27th International Conference on
Automated Planning and Scheduling
June 19-23, 2017, Pittsburgh, USA

IntEx 2017

Proceedings of the Workshop on **Integrated Execution of Planning and Acting**

Edited by:

**Mark Roberts, Sara Bernardini,
Tim Niemueller, and Tiago Vaquero**

Organization

Mark Roberts, Naval Research Laboratory, USA

Sara Bernardini, Royal Holloway University, London, UK

Tim Niemueller, RWTH, Aachen, Germany

Tiago Vaquero, MIT, USA

Program Committee

Ron Alford, Mitre Corporation, USA

J. Benton, NASA Ames, USA

Mark Boddy, Adventium Labs, USA

Michael Cashmore, King's College London, UK

Jeremy Frank, NASA Ames, USA

Andy Hertle, University of Freiburg, Germany

Nir Lipovetzky, University of Melbourne, Australia

Julie Porteous, Teesside University, UK

Wheeler Ruml, University of New Hampshire, USA

Scott Sanner, University of Toronto, Canada

Vikas Shivashankar, Knexus Research, USA

Foreword

Automated planners are increasingly being integrated into online execution systems. The integration may, for example, embed a domain-independent temporal planner in a manufacturing system (e.g., the Xerox printer application) or autonomous vehicles. The integration may resemble something more like a ”planning stack” where an automated planner produces an activity or task plan that is further refined before being executed by a reactive controller (e.g., robotics). Or, the integration may be a domain-specific policy that maps states to actions (e.g., reinforcement learning). Online learning may or may not be involved, and may include adjusting or augmenting the model, determining when to repair versus replan, learning to switch policies, etc. A specific focus of these integrations involves online deliberation, bringing to the foreground concerns over how much computational effort planning should invest over time.

In any of these systems, a planner generates action sequences that are eventually dispatched to an executive, yet taking action in a dynamic world rarely proceeds according to plan. When planning assumptions are challenged during execution, it raises a number of interesting questions about how the system should respond. Is the ”acting” side of the system responsible for a response or the ”planning” side? Or do the two need to cooperate and how much? When should the activity planner abandon or preempt the current goals? Should the task planner repair a plan or replan from scratch? Should the executive adjust its current policy, switch to a new one, or learn a new policy from more relevant experience?

This set of working notes consolidates the papers presented at IntEx 2017.

Mark Roberts, Sara Bernardini, Tim Niemueller, and Tiago Vaquero
June 2017

Contents

k-Robust Multi-Agent Path Finding

Dor Atzmon, Roni Stern, Ariel Felner and **Roman Bartak** and **Neng-Fa Zhou**
Ben Gurion University of the Negev, Be'er Sheva, Isreal
Charles University in Prague, Prague, Czech Republic
City University of New York, New York City, NY, USA

Abstract

In the multi-agent path-finding (MAPF) problem a plan is needed to move a set of agents from their initial location to their goals without collisions. In this paper we introduce and study the k-robust MAPF problem, where we seek a plan that is robust to k unexpected delays per agent. We show how to convert a popular optimal MAPF solver – Conflict-Based Search (CBS) – to solve the k-robust MAPF problem. To handle cases where there are more than k unexpected delays, we analyze several execution policies that can complement using a k-robust plan. The proposed algorithms and execution policies are evaluated experimentally, and we discuss their pros and cons. In particular, finding a k-robust solution is shown to reduce the overall number of replans needed when executing a plan.

Introduction and Overview

The *Multi-Agent Path Finding* (MAPF) problem is defined by a graph, $G = (V, E)$ and a set of n agents labeled $a_1 \dots a_n$, where each agent a_i has a start position $s_i \in V$ and a goal position $g_i \in V$. At each time step an agent can either *move* to an adjacent location or *wait* in its current location. The task is to find a sequence of move/wait actions for each agent a_i that moves it from s_i to g_i such that agents do not *conflict*, i.e., occupy the same location at the same time. MAPF has practical applications in video games, traffic control, and robotics (see Sharon et al. 2013a; 2015a for a survey).

In many cases there is also a requirement to minimize some cumulative cost function such as the sum of costs incurred by all agents before reaching their goals. Solving MAPF optimally is NP-Hard (Yu and LaValle 2013b; Surynek 2010). Nonetheless, efficient optimal algorithms exist, some are even capable of finding optimal plans for more than a hundred agents (Wagner and Choset 2015; Boyarski et al. 2015; Surynek 2012; Yu and LaValle 2013a).

In practice, unexpected events may delay some of the agents, preventing them from following their predetermined plan. Thus, it is desirable to generate a plan that can withstand such events. Such *robust* MAPF plans are needed especially in safety-critical settings or when re-planning (due to unexpected events) is very costly or even impossible, for example due to lack of communication between the agents. Practical applications include air traffic control (where safety is critical) and multi-robot settings, where re-planning is often costly. To this end, we introduce the k-robust MAPF (kR-MAPF) problem, where we seek a plan that is robust to k delays per agent during plan execution. In a k-robust plan, if an agent occupies location l at time step t then no other agent may occupy location l at time steps $\{t \dots t+k\}$. Obviously, MAPF is a special case of kR-MAPF with $k = 0$. We present an adaptation of an A*-based MAPF solver to kR-MAPF, showing that the underlying state space grows exponentially with k. We then present and analyze several ways to implement a k-robust solver based on the Conflict-Based Search (CBS) optimal MAPF solver.

Since an agent may experience more than k delays, even a k-robust plan may need to be modified during execution. To this end we follow the framework of Ma et al. (2017) in which a plan is coupled with an *execution policy* to handle delays online, possibly modifying the original plan. Finding and using a k-robust plan integrates naturally in this framework. The result is a complete and robust solution that significantly reduces the number of times that modifications to the plan are needed during execution. Moreover, we introduce several novel execution policies with different tradeoffs between CPU time, number of required plan modifications, and total cost of the executed plan. Our kR-MAPF algorithms and new execution policies are evaluated empirically on standard MAPF benchmarks, showing that finding k-robust plans is feasible, that their solution costs are not much larger from the baseline (non k-robust) plans and that using them as input for any of the execution polices is beneficial.

Prior work addressed similar MAPF settings where agents move according to known probabilistic dynamics, e.g., getting delayed with some known probability (Wagner and Choset 2017). They proposed an algorithm based on M* (Wagner and Choset 2015) that minimizes the sum-of-costs while keeping the probability of collisions below some threshold. Our constraint is stricter as we aim to avoid any collisions that re-

sult from k delays. In addition, they modified M* while we work with CBS, which outperforms M* in many cases (Boyarski et al. 2015). Ma et al. (2017) also proposed a MAPF algorithm that handles unexpected delays. Their algorithm was also based on CBS, but they aimed to minimize the solution makespan while we aim to minimize the sum-of-costs. Moreover, their algorithm did not provide any guarantee on the robustness of the solution generated.

