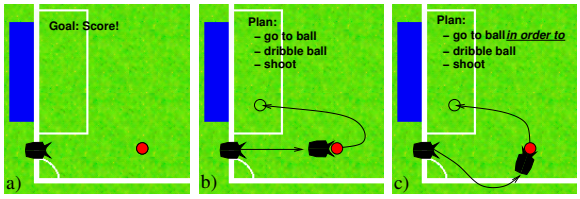


Figure 1: Alternative executions of the same plan

In this paper we propose a novel computational model for plan execution that enables the planner to keep its abstract action models and that optimizes action chains at execution time, shown in Figure 2. The basic idea of our approach is to learn performance models of abstract actions off-line from observed experience. Then at execution time, our system determines the set of parameters that are not set by the plan and therefore define the possible action executions. It then computes for each abstract action the parameterization such that the predicted performance of the action chain is optimal. This is done by refining the intermediate state between subsequent actions.

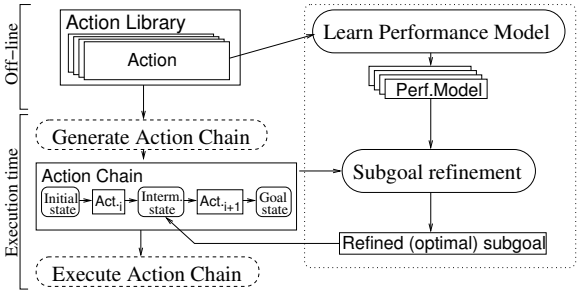


Figure 2: System Overview

Learning performance models

To optimize action chains, we need performance models of each abstract action that predict the performance, e.g. time, given specific situations. The execution time of the `goToPose` action, which is based on computing a Bezier

curve and trying to follow it as closely as possible, depends on the distance (*dist*) and angle (*angle2dest*) to the destination, as well as the angle between the current orientation and the desired orientation at the destination (*angle@dest*).

The performance function for this action (`goToPose.perform`(*dist,angle2dest,angle@dest*)→*t*) is learned by model trees from observed experience acquired in a simulator, similar to (Belker *et al.* 2002). Model trees are a generalization of decision trees. They are functions that map continuous or nominal features to a continuous value. The function is learned from examples, by a piecewise partitioning of the feature space. A linear function is fitted to the data in each partition.

In Figure 3, we depict an example situation in which *dist* and *angle2dest* are 2.0m and 0°. The plots depict the predicted execution time for different angles of approach (*angle@dest*). The model tree's piecewise linear approximation is obvious in the Cartesian plot. The polar plot more clearly shows the dependency of predicted execution time on the angle of approach for the example situation. Note that the model has learned to predict performance for all situations the action can perform, and not just this specific situation.

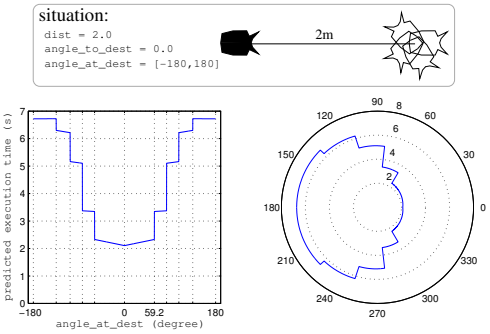


Figure 3: Temporal prediction with performance models

Automatic subgoal refinement

The set of possible intermediate states between two actions is limited by the post-conditions of the first, and preconditions of the second action. The actual intermediate state simply arises after having executed the first action, as can be seen in Figure 1b. As it turns out, this state leads to suboptimal overall performance. From all possible intermediate states, our subgoal refinement system chooses the state that optimizes the predicted performance of the action chain.

In our example, the only variable free for optimizing is the angle of approach of the intermediate position. Our system automatically determines this by reasoning about the performance model (which variables influence performance), the pre- or post-conditions of the subsequent action (which variables are bound), and the current state of the world (which variables are fixed in the current state).

In Figure 4 the first two polar plots represent the performance of the two individual actions for different values of angle of approach. The overall performance is computed by adding those two, and is depicted in the third polar plot. The fastest time in the first polar plot is 2.1s, for angle of approach of 0.0°. However, the overall time is 7.5s. These values can be read directly from the polar plots. This value is not the optimum overall performance, which is actually 6.1s, as can be read from the third polar plot. Below the polar plots, the situation of Figure 1 is repeated, this time with the predicted performance for each action.

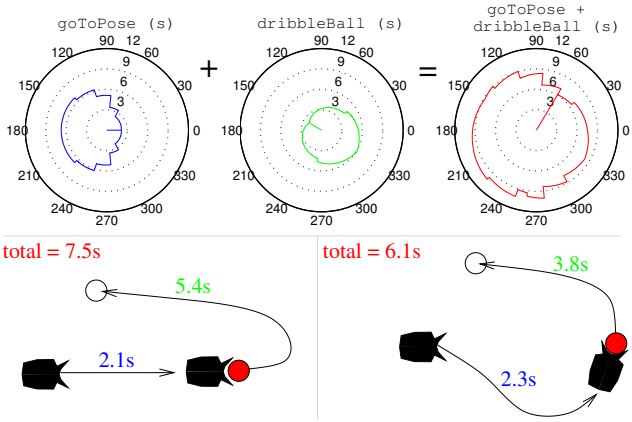


Figure 4: Computing the optimal intermediate goal.

Results

To determine the influence of subgoal refinement on the overall performance of the action chain, we generated 1000 situations with random robot, ball and final goal positions. The robot executed each navigation task twice, once with subgoal refinement, and once without. The overall mean improvement is 12%. We have split these cases into those in which the subgoal refinement yielded a higher, equal or lower performance in comparison to not using refinement. This shows that the performance improved in 505 cases, and in these cases causes a 23% improvement. In 485 cases, there was no improvement. This is to be expected, as there are many situations in which the three positions are already optimally aligned (e.g. in a straight line), and subgoal refinement will have no effect. A small decrease in performance (6%) occurred in 10 cases.