Problem Definition

A solution to a MAPF problem is a plan π consisting of n sequences $\pi_1, \dots \pi_n$ of move/wait actions such that π_i moves agent a_i from s_i to g_i, for every $i \in [1, n]$. $\pi_i(t)$ denotes the location that agent a_i would occupy after executing the first t move/wait actions in π without experiencing any delays. Each agent can perform one of five actions each time step. Either the agent moves to one of the four sides of his current location on the grid or the agent waits and stays at the same location.

Definition 1 (Conflict) *A conflict $\langle a_i, a_j, t \rangle$ in a plan π occurs iff agents a_i and a_j are located in the same location at time step t, i.e., when $\pi_i(t) = \pi_j(t)$.*[1]

We say that π is a **valid plan** if it is conflict free. A MAPF solver is *sound* if it outputs a valid plan. Multiple valid plans may exist for a given MAPF problem instance. Global cost functions that assign costs to plans are common; naturally, plans with lower costs are preferred. This paper focuses on a common cost function called the *sum of costs*, which is the summation of the number of time steps required by each agent to reach its goal (Standley 2010; Standley and Korf 2011; Sharon et al. 2013b; 2015b). A plan is **optimal** if it is a valid of minimal cost.

k-Robust MAPF

Definition 2 (k-delay Conflict) *A k-delay conflict $\langle a_i, a_j, t \rangle$ in a plan π occurs iff there exists $\Delta \in [0, k]$ such that agents a_i and a_j are located in the same location in time steps t and $t + \Delta$, respectively, i.e, when $\pi_i(t) = \pi_j(t + \Delta)$.*[2]

We say that a plan π is k-robust if it does not have any k-delay conflicts. Informally, this means that no conflicts will occur even if some of the agents are delayed by up to k time steps. The problem we address in this paper is how to find optimal sum-of-costs k-robust plans.

[1] Conflicts can also occur on edges, where agents traverse the same edge in different directions. We focus on conflicts on vertices for ease of presentation.

[2] Notice that setting $k > 0$ also prevents conflicts on edge of the form mentioned in the previous footnote. Also, it disallows a "train"-like motion where an agent moves to a location that is was occupied by another agent in the previous time step. This "train"-like motion is not allowed in some MAPF formulation.

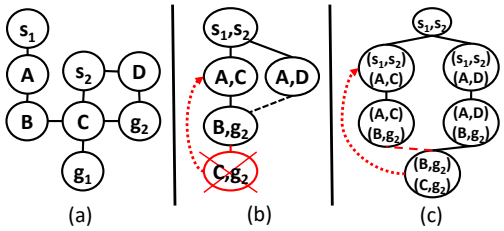


Figure 1: A MAPF problem (a) its search tree (b), and its k-robust search tree (c). The red lines show k-delay conflicts.

A*-based Solutions

The A* family of algorithms are natural solvers for MAPF (Silver 2005; Standley 2010; Goldenberg et al. 2012; Wagner and Choset 2015). They search in a n-agent state space which includes all the possible ways to place n agents into $|V|$ vertices, one agent per vertex. The *start* and *goal* states are $(s_1, \dots s_n)$ and $(g_1, \dots g_n)$, respectively. An action in this state space represents n move/wait single-agent actions, one single-agent action per agent. An action is applicable if the single-agent actions do not create conflicts. Clearly, a path from the start state to the goal state corresponds to a valid plan.

One way of adapting A* solvers to return k-robust plans is to modify state generation to prevent combinations of single-agent actions that lead to k-delay conflicts. However, this may lead to the solver returning non-optimal plans. For example, consider the problem in Figure 1(a), and assume we want a 2-robust plan. The initial configuration at time step 0 is (s_1, s_2). Note that the optimal plan (s_1, A, B, C, g_1) for a_1 and (s_2, D, g_2) for a_2 is 2-robust. Let's follow A* on this problem. First, A* expands state (s_1, s_2), generating two children (A, C) and (A, D). Assume that (A, C) was expanded first, generating state (B, g_2) with cost 4 (2 per agent). Next, (B, g_2) is expanded. Since a_2 was in C at $t = 1$, state (C, g_2) will not be generated due to the 2-robustness constraint. Next, state (A, D) is expanded. It will not generate (B, g_2), as this state was already reached via state (A, C) with the same cost (see Figure 1(b)). Thus, while there is a plan in which state (B, g_2) generates state (C, g_2), this specific run of A* will not find it. Indeed, in this case this will result in finding a suboptimal plan.

To remedy this, we need to modify the n-agent state space so that it records in each state the last k steps of each agent. Thus, in this state space a state represents a possible way to place n agents into $|V|$ vertices, one agent per vertex, *over $k - 1$ consecutive time steps*. Figure 1(c) shows the search tree of this extended state space. This state space, however, is exponentially larger than the n-agent state space because the same configuration at time step t might have many different possible configurations in the preceding $k - 2$ time steps.

Conflict-Based Search Solutions

Conflict-based search (CBS) (Sharon et al. 2015a) is a state-of-the-art MAPF solver that does not explicitly search the n-agent state space. In CBS, agents are associated with constraints of the form $\langle a_i, v, t\rangle$, representing that agent a_i is prohibited from occupying vertex v at time step t. A *consistent path* for agent a_i is a path that satisfies *all* of a_i's constraints, and a *consistent plan* is a plan composed only of consistent paths. Note that a consistent plan can be *invalid* if it contains conflicts despite each path satisfying the individual agent constraints.

CBS works by searching a *constraint tree* (CT) for a set of constraints such that a consistent plan w.r.t. this set of constraints is optimal. The CT is a binary tree, in which each node N contains: **(1)** a set of constraints imposed on the agents ($N.constraints$), **(2)** a single plan ($N.\pi$) consistent with these constraints, and **(3)** the cost of $N.\pi$ ($N.cost$).The root of the CT contains an empty set of constraints (thus, every plan is consistent with the root). A successor of a node in the CT inherits the constraints of the parent and adds a single new constraint for one agent. Generating a successor node N means finding a plan consistent with $N.constraints$ and identifying the conflicts in this plan, if they exist. A CT node N is a goal node when $N.\pi$ is valid. To search the CT for a goal node CBS runs a best-first search where nodes are ordered by their costs ($N.cost$).

Next, we describe three key components of CBS: how it finds a consistent plan to a given CT node N, how it identifies conflicts in N, and how to expand N and generate its successor CT nodes.

Finding a consistent plan. The algorithm used to find consistent plans is referred to as the CBS low-level solver. Any optimal single-agent path-finding algorithm can serve as a low level of CBS. A simple and effective low-level solver used in (Sharon et al. 2015a) is A* with the true shortest distance heuristic (ignoring constraints). Ties between low-level nodes were broken by preferring paths with fewer conflicts with known paths of other agents.

Identifying conflicts in a consistent plan. Once a consistent path has been found (by the low-level solver) for each agent, these paths are *validated* with respect to the other agents by simulating the movement of the agents along their planned paths ($N.\pi$). If all agents reach their goals without any conflict, N is declared as the goal node, and $N.\pi$ is returned. If, however, while performing the validation a conflict is found for two (or more) agents, the validation halts and the node is declared as a non-goal.

Resolving a conflict - the *split* action: When a non-goal CT node N is chosen in the best-first search of the CT, we generate its successor CT nodes. This is done by attempting to resolve a conflict in $N.\pi$. Let $\langle a_i, a_j, t\rangle$ be a conflict in $N.\pi$. This means $N.\pi_i(t) = N.\pi_j(t)$. Denote this location by v. We know that in any valid plan at most one of the conflicting agents, a_i or a_j, may occupy vertex v at time t. Therefore, at least one of the constraints, $\langle a_i, v, t\rangle$ or $\langle a_j, v, t\rangle$, must be satisfied. Consequently, CBS *splits* N and generates two new CT nodes as children of N, each adding one of these constraints to the previous set of constraints, $N.constraints$. Note that for each (non-root) CT node the low-level search is activated only for one agent – the agent for which the new constraint was added.

k-Robust CBS

Next, we describe *k-robust CBS* (kR-CBS), an adaptation of CBS designed to return optimal k-robust plans. kR-CBS differs from CBS in how it identifies goals, and in how it identifies and resolves conflicts.

Identifying k-delay conflicts. After the low-level solver returned a consistent plan for a CT node N, kR-CBS scans $N.\pi$ for k-delay conflicts by simulating the paths and checking for conflicts with the k-last locations of all other agents. Thus, finding a k-delay conflict $\langle a_i, a_j, t\rangle$ means that $N.\pi_i(t) = N.\pi_j(t + \Delta)$ for $\Delta \in [0, k]$. This process is easy to implement but its runtime is larger by a factor of k from the equivalent plan validation step in CBS. N is a goal CT node iff it has no k-delay conflicts.

Resolving conflicts (splitting CT nodes). Let N be a non-goal node in the CT selected to be expanded next by kR-CBS, and let $\langle a_i, a_j, t\rangle$ be a k-delay conflict in N. This means that there is a vertex v and a value $\Delta \in [0, k]$ such that $v = N.\pi_i(t) = N.\pi_j(t + \Delta)$. Note that there is no k-robust plan in which a_i is at v at time t while a_j is at v at time $t+\Delta$. Therefore, at least one of the constraints, $\langle a_i, v, t\rangle$ or $\langle a_j, v, t+\Delta\rangle$, must be added to the CT and must be satisfied by the low-level solvers. Consequently, kR-CBS generates two children to N, each having one of these constraints.

Proving that kR-CBS is sound and complete is straightforward. It is sound because it only halts when generating a CT node that has no k-delay conflicts. It is complete because when splitting a CT node we do not lose any valid plans. Similarly, kR-CBS returns optimal plans, as it searches the CT in order of the nodes' costs, and the cost of a node N is a lower bound on the cost of any optimal plan consistent with $N.constraints$.

Example. Consider a 2-robust MAPF problem on the graph in Figure 2(a), with two agents whose start-goal pairs are s_1-g_1 and s_2-g_2, respectively. Figure 2(b) shows the first two levels of the CT generated by kR-CBS, where every node N shows $N.constraints$ (labeled Con), $N.\pi_1$, $N.\pi_2$, and $N.cost$. Observe that the plan in the root is valid, but is not 2-robust, having a 2-delay conflict $\langle a_2, a_1, 2\rangle$ at location B for $\Delta = 1$, since $\pi_1(3) = \pi_2(2) = B$. To try to resolve this conflict, kR-CBS adds the constraint $\langle a_2, B, 2\rangle$ to the left child and the constraint $\langle a_1, B, 3\rangle$ to the right child. Both children of the root node are also not goal nodes. In fact, in this example we will need to generate a total of 7 CT nodes before finding an optimal plan. As we show next, it is possible to modify kR-CBS to find the goal sooner.

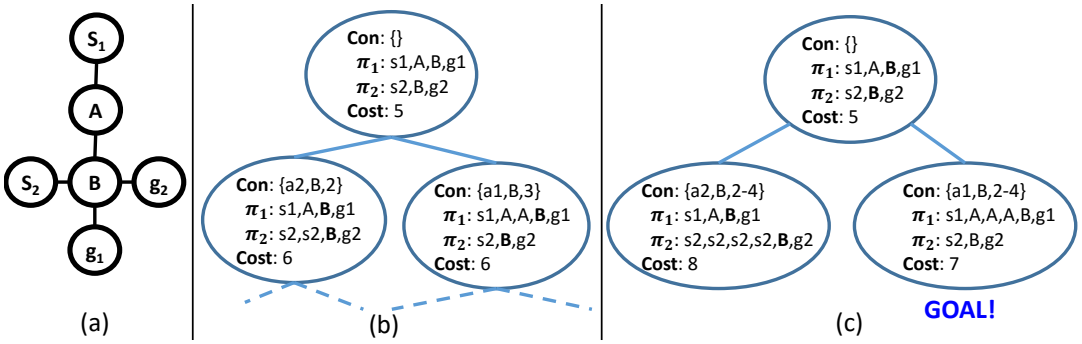


Figure 2: (a) The graph (b) The CT using the original time/location constraints (c) CT using the range constraints

Improved k-Robust CBS

kR-CBS is complete because whenever it expands a CT node N it generates two children N_1 and N_2 such that any solution consistent with $N.constraints$ will satisfy either $N_1.constraints$ or $N_2.constraints$. Thus, no solution is lost by considering N_1 and N_2 instead of N. For example, in the root CT node shown in Figure 2(b) there is no 2-robust plan in which a_1 is in B at time 2 and a_2 is in B at time 3. So, in every 2-robust plan either a_1 is not at B at time 2 or a_2 is not in B at time 3 Note, that this argument can be extended: in every 2-robust plan either a_1 is not in B at time 2 or a_2 is not in B at time 4. Thus, we can impose a stricter constraint on the left subtree by adding the constraint $\langle a_1, B, 4\rangle$. Imposing more constraints per CT node can reduce the size of the CT tree, and consequently the overall runtime.

To exploit this understanding, we introduce the Improved kR-CBS (I-kR-CBS) that resolves conflicts in a CT node N by imposing *range constraints* on its successors. A *range constraint* is defined by the tuple $\langle a_i, v, [t_1, t_2]\rangle$ and represents the constraint that agent a_i must avoid vertex v from time step t_1 to time step t_2. Ideally, we would like to construct range constraints as large as possible, to minimize the size of the CT tree. However, over-constraining CT nodes may result in losing completeness and optimality. The key question is thus which pair of range constraints can be safely used to resolve conflicts without loosing completeness and optimality.