Conclusion and Future Work

On-line optimization of action chains allows the use of planning with abstract actions, without losing performance. Optimizing the action chain is done by refining under-specified intermediate goals, which requires no change in the planner or plan execution mechanisms. To predict the optimal overall performance, performance models of each individual abstract action are learned off-line and from experience, using model trees. It is interesting to see that requiring optimal performance can implicitly yield smooth transitions in robotic and natural domains, even though smoothness in itself is not an explicit goal. Applying subgoal refinement to the presented scenario yields good results. However, the computational models underlying the optimization are not specific to this scenario, or to robot navigation.

References

T. Belker, M. Beetz, and A.B. Cremers. Learning action models for the improved execution of navigation plans. *Robotics and Autonomous Systems*, 38(3-4):137–148, 2002.

F. Stulp, M. Beetz. Optimized execution of action chains using learned performance models of abstract actions. *IJCAI*, 2005.

Plan Execution and Coordination

Pedro Szekely
Robert Neches
University of Southern California

Marcel Becker
Stephen Fitzpatrick
Kestrel Institute

Chris van Buskirk
Doug Fisher
Gabor Karsai
Vanderbilt University

Abstract

We investigate the problem of keeping the plans of multiple agents synchronized during execution. We assume that agents only have a partial view of the overall plan. They know the tasks they must perform, and know the tasks of other agents with whom they have direct dependencies. Initially, agents are given a schedule of tasks to perform together with a collection of contingency plans that they can engage during execution in case execution deviates from the plan. During execution, agents monitor the status of their tasks, adjusting their local schedules as necessary and informing dependent agents about the changes. When agents determine that their schedule is broken or that a contingency schedule may be better, they engage in coordinating plan changes with other agents. We present a "dynamic partial centralization" approach to coordination. When a unit detects a problem (task delay, inability to perform a task), it will dynamically form a cluster of the critically affected agents (a subset of all potentially affected agents). The cluster will elect a leader, who will retrieve all task and contingency plan information from the cluster members and compute a solution depending on the situation.

Introduction

Plan execution often involves a collection of agents, such that each agent gets a copy of its own plan, but does not know the overall plan, for all the agents. This is a very common situation in the military, where fielded human units do not have access to the full plan. Equally often, plan execution fails and there is a need for repairing the plans of the individual agents such that the overall goals of the agent ensemble are achieved. In the case of human agents this repair is facilitated by a coordination process that includes rapid communication and ad-hoc adaptation of plans by humans. In our work, we are looking at computational approaches that tie monitored plan execution to rapid plan repair and coordination among autonomous executor/planning agents.

We assume that an active component: a coordinator agent, is monitoring the execution of its plan. It receives notifications from its external world about events that could indicate the success or failure of plan execution. When the execution of its plan fails at a specific point in time, it goes into a coordination mode, where it computes changes to its plan such that the overall goals of the plan are achieved. The problem is that the plans of the individual coordinators were created dependencies among them, and failure in one plan step may impact the agent but also other, dependent agent's plans as well. Our interest is to develop distributed algorithms that facilitate rapid coordination: plan repair across a multitude of related and dependent coordinator agents.

Our approach to coordination is "dynamic partial centralization". When a unit detects a problem (task delay, inability to perform a task), it will initiate a process that will dynamically form a cluster of the critically affected units (a subset of all potentially affected units). We believe this cluster formation is crucial for improved performance. The problem of coordination can be solved in a fully centralized or in a fully distributed way. In the first case, a central server is needed (organizationally unacceptable for our domain), in the second case a distributed constraint solving process can be used (which has performance problems). The cluster solution combines the advantages of the centralized approach (better performance) without the problems of the fully distributed solution (using too much communication).

Modeling

Our work utilizes the TAEMS modeling framework (Decker and Lesser 1993). TAEMS models plans hierarchically: the leaves of the tree are called *methods*, and represent activities that agents can perform directly. The internal nodes are called *tasks*, and represent procedures to combine multiple activities (tasks or methods) to achieve higher level goals.

The TAEMS formalism offers several features appropriate for modeling the dynamics of a real world execution environments. We present the main concepts here (formal specifications are described in (Decker and Lesser 1993), details of the TAEMS framework are described in (Lesser *et al.* 2004)).

1. *Quality*: methods define a probability distribution for the quality that results when a method is executed.
2. *Duration*: methods define a probability distribution for the amount of time it takes to perform a method.
3. *Cost*: methods define a probability distribution of the cost that will be incurred when executing a method.

Each task defines a *quality accumulation function* (qaf) that specifies how the quality of the task is computed based on the quality of the children. TAEMS offers a collection of about 20 qafs. The most relevant to our work are:

- *Min*: the quality of a task is the minimum of the quality of the children. Min corresponds to the traditional And logical operator given that unattempted or failed tasks receive quality zero.
- *Max*: the quality of a task is the maximum of the quality of the children. Max corresponds to the traditional Or logical operator given that the quality of the task will correspond to the best quality of any attempted subtask.
- Sum, Xor, Sequence, etc. enable modeling of other situations that arise in real world applications.

TAEMS also provides a capability to define interrelationships among tasks:

- *Enables, Disables*: these are hard constraints among tasks or methods. If A enables B, then attempting to perform B before A completes will result in B failing and accumulating zero quality. Disables is defined similarly.
- *Facilitates, Hinders*: these are soft constraints. If A facilitates B, then when B is executed, its quality will be multiplied by a facilitation factor that depends both on a power factor defined in the Facilitates relation and the percentage of maximum quality that A obtained. Hinders is defined similarly.

The TAEMS model of agents is very simple. Methods and Tasks can be associated with a collection of agents that can perform it. During execution, methods can only be performed by a single agent, so part of the planning/coordination process involves selecting a single agent from the collection of possible agents.

TAEMS distinguishes between subjective and objective views of the world. The subjective view of the world is a TAEMS structure that defines the portion of plans that an agent knows about. Initially, agents are given a TAEMS structure containing the tasks and methods where the agent is listed, as well as all tasks an methods of other agents directly linked to the agent's tasks and methods (via Enables, Disables, Facilitates, Hinders and Parent relationships). During execution agents may communicate their subjective structures to other agents. The objective view is a conceptual entity containing all TAEMS structures of all agents. It may not be known to any agent, but for experimentation, the objective view is known to a simulator.