Definition 3 (Sound Range Constraints) *A pair of range constraints are called* sound *iff all optimal k-robust plans satisfy at least one of these constraints.*

Corollary 1 *A kR-CBS variant that uses range constraints is sound, complete, and returns optimal k-robust plans if it resolves conflicts only with sound pairs of range constraints.*

Proof outline: For a CT node N let N_1 and N_2 be its children, generated by the sound pair of range constraints R_1 and R_2, respectively. Now, $\pi(N)$ denotes all the k-robust plans that do not violate $N.constraints$. Observe that $\pi(N_1)$ contains all the plans in $\pi(N)$ that satisfy R_1 (but violate R_2), and similarly $\pi(N_2)$ contains all the plans in $\pi(N)$ that satisfy R_2 (but violate R_1). Thus, $\pi(N) = \pi(N_1) \cup \pi(N_2)$, since there is no plan that violates both R_1 and R_2 as they are a sound pair of constraints. Thus, splitting CT nodes by resolving conflicts with a sound pair of constraints does not lose any plans and thus preserves the soundness, completeness, and optimality. □

Corollary 2 (Symmetric range constraints) *For any time step t, vertex v, and agents a_i and a_j, the range constraints $\langle a_i, v, [t, t+k]\rangle$, $\langle a_j, v, [t, t+k]\rangle$ are sound for solving a k-robust MAPF problem.*

Proving Corollary 2 is straightforward. We call a pair of range constraints *symmetric* if they constrain the same vertex and the same time range. Note that setting a symmetric range constraint on a range larger than k is in general not sound. Thus, Corollary 2 gives an upper bound on the size of the largest pair of symmetric range constraints that is sound. Nevertheless, there is more than one k-sized symmetric pair of range constraint for a given k-delay conflict $\langle a_i, a_j, t\rangle$ over vertex v. For example, in our implementation, we used the time range $[t, t+k]$, but the pair of range constraints $[t-k, t]$ is also sound.

A pair of sound range constraints can also be *asymmetric*, i.e., constrain one agent to a longer time range than the other agent. For example, consider a conflict $\langle a_i, a_j, t\rangle$ over vertex v and pair of range constraints $R_1 = \langle a_i, v, [t-k, t+k]\rangle$ and $R_2 = \langle a_j, v, [t]\rangle$. R_1 and R_2 are a sound pair of constraints, because a solution must satisfy either R_1 or R_2, since violating both results in a k-delay conflict. R_1 and R_2 are extremely asymmetric, but one can imaging asymmetric range constraints that are more balanced. An open question for asymmetric range constraints is how to choose which agent to impose the more restricted constraint upon. In our experiments below we implemented the symmetric range constraints and the aforementioned extreme asymmetric constraints (of one time point and a $2k$ time range) with arbitrary choice of the more constrained agent.

Improvements of CBS A number of improvements to CBS have been introduced. In *Meta-agent CBS* (MA-CBS) (Sharon et al. 2012) agents with many mutual conflicts are merged into a *meta-agent*. Meta-agents are then treated as a joint composite agent by the low-level solver. A k-robust version of MA-CBS requires that the

low-level solver is also k-robust for meta-agents consisting two or more agents. Otherwise, the meta-agent might end up having internal k-delay conflict.

Improved CBS (ICBS) (Boyarski et al. 2015) adds three technical enhancements to CBS. First, it splits conflicts with high likelihood to cause increase in the f-cost below the corresponding node. Second, it provides a way to "bypass" some conflicts and avoid adding nodes to the CT. Finally, it restarts the search from scratch when merging actions occur in MA-CBS. These improvements apply directly to a k-robust MA-CBS solver and no further adjustments need to be done.

Experimental Results

Next, we experiment with kR-CBS and I-kR-CBS using symmetric and asymmetric pairs of range constraints. Random MAPF problem instances were generated in an open 8x8 grid. Then a kR-MAPF solver for $k = 0, 1,$ and 2 was executed and the resulting plan cost and the CPU runtime were measured.

Table 1 shows the average plan cost and average CPU runtime when finding k-robust solutions using kR-CBS (labeled KR) and I-kR-CBS with the asymmetric and with the symmetric range constrains (labeled IKR(A) and IKR(S), respectively) for 4, 6, 7, 8, 9, and 10 agents (different rows). Note that the plan cost was identical for all solvers, so we only show this once,. Note also that $k = 0$ is standard CBS.

First, consider the plan costs. As can be seen, the k robust plans are not much more costly than a plan for the basic definition of MAPF (i.e., for $k = 0$). This suggests that searching for k-robust plans is advisable if one needs a safety zone or expects delays. Next, as expected, both I-kR-CBS variants runs much faster than kR-CBS and this improvement increases when increasing k and when more agents exist. When comparing the symmetric and the asymmetric range constraints, we see a clear advantage for the symmetric range constraints. We conjuncture that this is due to the arbitrary way in which we choose which agent to constrain more when using the asymmetric range constraints. Future work may investigate a more intelligent way of doing so.

We also performed some experiments on a larger map from the Dragon Age Origins video game, which is available in the movingai repository (Sturtevant 2012). Specifically, we generated 90 randomly generated instances with 30 agents on the `brc202d` map, which has 43,151 vertices. This map is very large, so that a k-robust plan often has the same cost as a plan that is not robust. When averaging over 50 random instances, the average plan cost was 3,818.35, 3,818.43, and 3,818.53 for $k = 0$, 1, and 2, respectively. Indeed, the plan cost grows with k, but negligibly. This emphasizes the usefulness of finding a k-robust plan, as one can be found in such a domain without wasting too much of the plan cost. That being said, finding k-robust plans is more time consuming. In the above experiments, finding the k-robust plans required 213, 284, and 381 seconds, for $k = 0$, 1, and 2, respectively.

	Cost			Planning time (ms)					
	0	1	2	0		1		2	
Obs		Picat		CBS	Picat	CBS	Picat	CBS	Picat
12	33.06	35.12	36.72	**77**	1,523	**1,627**	3,464	**559**	5,293
16	36.78	39.07	40.49	4,003	**2,008**	**4,883**	5,243	**7,227**	7,524
19	36.38	39.17	43.07	**696**	2,025	**2,122**	4,672	**6,836**	9,012
22	30.66	34.39	36.77	**587**	1,197	6,594	**3,959**	14,073	**7,464**
25	24.42	28.86	30.30	1,700	**912**	12,878	**3,489**	20,902	**5,443**
32	14.46	16.26	18.57	5,609	**156**	**107**	526	1,582	**1,213**
38	9.33	11.20	12.38	553	**111**	2,179	**313**	**387**	446

Table 2: Experimental results comparing the Picat-based kR-MAPF solver and kR-CBS. The table shows the average plan cost and running time, for different values of k and different number of obstacles.

A Constraint Programming Solution

An alternative approach to solve MAPF problems is to compile them into other known NP-hard problems that have mature and effective general purpose solvers (Surynek 2012; Yu and LaValle 2013a; Erdem et al. 2013; Surynek et al. 2016). In particular, Surynek and others have developed effective optimal MAPF solvers that are based on encoding the MAPF problem to Boolean Satisfiability (SAT) and then applying a state-of-the-art SAT solver. Indeed, modern SAT solvers are known to be very effective and these SAT-based solvers have shown to be comparable and sometimes even better than other MAPF solvers.

Generally speaking, these *compilation-based* approaches express a set of *constraints* that define the MAPF problem and then call a general purpose solver, e.g., a SAT solver or a Mixed Integer Linear Program (MILP) solver, to obtain the solution. Adapting such solvers to be k-robust is relatively simple, requiring a simple modification to these constraints. To demonstrate this, we implemented a MAPF solver using the Picat (Zhou, Kjellerstrand, and Fruhman 2015), a logic-based programming language that is based on Prolog. The encoding we used follows Surynek's SAT-based MAPF solver (Surynek et al. 2016), in which there is a Boolean variable for every triplet (a, t, v) of agent (a), time (t), and location (v), where this variable is true iff agent a is planned to be at location v in time t. A set of constraints are imposed on these variables, namely:

1. Each agent occupies exactly one vertex at each time step.
2. No two agents occupy the same vertex at any time.
3. In every time step an agent may only transition between two adjacent locations.

For producing k-robust solutions, the second constraint is extended such that No two agents occupy the same vertex in time steps that are closer than k from each other. The exact Picat model we developed is available at `https://tinyurl.com/kRobust`. The advantage of using Picat to encode MAPF is that it can compile to SAT, to a constraint program (CP), or to a Mixed Integer Linear Program (MILP). In our experiments we only run a SAT compilation.

	Plan cost			Plan time (ms)						
				k=0	k=1			k=2		
m	k=0	k=1	k=2	All	KR	IKR(A)	IKR(S)	KR	IKR(A)	IKR(S)
4	21	22	22	6	15	14	15	193	110	67
6	31	32	32	5	28	26	20	990	388	94
7	36	37	39	7	31	26	17	1,618	826	184
8	41	41	43	6	29	23	20	2,625	1,051	229
9	48	49	51	9	379	218	76	20,006	4,408	556
10	49	51	53	41	162	124	78	22,464	7,097	875

Table 1: Average plan cost and planning runtime for different CBS-based k-robust solvers, on an 8x8 open grid

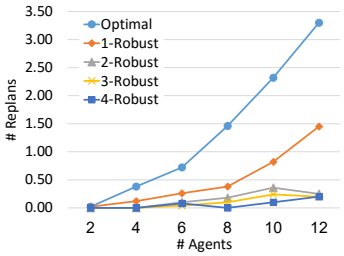


Figure 3: KR-MAPF for MCP

Experimental Results

We evaluated experimentally our Picat-based kR-MAPF solver in a similar 8x8 grid problems as described above. Our aim in these experiments is to demonstrate that a Picat-based solver for kR-MAPF is feasible and comparable with kR-CBS. A more comprehensive comparison between Picat-based and CBS-based approaches includes evaluating on a range of MAPF domains and is beyond the scope of this paper.

Table 2 shows the average plan cost and planning runtime for problem instances with 6 agents and $k = 0, 1$, and 2. The experiments in Table 1 was on open grids with no obstacles. Here, we experimented with problem instances with a different number of randomly allocated obstacles (the "Obs." column). As expected, increasing k results in plans of higher cost, and, higher runtime. For both algorithms, the impact of varying the number of obstacles on the planning time follows a classical easy-hard-easy pattern: with either a few or many obstacles is easy, and it becomes harder for the middle-ground, where the problem is not under- or over-constrained. Note that the plan cost is small for the over-constrained setting (where the number of obstacles is 32 – half the cells in the 8x8 grid) since we only experimented with solvable instances.

Now we compare the results of kR-CBS and the Picat-based solver. Since both Picat and kR-CBS solvers solve kR-MAPF optimally, their solution cost is the same, and the comparison between them is in the runtime of finding the optimal solution. We can see that in general, CBS is better for the less constrained settings – with fewer obstacles, while the Picat-based solver is better in the denser scenarios (having more obstacles). To explain this, observe that having more obstacles results in higher chances for paths found by the individual agents to have a conflict with each other, and consequently higher kR-CBS runtime.[3]

[3]For $k = 0$ there is a slight difference in the problem formulation used by kR-CBS and the Picat-based solver: in Picat the agents could swap locations while this caused an edge conflict in kR-CBS. Thus, the runtime results for $k = 0$ are not directly comparable. However, for $k > 0$ both algorithms do not allow edge conflicts and the results can be compared safely.

Execution Policies

A k-robust plan is not enough to provide a completely robust solution, as there may be more than k delays. To this end, we use the two-phase framework proposed by Ma et al. (2017) and mentioned in the introduction. After an original (possible k-robust) plan is generated, delays are handled online by modifying the plan according to a given *execution policy*. In this section we propose several intuitive execution policies that integrate well with having a k-robust plan.

In the following discussion on execution policies, we consider the following setting. The agents are collaborative and share their location, e.g., via a centralized controller. So, all agents are notified whenever an agent is delayed. If, due to unexpected delays, an agent finds that its destination location is occupied by another agent it will stay idle in its current location to avoid a collision. In this setting, an *execution policy* accepts as input the *plan* the agents are following (π), the *current time step* (t), and the set of *delayed agents*, i.e., the agents that were just delayed from performing their planned action. An execution policy is activated whenever one or more agents are delayed and may decide to modify π to take into account these delays. We classify execution policies according to *when* they modify π and *how* they do so.

When to Modify the Plan

We experimented with three classes of execution policies that are invoked after a delay is detected at time t. They differ in when they choose to modify π as follows:

(1) Eager. Modify immediately when a delay occurs.

(1) Reasonable. Modify only when a delay occurs and that delay is expected to cause a conflict later between the delayed agent and some other agent.

(1) Lazy. Modify only if a conflict is expected to occur in $t+1$ between the delayed agent and some other agent.

Checking whether a conflict is expected to occur is done by simulating the execution of π from the current locations of the agents. The logic behind "Reasonable" is that if a conflict is expected occur then it is best to modify π to avoid it as soon as possible, while the logic behind "Lazy" is that future unexpected delays may resolve expected conflicts even without modifying π earlier.

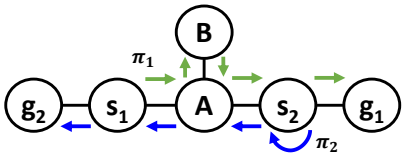


Figure 4: Example where lazy repair leads to a deadlock

How to Modify a Plan

The different execution polices can also be classified by how the plans are modified:

(1) Replan policies. These policies create a completely new plan by using the same MAPF solver used to generate the original plan but execute it on the current state at time t.