Plan Execution

The main objective of our work is to build agents that reason in real-time about the outcomes of method execution (quality, duration and cost) and adjust their plans in order to optimize the quality of the root level goal of the plan while staying within cost deadline and cost constraints.

Our work does not assume that execution follows the follows the subjective models faithfully. Our system is reactive and will always attempt to optimize the final outcome with respect to the current state, irrespective of how the current state came to be.

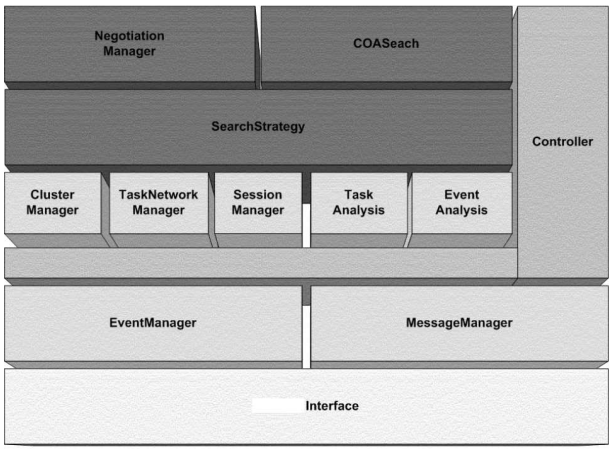


Figure 1: Architecture

Assumptions

The main assumption of our work is that the agents model human activities, whose duration is typically measured in minutes. This is in contrast to sensor network domains where time frames are in the order of seconds or less (e.g., react to an incoming missile).

The human activity assumption means that response times for adjusting plans can often be in the order of tens of seconds. For example, if I miss my flight at the airport, it is OK to wait 10 or 30 seconds for a new plan that directs me to take an alternative flight, redirects me to a new city, etc.

Architecture

Figure 1 shows the main components of an individual agent. The architecture builds upon an Interface that links the agent to the external world. The agent receives events in the Event Manager and sends and receives messages via the Message Manager. Events inform the agent about the success or failure of the execution of plan steps, while messages are used to communicate (and coordinate) with other agents. These managers convert events and messages into the internal representations of our system and give control to the Controller for further processing.

Events trigger Event Analysis to determine if there is a potential impact of the event on the plan of the agent. For example, the event may indicate that a method that a task depends on has not finished on time as expected. In such cases Event Analysis produces a task impact report and gives control back to the Controller.

Task impact reports are processed in the Task Analysis component. This component uses a low cost distributed constraint propagation algorithm to repair the schedule when execution deviates from the plan scheduled. When a task or method slips, this algorithm will tighten the start window for dependent tasks and methods. When the window of a task or method owned by a different agent is tightened, then a message is sent to the other agent to force it to tighten its representation of the time window. The effects on time windows are propagated as necessary. If the schedule has

enough slack to absorb the delay, then the propagation will die out. If a start window becomes empty (i.e., a task or method should finish before it starts), then the impact of the triggering event cannot be absorbed by simply adjusting the start time of the methods in the current schedule. At this point the impacted agent must engage a more sophisticated search algorithm that will change the methods for achieving the current goals. This search is started by the producing a clustering report and returning control to the Controller.

The clustering reports are first processed in the Cluster Manager, which initiates the cluster formation processes, by defining a new cluster that initially contains just the task whose start window is empty. A search report is produced and control returns to the Controller.

The Search Strategy initiates the cluster-based search algorithm by first expanding the cluster to contain the neighbors of the seed element of the cluster, and then selects a COA Search (Course of Action Search) algorithm to try to find a feasible schedule within the elements of the cluster, but without changing the time windows or dependencies for any of the tasks or methods in the boundary of the cluster. If such a solution exists it a search report is produced containing the new solution. If no solution is found or a given time threshold is exceeded, then a failure search report is produced requesting cluster expansion.

The Cluster Manager expands clusters following dependency links on the tasks and methods already in the cluster and sending messages to their agents to query the corresponding TAEMS structures. The Cluster Manager ensures that no task or method belongs to more than one cluster.

When the request to grow a cluster fails (clusters collide) the Search Strategy decides whether to merge clusters, nominating one of the leaders as the leaders of the new merged cluster, or whether to stop centralization and engage in distributed constraint satisfaction between the leaders of the colliding clusters. These interactions are governed by the Negotiation Manager that implements a negotiation protocol among cluster leaders.

When the COA Search algorithms stop, producing a new schedule (even if it is only a partial schedule), the leader will distribute the new COA to all the cluster members, who will incorporate it into their subjective view. The receiving agents will apply all events that have arrived since the cluster-based search process started.

Given the human activity assumption, it is expected that in the majority of cases cluster sizes can be kept small enabling the use of fast, centralized solutions techniques that enable leaders to quickly compute high quality solutions before the world changes in a significant way. As a last resort, if merging of clusters would result in large clusters, leaders will negotiate using slower, less optimal distributed constraint satisfaction techniques.

Depending on circumstances, human users may need to approve these plans. For this reason, plan choices are presented in a ranked order of utilities to a human, who can then approve the chosen plan.

Conclusions and Status

The approach presented here represents an interesting compromise between approaches based on traditional planning representations (e.g., PDDL 2.2) and an approaches based on a task-oriented representation such as TAEMS. TAEMS is not a planning language: preconditions and effects of methods are not defined explicitly . The effects are encoded implicitly in the inter-relationships among tasks and methods. TAEMS is expressive enough to encode contingency plans, i.e., alternative ways to accomplish goals. Each of the contingency plans is a fragment of a larger plan and they must be combined into a consistent plan. However, TAEMS provides a good model of task and method execution.

The partial centralization approach for distributed coordination has been explored before (Scerri *et al.* 2004). Our approach is more closely related to Mailler's work on cooperative mediation of distributed constraint optimization (Mailler and Lesser 2004).