(2) Repair policies. These policies perform minor modifications to π by forcing some of the agents that were not delayed to wait at time step $t+1$. Specifically, we experimented with two such repair policies that differ by which agents are enforced to wait at time step $t+1$: **(i) MCP.** Only the agents that are expected to conflict due to the delay of agent a_i at time step t. This policy is based on the Minimal Communication Policy (MCP) suggested by Ma et al. (2017). **(ii) All.** All agents that were not delayed at time t are forced to wait at time step $t+1$. The advantage of "All" is that unless other delays occur, the configuration at time step $t+1$ is identical to the original configuration at time step t. This is advisable in scenarios where preserving the relative positions of agents during the plan is important. Obviously, there is a tradeoff. Replan policies incur significant CPU overhead compared to repair policies, but they may end up having lower execution cost. In particular, when there are many delays it may be better to create a completely new plan than to modify the original one.

Combinations of Execution Policies

The choice of when to modify π is orthogonal to the choice of how to update it. Thus, we have 9 execution policies: {Eager, Reasonable, Lazy} $\times$ {Replan, MCP repair, All repair}. Some configurations are problematic. Lazy-MCP and Lazy-All are ill-defined, as when a conflict is expected to occur in the next time step, it is not clear which of the conflicting agent should wait. Moreover, Lazy-All is not complete and may end up in a deadlock, as demonstrated in Figure 4. Assume that the original 1-robust plan π is: $\pi_1 = (s_1, A, B, B, B, A, s_2, g_1)$ and $\pi_2 = (s_2, s_2, s_2, A, s_1, g_2)$. If a_1 is delayed in s_1 for two time steps, we reach a state where a_1 is at s_1 and a_2 is at A. At this stage, a conflict occurs as according to π the agents will cross paths. But, enforcing either of them to wait will not help. Thus, we did not implement Lazy-All and Lazy-MCP. In addition, since MCP only delays the agents involved in an expected conflicts then Eager-MCP is in fact equivalent to Reasonable-MCP.

	Cost	#Modifications	Time (ms)
Delay probability = 0.001			
Eager All	109.12	0.24	**0.00**
Reasonable All	106.64	**0.03**	**0.00**
Reasonable MCP	106.52	0.21	**0.00**
Eager replan	**106.18**	0.21	6.93
Reasonable replan	106.30	**0.03**	15.93
Lazy replan	106.58	0.06	2.67
Delay probability = 0.01			
Eager All	127.03	1.52	**0.00**
Reasonable All	112.91	**0.39**	**0.00**
Reasonable MCP	109.42	2.00	**0.00**
Eager replan	**106.18**	1.06	228.34
Reasonable replan	106.52	0.76	960.85
Lazy replan	109.03	0.61	379.36
Delay probability = 0.1			
Eager All	535.21	26.67	**0.00**
Reasonable All	268.97	9.06	**0.00**
Reasonable MCP	124.27	7.64	**0.00**
Eager replan	**106.21**	26.42	25802.06
Reasonable replan	111.97	10.12	10955.64
Lazy replan	128.09	**3.52**	2173.98

Table 3: Comparison of the different replanning polices

Experimental Results: Repair vs. Replan

We compared the different execution polices on an 8x8 open grid with 20 agents. Delays were inserted randomly, with probability p per each move of each agent, where p is a parameter. We experimented with $p = 0.1, 0.01$, and 0.001. In this set of experiments the original plan was optimal but 0-robust. Table 3 presents our results, averaged over 32 instances. The columns report the execution costs (i.e., the overall sum-of-costs incurred until all agents reached their goals), the number of times the plan was modified, and the CPU runtime in ms required by the execution policies.

A few observations are learned. First, the replan polices achieve significantly better execution costs than the repair polices (All and MCP). On the other hand, replanning is significantly more costly in terms of CPU overhead while repairing is done instantaneously, thereby establishing a tradeoff between execution cost and (re)planning time. Second, the CPU time and execution costs increase with the delay probabilities. Third, within the repairing policies Reasonable MCP achieves the minimal execution cost. This is because its repair policy forces only a subset of the agents to wait instead of all of them. On the other hand, the resulting executed plan of Eager-All reverts back to the original plan as soon as possible (losing one time step) and this may be desirable in several circumstances (Felner et al. 2007). Forth, as expected, Eager required more modifications then Reasonable, and Lazy required the fewest. The number of modifications directly contributes to the overall CPU time spent in replanning (the "Time" column). On the other hand, Eager replanning achieved better execution cost, as it replans earlier. Nevertheless, the execution costs of all the replanning polices were not far from each other. To summarize, there are tradeoffs and each policy has pros and cons. One should

	Cost		#Modifications	
	0-robust	1-robust	0-robust	1-robust
Eager All	89.10	91.76	6.50	6.40
Reasonable All	55.22	54.00	0.78	0.23
Reasonable MCP	50.94	52.14	0.90	0.19

Table 4: Repair policies for 0- and 1-robust input plan

choose the policy that is best suited for the given circumstance.

Figure 3 shows the results of using a k-robust plan as the input plan π to the Reasonable-MCP policy, which provided a good balance between execution cost, number of modifications, and replanning time (see Table 3). The x-axis corresponds to number of agents while the y-axis shows the average number of modifications. The different curves represents different values of k. The results show the benefit of using a k-robust plan: by increasing k we reduce the number of modifications. Indeed, when $k = 4$ the number of modifications is close to zero, even for 12 agents, while it increases to almost 3.5 when using the optimal plan (where $k = 0$).

Finally, Table 4 compares the different repair polices for 9 agents on the 8x8 grid when the input plan was 0-robust and 1-robust. Reasonable All and Reasonable MCP are very close together and are much better than Eager. In addition, we see that using an input of 1-robust plan results in fewer replans for all polices, while incurring minimal added execution cost.

Discussion and Conclusion

In this paper we studied how to modify MAPF planners to cause them to generate multi-agent plans that more robust to unexpected changes. We formalized this as the k-robust MAPF problem, where k is the number of delays each agent can experience while still preserving the ability to follow the generated plan. Then, we discussed several execution policies that can be coupled with a k-robust plan to provide a completely robust solution. The tradeoffs between these variants were studied, and showing that k-robust plans can be found with relatively small overhead in plan cost.

We discussed how to create a k-robust version for A*, CBS, and a constraint programming approach. However, there are other optimal MAPF solver that are also successful. Future work can adapt them to find k-robust plans as well. For example, the ICTS algorithm (Sharon et al. 2013b) can be adapted to find k-robust plans in a similar manner as A* based solvers, , but its tree pruning heuristic will be more costly, requiring comparison with the last k time steps. Another direction for future work is to develop MAPF solvers that generate plans that can be followed with probability greater than a parameter (Wagner and Choset 2017), and to study more reactive execution policies.

Acknowledgments

This research was supported by the Israel Ministry of Science and the Czech Ministry of Education Youth and Sports through a joint grant given to Roni Stern, Ariel Felner Pavel Surynek, and Roman Bartak.

References

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Shimony, E.; Bezalel, O.; and Tolpin, D. 2015. Improved conflict-based search for optimal multi-agent path finding. In *IJCAI-2015*.

Erdem, E.; Kisa, D. G.; Oztok, U.; and Schueller, P. 2013. A general formal framework for pathfinding problems with multiple agents. In *AAAI*.

Felner, A.; Stern, R.; Rosenschein, J. S.; and Pomeransky, A. 2007. Searching for close alternative plans. *Autonomous Agents and Multi-Agent Systems* 14(3):211–237.

Goldenberg, M.; Felner, A.; Stern, R.; and Schaeffer, J. 2012. A* Variants for Optimal Multi-Agent Pathfinding. In *Workshop on Multi-agent Path finsing. Colocated with AAAI-2012*.

Ma, H.; Kumar, S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *AAAI*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. 2012. Meta-agent conflict-based search for optimal multi-agent path finding. In *Symposium on Combinatorial Search (SOCS)*.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013a. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* 195:470–495.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013b. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* 195:470–495.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015a. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.* 219:40–66.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015b. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219:40–66.

Silver, D. 2005. Cooperative pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 117–122.

Standley, T. S., and Korf, R. E. 2011. Complete algorithms for cooperative pathfinding problems. In *IJCAI*, 668–673.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games* 4(2):144–148.

Surynek, P.; Felner, A.; Stern, R.; and Boyarski, E. 2016. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*.

Surynek, P. 2010. An optimization variant of multi-robot path planning is intractable. In *AAAI*.

Surynek, P. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI*. 564–576.

Wagner, G., and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artificial Intelligence* 219:1–24.

Wagner, G., and Choset, H. 2017. Path Planning for Multiple Agents Under Uncertainty. In *IROS (to appear)*.

Yu, J., and LaValle, S. M. 2013a. Planning optimal paths for multiple robots on graphs. In *ICRA*, 3612–3617.

Yu, J., and LaValle, S. M. 2013b. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*.

Zhou, N.-F.; Kjellerstrand, H.; and Fruhman, J. 2015. *Constraint solving and planning with Picat*. Springer.

Integrating Execution and Rescheduling

Jeremy Frank
NASA Ames Research Center
Mail Stop N269-3
Moffett Field, California 94035-1000

Abstract

Plan execution systems can represent very complex plans with conditions, loops, sequences, and other structures. However, they cannot represent every possible ordering of large plans or schedules explicitly, and it is helpful to integrate them with a planner or scheduler in the event that complex plan synthesis tasks must be performed during execution. While full-featured planners may be needed in autonomous systems, the high complexity of these planners may make them too slow, too large (in memory), or both, to be used in applications with limited computational resources and low latency response time requirements. We describe how PLEXIL, a plan execution system, is integrated with a Single Machine Scheduler (SMS) scheduler. The scheduler is not as powerful as a planner, but it can handle unexpected events that PLEXIL all by itself is not able to handle. We formally describe the integration between scheduler and executive for a fixed number of activities, then describe how the integration can be extended to more sophisticated problems (arbitrary numbers of tasks). This treatment will pave the way for the formal integration of execution with more complex scheduling tasks, and eventually, automated planning.

Introduction

Planning and Execution have been integrated in a variety of systems. Plan execution systems can represent very complex plans with conditions, loops, sequences, and other structures. It is helpful to integrate plan execution systems with a planner or scheduler in the event activity reordering, or more complex plan synthesis tasks, must be performed. While full-featured AI planning systems may be generally useful, the high complexity of such planners may make them too slow, too large (memory), or both, to be used in realtime. Finally, while a plan execution system may (and should) be able to load new plans or schedules, it is beneficial if the plan execution system can receive updates from the planner or scheduler to respond to unexpected events in real time.

To provide motivation, let us assume the system being managed is a robotic spacecraft. Changes to mission goals can change action duration, e.g. duration of image acquisition tasks. Changes to a spacecraft's communications plan or orbit can change activities' feasibility windows, e.g. when to communicate and communication windows duration is a function of orbit and ground station location. Faults can change activity duration or conditions on activity completion. Spacecraft are notorious for having limited compute resources, both time and memory, due to mass, radiation, and other considerations, which place a premium on the complexity of computations to handle unexpected events. Notably, the scheduler may need to make changes in the schedule because of unexpected events detected at execution time, or due to faults or unexpected events that may change the future schedule. In addition, the executive must be designed to invoke the scheduler at the right time, and receive updates. Finally, in real-time applications with limited compute resources, timing considerations drive when to reschedule and what rescheduling algorithms to use.

In this paper, we describe the integration of the Plan Execution Interchange Language (PLEXIL) executive with a scheduler that responds to unexpected events during execution of the schedule. Our approach here is not to create novel rescheduling algorithms, but to show formally how scheduling and execution are integrated. We choose to integrate PLEXIL with scheduling to demonstrate small, but significant, increases in functionality when compared with the executive alone. A scheduler is also more likely to fit within the limited spacecraft compute resources budget than a full-featured AI planner.

We will first describe key features of PLEXIL. Then we describe Single Machine Scheduling (SMS), and in particular, how unexpected events and faults can cause rescheduling of SMS activities. Then we describe the PLEXIL SMS integration in phases. The integration is first described for a fixed size known schedule that can only be disrupted in the future by faults or unexpected events. Next, we show how to handle execution time uncertainty in schedules of fixed size. Then we extend this integration to show how PLEXIL can manage SMS schedules of arbitrary size. Finally, we conclude and discuss future work.

PLEXIL

PLEXIL (Verma *et al.* 2006) is a plan execution language and software system. Plans in PLEXIL are written similar to programs; PLEXIL has variables, looping, and input-output features. Unlike other programming languages, PLEXIL has first-class support for *tasks*, *task ordering and concurrence*, *conditions*, *hierarchical decomposition*, and *task execution*

state. Also, since PLEXIL is meant to work in real-time environments, there are some interface management constraints to ensure real-time performance. For the purposes of this paper, we describe only a small number of important PLEXIL features below.

- Conditions and State. PLEXIL tasks are represented by *nodes*, with *conditions* governing when nodes start and end. We will only use Start conditions and Repeat conditions, but other conditions are supported as well. As time proceeds, PLEXIL evaluates all conditions on nodes to update the *execution state* (e.g. if an unexecuted node's start conditions are satisfied, the node transitions from WAITING to EXECUTING).
- Control structures. PLEXIL supports traditional programming language control. By default, PLEXIL nodes execute at most once, when their Start conditions are satisfied; conditions also can be used to repeat nodes.
- State references. PLEXIL nodes may refer to the execution state of sibling, children, or ancestor nodes; this is especially helpful in node conditions (e.g. Start conditions).
- Concurrence. PLEXIL allows numerous node decomposition and ordering rules. We will make use of the *concurrent* decomposition, in which all child nodes of a parent can execute in arbitrary order, including 'simultaneously'.
- Lookups and queue handling. PLEXIL can look up information from external systems. Information is managed in queues. Lookups only read queues; function calls or commands inside PLEXIL nodes must remove information from these queues explicitly. The exception to this rule is time, which is continuously updated.
- Variable declarations and scoping. PLEXIL nodes use variables and typical programming operations to assign values to variables. Variables have scoping rules; child nodes in decompositions can share information by reading and writing the values of variables in their parents.
- Commands. PLEXIL nodes can issue function calls to arbitrary code.