This paper presents the approach for a new system. We have designed the system architecture, have high level ideas for the search algorithms involving centralized search over TAEMS structures. We plan to demonstrate the system created on small scale, abstract examples by the end of the current year (Nov 2005).

Acknowledgments

The work presented here is funded by the DARPA COORDINATORS Program under contract FA8750-05-C-0032. The U.S.Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, 1993.

V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X.Q. Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.

Roger Mailler and Victor Lesser. Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 438–445. IEEE Computer Society, 2004.

Paul Scerri, Regis Vincent, and Roger Mailler. Comparing Three Approaches to Large Scale Coordination. *Proceedings of the First Workshop on the Challenges in the Coordination of Large Scale Multi-agent Systems*, July 2004.

Survey of Command Execution Systems for NASA Spacecraft and Robots

Vandi Verma, Ari Jónsson, Reid Simmons, Tara Estlin, Rich Levinson

QSS / NASA Ames Research Center
USRA-RIACS / NASA Ames Research Center
Carnegie Mellon University
Jet Propulsion Lab
Attention Control Systems Inc.

MS 269-1 NASA Ames, Moffett Field CA 94035
MS 269-2 NASA Ames, Moffett Field CA 94035
3205 Newell-Simon Hall, Carnegie Mellon University, Pittsburgh, PA 15213
M/S 126-347, JPL, Pasadena CA 91109-8099
650 Castro St., Ste 120 PMB 197, Mountain View CA 94041

vandi@email.arc.nasa.gov, ajonsson@arc.nasa.gov, reids@cs.cmu.edu, tara.estlin@jpl.nasa.gov, rich@brainaid.com

Abstract

NASA spacecraft and robots operate at long distances from Earth. Command sequences generated manually, or by automated planners on Earth, must eventually be executed autonomously on-board the spacecraft or robot. Software systems that execute commands on-board are known variously as *execution systems*, *virtual machines*, or *sequence engines*. Every robotic system requires some sort of execution system, but the level of autonomy and type of control they are designed for varies greatly. This paper presents a survey of execution systems with a focus on systems relevant to NASA missions.

Introduction

As NASA's missions become more complex, autonomy becomes increasingly important to the success of those missions. At the heart of such autonomous systems are sub-systems, known variously as *execution systems*, *virtual machines*, or *sequence engines*, that execute commands and monitor the environment. Such execution systems vary in sophistication, from those that execute linear sequences of commands at fixed times, to those that can plan and schedule in reaction to unexpected changes in the environment.

For NASA missions, sophisticated execution systems are highly desirable for several reasons. One is that NASA spacecraft and robots typically operate far from Earth, and so must take on significant responsibility for their own health and safety. Second, models of robot sensors and effectors are often uncertain and the environment is generally only partially known and may even be dynamic. To account for uncertainty even a simple deterministic sequence of commands needs to use worst-case estimates of action duration and resource use. In some cases even this suboptimal sequence may not be robust to the uncertainty.

This paper presents a survey of execution systems that have been developed for various applications. We focus on those systems that are relevant to NASA-type applications. Before presenting the individual systems, we define some terms that are critical to understanding execution systems.

An *executive* is a software component that realizes pre-planned actions. Executives are particularly useful in the presence of uncertainty. Classical executive functions include selecting an action from a set of possibilities based on the current state of the robot and environment and outcome of previous actions, hierarchical task decomposition, coordinating simultaneous actions, resource management, monitoring of states, resources command status, and fault diagnosis and recovery. One way to view an executive is as an onboard system that takes a plan that assumes a certain level of certainty and expected outcomes and executes it in an unknown and possibly dynamic environment.

A *plan* is a series of actions designed to accomplish a set of goals but not violate any resource limitations, temporal or state constraints, or other spacecraft or rover operation rules. Desirable characteristics of a plan are that it be valid, complete and optimal (or of high quality). Algorithms that can reason about achieving goals over a future time period and in the face of various constraints are called *planners*.

However, a plan, as generated by most any current planner, still requires the help of an execution system to be useful for real-world execution. Making these plans executable may not involve complex AI algorithms, but is essential for achieving the plan. In order to perform plan execution, control structures such as conditional statements that catch violated assumptions, looping constructs that can retry an action until it succeeds, and more detailed descriptions of preconditions that must be checked before an action is executed must be added. Execution languages provide constructs to represent essential plan execution information in addition to the plan.

An *execution language* is a representation of actions and plans that takes into account the state of robot and environment at the time the action is executed, and the interdependence between actions, in terms of temporal, precedence, or other constraints.

Model-based systems are represented by a knowledge-base (model) of its structure and behavior and are typically specified using a declarative representation. In other words, these models do not specify the sequence of actions required to fulfill specific high-level goals of the system, but instead they specify the expected effect each action or external event may have on the modeled state. Models are often specified in a modular manner, where only the local effect of an event is described. Planners may use these models to find sequences of actions directed toward the goal or a fault diagnosis system may use them to detect and identify faults.

Some execution systems use no automated planners; we call these *execution-only* systems. Other execution systems have explicit interfaces to planners, (through an execution language or a standard format like XML), we call these *execution systems coupled with an external planner*. Yet another class of execution systems integrate planning and execution more tightly by using a planner internally within the execution system to select control actions. We call these *execution systems with internal planner*. Note this is not a mutually exclusive classification. Some execution system may be used as an execution only system with manually encoded plan execution, but may also have well defined interfaces to one or more automated external planners that may be used in other applications. An execution system may also provide an external interface to a planner in addition to having an internal planner. Finally, note that the coupling of execution systems with external planners can differ in tightness, ranging from infrequent requests for assistance to continuous information sharing. In cases where the coupling is tight, the combined functionality is similar to integrated execution-planning systems.

Traditional Command Execution

At this time, most spacecraft and rovers are operated via sequences of commands. The command sequences are fairly simple in structure and the interpretation on board the spacecraft is straightforward. Dynamic outcomes and environmental uncertainties are handled partially by making sequences conformant to possible outcomes, and partially by relying on on-board fault detection and system health software.

In this context of traditional spacecraft operations, the executive is the flight software system on board the spacecraft; more specifically, the sequence execution system and the health monitoring and fault detection system. The execution language is simple; an execution plan is a fairly small set of branching command sequences and sub-sequences, where each command is either executed at a specific time, or immediately following the completion of another command. Typically, there are no conditionals, no loops, no constraints, etc.

The most notable properties of this approach are:

- Plans become inherently conservative, so as to be conformant to expected outcomes. For example, activities are assumed to take the longest they can possibly take.
- The on-board health monitoring system is limited to general responses to failures, which often leads to unnecessary execution aborts and spacecraft operations halts. For example, a certain failure might lead to abandoning the whole plan, whereas portions of the plan could still be safely continued.

Virtual Machine Language (VML) [14] is a sequencing language that has flown on numerous NASA spacecraft. VML is currently in use on the Spitzer space telescope, Mars Odyssey, Mars Reconnaissance Orbiter, Dawn, Genesis, and Stardust. It is slated for future New Frontier and Discovery class missions, including the Mars Telecom Orbiter and possibly the Mars Science Lander.

VML is an execution language that was developed to take into account the needs of spacecraft operations. It provides a "safe-sandbox", with the aim of shielding operations personnel from most of the mistakes possible in contemporary programming languages like C. Sequences are procedural, and have symbolic names. At any time only one instruction is active in a sequence engine (also known as a virtual machine). The language accommodates a variety of spacecraft commanding architectures. It features absolute and relative constraints, event-driven sequencing, programmable delays, arithmetic and bit-level operations, parameters with polymorphism, and a number of numeric and string data types. VML dynamically builds spacecraft commands with values derived from variables, and has reusable blocks that can be called or spawned from sequences. The on-board sequencing component can also

be configured to access telemetry values for use within sequences.

The VML language is compiled to an uplinkable file form in a Unix-based ground system by the VML Compiler. This process translates human-readable text into a binary file for interpretation onboard by the VML Flight Component. The flight component is implemented in C for compatibility with the widest possible range of missions. In addition, a Unix tool known as Offline VM (OLVM) is available for ground-based execution and debugging of developed blocks and sequences. OLVM encapsulates the actual flight code for high fidelity testing with very fast turnaround when developing using VML.

Execution Systems

This section presents several NASA-relevant execution systems, in alphabetical order.

Apex

Apex [10] is an execution system and has been used in numerous large-scale applications including control of real autonomous helicopters, control of simulated aircraft for wildfire detection, and in simulating humans for Human Computer Interface (HCI) analysis.

Apex is a reactive execution system that selects for execution one or more procedures (partial plans) from its library of procedures at each execution step. In most applications Apex has been used as an *execution-only* system. Apex is designed to unify plan-running and mission-management functionality. Planners may be called on to produce or extend a mission plan, to solve a local planning problem within a mission plan or both. Apex may therefore potentially be used as an *execution system coupled with an external planner*.

The execution language used by Apex is the Procedure Description Language (PDL). PDL can represent a hierarchical decomposition of a high-level task into basic primitives, event driven floating contingencies, and also calls to Lisp (the underlying programming language). A PDL procedure consists of a unique identifier, a description of a class of goals the procedure applies to, and one or more step clauses. The step clauses are concurrently executable and may call other procedures (sub-tasks).

The input to Apex is a set of human-fabricated procedures represented in PDL. Apex is a reactive system that chooses an action at every execution step. Key capabilities of the executive (and of PDL) are:

- Monitoring/querying for complex temporal events patterns
- Opportunistic (reactive) task refinement and resource allocation
- Management of concurrent and periodic tasks

Continuous reaction allows Apex to use the most recent measurements to guide the selection of the next action. In addition it allows dynamic update of high level goals. Apex also provides a number of tools for debugging, demonstration, and monitoring.

CRL and C-CRL Executive

The *Contingent Rover Language* (CRL) [4] is a declarative plan execution language that was designed to represent contingent plans. It uses a hierarchical representation and can represent simple and floating branches, nesting, flexible time, and state and resource conditions. The CRL executive has been used on NASA's Marsokhod, ATRV, and K9 rovers as a high-level plan interpreter. It has also been used with the Mission Simulation Facility (MSF) rover simulator. C-CRL is an extension of CRL that is capable of concurrent execution and has been used for the single-cycle instrument placement demonstration on the K9 rover [21].

The CRL executive may be used as an *execution-only* system with manually written CRL constructs. The external planner that generates CRL plans is the PICO contingent planner [5]. CRL does not support loops and periodic tasks, or have a mechanism for providing feedback to planners.

IDEA

Intelligent Distributed Execution Architecture (IDEA) [20] is a model-based planning and execution system. One of the two glitches experienced by Remote Agent was due to undocumented and subtle differences in semantics between models in the planning, execution and diagnosis layers. IDEA was developed to address this problem by building an architecture that supports controllers/planners at multiple levels of abstraction. Controllers (agents) at every level of abstraction share the same model. The semantics of the structure of a task, the structure of an execution cycle responsible to activate a task in response to an asynchronous or synchronous event, the structure of events communicated between controllers, how the communication of tasks maps into the transport layers responsible of delivering them across agents, are thus uniform. Each controller (control agent) at every level of abstraction is assumed to perform planning as the sole computational process to decide how to respond to events.

IDEA uses the classic sense-plan-act cycle. One of the novel features in IDEA is the use of an on-board planner from first principles (i.e., the sub-goaling model) to plan for a limited horizon into the future and execute the current task at hand simultaneously. The advantage is that this allows it to dynamically update the plan based on the

current state of the world and previous actions, which can yield a wider range of robust behaviors than possible with traditional execution scripts. The disadvantage of using planning from first principle at every execution cycle is that patterns of constraints (temporal and parametric) are always assembled from scratch, causing higher latency than possible when using pre-compiled execution scripts. Consequently, execution may halt if the planner can't deliver a response in time. IDEA agents can also use an arbitrary number of deliberative planners to optimize agent behavior over a long, future horizon. IDEA is thus an *execution system with internal planner* (reactive planners) and may also be used as an *execution system coupled with an external planner* (deliberative planners) at the same time.

XIDDL is the execution language used in IDEA. It is a modeling language amenable to temporal/hybrid planning through subgoaling, used to describe the model of the world, the internal logic and the input/output behavior of each IDEA controller. This uniformity aims to facilitate system-level validation for an autonomy system without the need for understanding the details of each specific controller, since it is expensive and error prone to assume that mission personnel will examine software written in different computer languages in order to ascertain its ability to satisfy mission requirements. IDEA has been used for autonomously controlling a telescope, PSA (personal satellite assistant), and a number of mobile robots.

While IDEA is designed to use any planner that uses a representation that is compatible with the XIDDL modeling language, all of the IDEA systems developed so far use the Europa planning technology [9] both for reactive and deliberative planning.

MPE

Mission Planning and Execution (MPE) [1] is the execution subsystem of the Mission Data System (MDS) [24]. MDS uses an explicit state-based representation. Knowledge about the spacecraft and the environment is provided by state estimates. Knowledge about the behavior of the system is stored in state models. Information is reported via a history of states, measurements, and control commands. The input to MPE is operator "intent" (expressed as temporal constraints, and constraints on states), flight rules, and hard constraints on variables. MPE is an *execution system with internal planner* that can locally adapt the original plan to recover from faults and handle uncertainty.

PROPEL

Program Planning and Execution Language (PROPEL) [17] [18], is a unified planning and execution system that uses a procedural representation. This is different from IDEA, which exclusively uses a declarative action representation.

The motivation was that since most software is not written as a declarative model it tends to be outside the scope of a planner's reasoning. PROPEL was designed to increase the scope of the planner's model to include software in order to address the problem of software failure detection and recovery.

Propel was designed to close the gap between the declarative action model used by a planner and the procedural languages used to develop real-world software. The representation is intended to be expressive enough to be used in system software including the planner and executive software. Motivation for using a procedural representation includes:

- Desire to include all software within the planner's model in order to increase the scope of failure recovery to include infrastructure software failures.
- Desire to represent complex procedures including loops, conditionals, local variables, and multiprocessing.
- Desire to reduce the need to develop and maintain different models for the planner and execution system.
- Reduce risk of loss of information in translation between execution and planning (and vice versa).

Propel is both an architecture and a language. The architecture provides integrated planning and execution modules that monitor and manipulate application-level processes written in the Propel language. The language is a library of methods for embedding search and temporal constraint information into C++, thus creating a "superset" of C++ like TDL. This library provides an interface from the Propel application code to the supervisory meta-processes (the planning and execution modules), which monitor the application to provide failure detection and recovery.

The language provides an action representation that captures control constructs and can also be projected by a search-based planner. The planner can provide a useful partial plan even when it is interrupted after an arbitrary amount of computation. The planner and the controller share identical data structures and algorithms for interpreting a shared representation of control actions. PROPEL is an *execution system with internal planner*.

PRS

The *Procedural Reasoning System* (PRS) [12] was developed to address the problem encountered in developing autonomous systems that were required to be continuously active and have real-time response.

Traditional programming languages imposed an order on task execution through the language's control structure that makes it difficult to respond quickly to a large set of possible events.

PRS is a reactive goal-driven system that selects procedures (partial plans) from its library of procedures at each execution step. PRS is an *execution-only* system.

PRS has a knowledge-base of procedures. Each procedure requires the specification of an event, the state of the world that will trigger that event, the steps that are executed by the procedure, and the sub-goals that it achieves.

PRS has been used on a number of mobile robots and also in a simulation of the space shuttle. PRS was originally written in Lisp and is now known as PRS-CL. The C version of it is called C-PRS or *Propice* [15].

RAP System

Reactive Action Package System (RAPs) [8] was designed to support reactive planning and execution. It is a representation language for general-purpose execution. It uses a Lisp-based interpreter to manage a task network and to interface to a behavioral layer. RAPs may thus be used as an *execution-only* system or *execution system coupled with an external planner*.

The main idea behind RAPs is that all capabilities of goal-achieving behaviors – task decomposition, different tactics for achieving a goal, monitoring, error recovery, checking of pre- and post-conditions – should be represented in a single "package." Each RAPs is thus a self-contained module that knows how to achieve a particular goal in the face of uncertainty.

The RAPs execution system uses a library of goal-achieving behaviors and a symbolic world database to choose which RAPs to execute, how to decompose them, and when they succeed or fail. The execution system schedules RAPs according to their priority and temporal constraints, interrupting execution of one RAP if higher priority RAPs become active.

Remote Agent (RA) Executive

Remote Agent [22] is an AI system that flew on-board the Deep Space One (DS-1) spacecraft in 1999. The main characteristics of the Remote Agent are that it is model-based with on-board planning, fault detection, identification, and recovery.

The executive in the Remote agent [23] is the central controller. The input to the Remote Agent executive is a high-level state and duration for which the state must be maintained. The executive autonomously calls the planner to generate a plan to satisfy a high-level goal. It uses a domain model to monitor plan execution and commands the planner to generate an updated plan if any of the constraints are violated during execution.

The Remote Agent executive was based on the *Execution Support Language (ESL)* [11]. ESL is a declarative execution language that is an extension of Lisp. It is implemented as a set of macros that expand into Common Lisp and invoke Lisp's multi-tasking library. ESL provides task-level control constructs, resource management, and a database built on Prolog

The novel features demonstrated by the RA executive in the DS-I experiment were integrated planning and execution with low-latency response time to contingencies and deficiencies in the plan and the lack of intervention required by the human operator after issuing high level mission goals. The RA executive is an *execution system with internal planner* and also an *execution system coupled with an external planner* at the same time.

One of the main challenges with this approach is building and maintaining models. The emergent behavior that results from subtle interactions between qualitative models of weakly interacting subsystems is hard to predict since the range of input conditions and responses are extremely large. "Incorrect knowledge in the domain model could endanger or even lose the mission" [2].

RMPL, Titan, Kirk, Moriarty

Titan [29] is a model-based execution system that supports both execution control and model-based goal achievement specifications. The execution control component generates goal states, which are then given to the model-based goal achievement component. The goal achievement component uses automated diagnosis methods to estimate the current state from observable data (*mode identification*), and then uses automated planning (*mode reconfiguration*) to generate command sequences to achieve the given goals from the current state.

The execution language used in Titan is the *Reactive Model-based Programming Language* (RMPL) [28]. It is used to specify both the control information used by the control component and the model-based state estimation and planning component. The control information supports control constructs such as loops, conditions, iterations and contingencies, over model-based specifications of concurrent and sequenced goals. The control elements of RMPL are compiled into hybrid control automata (HCA), while the mode identification and reconfiguration is specified in terms of concurrent control automata (CCA).

Titan differs from Propel because it compiles procedural constructs into a declarative model, which is then interpreted by during execution. Titan is similar to IDEA

this way, but differs from IDEA by using an explicit description of control behavior.

The core Titan system and the RMPL language have been extended to handle hybrid (continuous/discrete) state information, resulting in a system called Moriarty. A different extension, implemented in the Kirk execution system [30], supports distributed cooperative execution. Titan, Moriarty and Kirk may be described as *execution systems with internal planner*.

RPL

Reactive Plan Language (RPL) [19] was inspired by RAP and PRS. It is a Lisp-like language and includes rich set of control constructs, such as conditionals, looping, and the ability to specify "policies" that hold during the execution of particular sub-tasks.

RPL was designed to support replanning and debugging of task definitions [2]. Based on experience obtained during execution and Monte-Carlo simulations of task execution, situations can be identified where tasks are likely to fail. Heuristic "critics" are then used modify the task (e.g., adding new constraints, adding new policies) in order to fix the bugs found. RPL is an *execution system with internal planner*.

TDL

The Task Description Language (TDL) [25] uses a procedural representation to support plan execution. It is an extension of C++, adding syntax for specifying high-level control. A Java-based compiler translates TDL into pure C++, together with calls to a domain-independent task management library. The resulting code can then be compiled with any existing compiler and linked with existing C++ code. There are options in the language to specify that the resulting code should be threaded and/or distributed (the latter used for coordinating multiple robots.

TDL provides the ability to represent high-level control constructs including task decomposition, task coordination and synchronization, execution monitoring and exception handling, as well as distributed coordination between multiple agents. Being an extension of C++ makes it very easy to integrate TDL into projects – developers can use as much, or as little, of the TDL functionality as they need to augment the standard C++ functionality.

High level control constructs are represented in TDL as task-trees. Task trees represent the execution trace of hierarchical plans and are created dynamically at run time. The task-tree decomposition can be created from conditional and recursive task representations. The temporal constraints in the task-tree decomposition (partially) order task execution. Planning and sensing are treated as schedulable activities. In other words, the executive runs the main loop and calls the planner when required. TDL is an *execution system coupled with an external planner.* In several projects, a symbolic *Plan Representation Language (PRL)* was used to transfer data between a planner and a TDL-based executive [13] To date, TDL has been used in about a dozen mobile robot and autonomous system projects at various universities and institutions, including several NASA rovers [7].

Universal-Executive

The Universal-Executive is currently under development in a collaborative effort of researchers at NASA Ames Research Center, NASA's Jet Propulsion Laboratory and Carnegie-Mellon University. It is being designed to facilitate reuse and inter-operability of execution and planning frameworks. Plan execution systems often have a close relation to the planners that they are associated with, which makes information sharing between different execution and decision-making systems difficult.

The Universal-Executive builds on the Coupled Layer Architecture for Robotic Autonomy (CLARAty) [27], which is a two layer software architecture that was developed to enable both a plug-and-play capability and a tighter coupling of high level decision making planners and the interface to hardware. The CLARAty architecture has successfully enabled interoperability at the Functional Layer, which is the interface to the hardware. Current work, including the development of the Universal Executive, is addressing this same goal at the Decision Layer.

The execution language to be used in the Universal-Executive is called *Plan Execution Interchange Language (PLEXIL)*. PLEXIL extends many execution control capabilities of other systems. The key characteristics of PLEXIL are that it is compact, semantically clear, and deterministic given the same sequence of events. At the same time, the language is quite expressive and can represent simple branches, floating branches, loops, time and event driven activities, concurrent activities, sequences, and temporal constraints.

The input to the Universal-Executive will be a PLEXIL representation of an execution control instance and a description of relevant domain information. Execution nodes describe both initiation of real-world actions, and the control of execution. The nodes are arranged into hierarchical trees where leaf nodes are action nodes and internal nodes are control nodes. This is different from TDL, where task trees are a type rather than an instance.

The execution of each node is governed by a set of conditions, such as when the node gets activated and when it is done. The Universal-Executive will be capable of executing multiple nodes concurrently. When action nodes

are executed, commands are sent to the rover, whereas when internal nodes are executed, they are expanded to the next level of nodes in the tree.

The expressiveness of the language enables the Universal Executive to handle dynamic outcomes and environmental uncertainty. The executive can also provide execution information and outcomes back to higher-level systems. Consequently, it can be used both as a stand-alone *execution-only system*, and as an *execution system coupled with an external planner*.

Conclusions

The demands of future NASA spacecraft and robotic missions dictate an execution system that has great flexibility, expressiveness, and ease of use. This paper has presented a number of execution systems and execution languages that are relevant to NASA-type missions.

Acknowledgements: We thank Emmanuel Benazera, Howard Cannon, Mike Freed, Nicola Muscettola, Corina Pasareanu, and Rich Volpe for many useful discussions and all the attendees of the "Workshop on Existing Planning and Execution Systems", Nov 16, 2004 at NASA Ames Research Center, which motivated this paper. In addition, Mike Freed, Chris Grasso, and Nicola Muscettola provided invaluable comments on this paper.

References

1. Barrett A., Knight R., Morris R., Rasmussen R., *Mission Planning and Execution Within the Mission Data System*, International Workshop on Planning and Scheduling for Space (IWPSS 2004). Darmstadt, Germany, June 2004.
2. Beetz M. and McDermott D., *Declarative goals in reactive plans,* In James Hendler (ed.), *Proc. First Int. Conf. on AI Planning Systems*, San Mateo: Morgan Kaufmann, pp.~3--12
3. Bernard D. et al. *Final Report on the Remote Agent Experiment*, NMP DS-1 Technology Validation Symposium Feb 8th and 9th 2000, Pasadena, CA
4. Bresina J.L. and Washington, R., *Robustness via Run-time Adaptation of Contingent Plans*, In Proceedings of the AAAI-2001 Spring Syposium: Robust Autonomy. Stanford, CA
5. Dearden R., Meuleau N., Ramakrishnan S., Smith D., and Washington R., *Incremental Contingency Planning, ICAPS-03 Workshop on Planning under Uncertainty,* Trento, Italy, June 2003.
6. Drummond M., Bresina J., Kedar S., The Entropy Reduction Engine: Integrating Planning, Scheduling, and Control, in SIGART Bulletin 2, 1991, 48-52
7. Estlin T., Gaines D., Chouinard C., Fisher F., Castano R., Judd M., Anderson R., and Nesnas I.. "Enabling Autonomous Rover Science Through Dynamic Planning and Scheduling," *Proceedings of 2005 IEEE Aerospace Conference*, Big Sky, Montana, March, 2005.
8. Firby J, *Adaptive Execution in Complex Dynamic Domains*, Ph.D. Thesis, Yale University Technical Report YALEU/CSD/RR #672 *January 1989*
9. Frank J., and Jonsson A. K., *Constraint-based Attribute and Interval Planning*, in Constraints, 8(4), p 339-364, 2003.
10. Freed M., *Managing Multiple Tasks in Complex, Dynamic Environments.* In Proceedings of the 1998 National Conference on Artificial Intelligence. Madison, WI. 1998
11. Gat E.. *ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents*, Proc. AAAI Fall Symposium on Plan Execution, Boston MA, October 1996.
12. Georgeff M. and Lansky A., *Procedural Knowledge*, in Proceedings of the IEEE Special Issue on Knowledge Representation, Volume 74, pages 1383-1398, 1986.
13. Goldberg D., Cicirello V., Dias M. B., Simmons R., Smith S., and Stentz A., *Market-Based Multi-Robot Planning in a Distributed Layered Architecture,* In Proceedings of the Multi-Robot Systems Workshop, Washington, D.C., March 17-19, 2003
14. Grasso C., *The Fully Programmable Spacecraft: Procedural Sequencing for JPL Deep Space Missions Using VML (Virtual Machine Language)*, IEEE Aerospace Applications Conference Proceedings, march 2002
15. Ingrand F., R. Chatila, R. Alami and F. Robert, *PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots*, IEEE ICRA 96, Minneapolis, USA.
16. Kim P., Williams B., Abramson M., 2001. *Executing Reactive, Model-based Programs through Graph-based Temporal Planning*. IJCAI '01. AAAI, Menlo Park, CA.
17. Levinson R., *A General Programming Language for Unified Planning and Control*. Artificial Intelligence, special issue on Planning and Scheduling, Vol. 76. Elsevier Press. July 1995.
18. Levinson R., Unified Planning and Execution for Autonomous Software Repair. ICAPS 2005. Workshop on Plan Execution: A Reality Check. 2005.
19. McDermott D., *A Reactive Plan Language,* Research Report YALEU/DCS/RR864 Yale University 1991
20. Muscettola N., Dorais G., Fry C., Levinson R., and Plaunt C., "IDEA: Planning at the Core of Autonomous Agents," (AAAI 2001)
21. Pedersen L., Smith D., Deans M., Sargent R., Kunz C., Lees D. and Rajagopalan S., *Mission planning and target tracking for autonomous instrument placement*, 2005 IEEE Aerospace Conference.
22. Pell B., Bernard D.E., Chien S. A., Gat E., Muscettola N., Nayak P. P., Wagner M. D., and Williams B. C.. *A Remote Agent Prototype for Spacecraft Autonomy*. In Proceedings of

the SPIE Conference on Optical Science, Engineering, and Instrumentation, 1996.
23. Pell B., Gamble E., Gat E., Keesing R., Kurien J., Millar B., Nayak P. P., Plaunt C., and Williams B., *A Hybrid Procedural/Deductive Executive For Autonomous Spacecraft*. In Proceedings of the Second International Conference on Autonomous Agents, Minneapolis, MI 1998
24. Rasmussen R., *Goal-Based Tolerance for Space Systems Using the Mission Data System*, In Proceedings of the 2001 IEEE Aerospace Conference.
25. Simmons R. and Apfelbaum D.. A Task Description Language for Robot Control, Proceedings of Conference on Intelligent Robotics and Systems, Vancouver Canada, October 1998.
26. Simon D., Espiau B., Kapellos K., Pissard-Gibollet R. , *Orccad: Software Engineering for Real-time Robotics, A Technical Insight*, Robotica, Special issues on Languages and Software in Robotics, vol 15, no 1, pp 111-116
27. Volpe R., Nesnas I. A. D., Estlin T., Mutz D., Petras R., Das H., *The CLARAty Architecture for Robotic Autonomy*. Proceedings of the 2001 IEEE Aerospace Conference, Big Sky Montana, March 10-17 2001.
28. Williams B. C., Ingham M., Chung S. H., and Elliott P. H., January 2003. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers, invited paper in Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pp. 212-237.
29. Williams B. C., Ingham M., Chung S., Elliott P., and Hofbaur M., *Model-based Programming of Fault-Aware Systems*, AI Magazine, vol. 24, no. 4, Winter 2004, pp. 61-75.
30. Kim P., Williams B. C. and Abramson M, 2001, *Executing Reactive, Model-based Programs through Graph-based Temporal Planning*, Proceedings of the International Joint Conference on Artificial Intelligence, Seattle, Wa.