Single Machine Scheduling and Rescheduling Scenarios

Single Machine Scheduling (SMS) is the problem of scheduling activities that are mutually exclusive, i.e. they may not overlap. While SMS comes in several varieties, we consider a version in which activities have disjoint release times and due dates, i.e. feasibility windows, and simple temporal constraints (Dechter, Meiri, & Pearl 1991). In general activities may have multiple feasibility windows. We assume all activities are mandatory, i.e. they must be performed. The traditional SMS problem is concerned with minimizing makespan; we treat the problem as a feasibility problem only. For more information on SMS see (Brücker 1998).

Notation: Let A be an activity. Let A^i be the i^{th} window A may occur in. Let A^i_s be the start time of window A^i, and A^i_e be the end time of A^i. Let A_d be the duration of A, A_s be the assigned start time of A, and A_e be the end time of A.

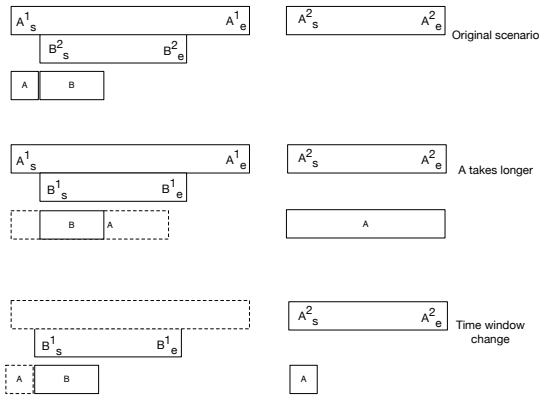


Figure 1: A has two windows, B has one window. $A^1 \prec B^1$ and $A_s = A^1_s$.

The list of unexpected events that may disrupt the schedule is as follows:

1. +A^i_s: Start time of A^i increases (A^i becomes smaller).
2. -A^i_s: Start time of A^i decreases (A^i becomes larger).
3. +A^i_e: End time of A^i increases (A^i becomes larger).
4. -A^i_e: End time of A^i decreases (A^i becomes smaller).
5. +A_d: activity A duration increases (includes +A_e).
6. -A_d: activity A duration decreases (includes +A_s, -A_e).

These events may occur either prior to the start of execution, or (in the case of +A_s, -A_d +A_d) after the start of execution, with different consequences, as discussed further below. The results of a disruption of the schedule could be:

- Activity order is unchanged, but activity start or end times may change (e.g. propagating constraints is sufficient to handle rescheduling).
- Activity time window assignment may change.
- Activity order may need to change.

Several scenarios are described in figures 1 and 2. Scenarios are limited to two activities with one or two windows each. They illustrate cases in which activity order changes, or time window assignments change, or both. Suppose w.l.o.g. that $A^1_s \leq B^1_s$ and that A is scheduled first. Then $A_s + A_d \leq B_s$. For brevity we will denote this $A^1 \prec B^1$. When actions have multiple windows, this notation allows us to specify a combination of window assignment and action ordering, e.g. $A^i \prec B^j$ indicates A is contained in A^i, B is contained in B^j, and $A_s + A_d \leq B_s$.

A rigorous analysis of the impact of all of the changes is outside the scope of this paper. However, it is worth making some points about the timing of unexpected events and their impact on rescheduling. If an unexpected event takes place prior the start of activity execution, the scheduler has the flexibility to reorder the activity or move it to a later window. However, if an activity starts or ends late, the scheduler is limited in the response it can take. Since the focus of our

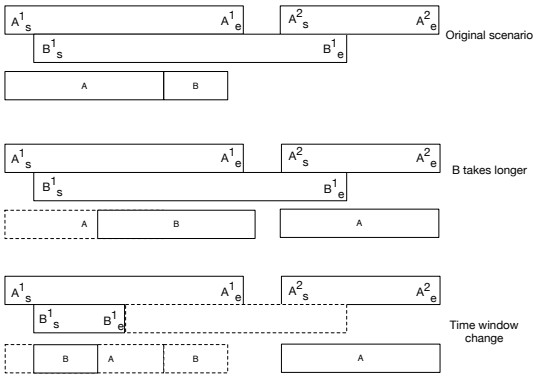


Figure 2: A has two windows, B has one window. $A^1 \prec B^1$ and $A_s = A^1_s$.

work is on the interface between the scheduler and the executive, especially when rescheduling is needed, we will assume that the change results in a new feasible schedule, even if activities must move to new windows. This precludes handling changes after A has started that cause A to end after the end of the window, or elimination of all feasible windows.

Scheduling and Execution with a Fixed Number of Activities

We now show how to write a PLEXIL plan that manages a fixed, known set of activities using a unary resource. First, we will write the PLEXIL assuming that activities start and end as scheduled, but that the future schedule is subject to disruption because of faults or unexpected events. We assume events make a single change as noted above, leading to a single new constraint. We then will rewrite the PLEXIL to allow activities to start or end late.

The PLEXIL plan for activities that are well behaved (start and end on time) but in which the future schedule is subject to new or changed constraints due to unexpected events, is shown below:

```
ManageSpacecraft:Concurrence {
  Constraint NewConstraint;
  ...
  DoActivityA: {
    Start
      (A_s == Lookup(t)) &&
      (ScheduleOnEvent.state!=EXECUTING)
    CmdA: a();
  };
  DoActivityB: {
    Start
      (B_s == Lookup(t)) &&
      (ScheduleOnEvent.state!=EXECUTING)
    CmdB: b();
  };
  ...
  ScheduleOnEvent: {
    Repeat TRUE;
    Start
      (Lookup(Event))
    NewConstraint=pop(Event);
    reschedule(NewConstraint);
    A_s=getstart(A);
    B_s=getstart(B);
    ...
  };
};
```

This plan appears quite simple, but uses a number of features of PLEXIL, and also shows how we will formally define the interface between PLEXIL and the scheduler. While this plan does not depend specifically on the scheduling problem being an SMS problem, this restriction will become more important in the next part of the paper.

The first feature to note is that the PLEXIL plan has one node for each activity. The second feature is that all of the nodes are declared *concurrent*. Strictly speaking, only one of the PLEXIL nodes is intended to be EXECUTING at any time, because all of the activities are mutually exclusive, and no activities should be started while rescheduling is taking place. However, PLEXIL must be able to execute the activities in arbitrary order (due to rescheduling). Declaring them as concurrent ensures that arbitrary execution order is possible. Third, note that ScheduleOnEvent has a Repeat condition. An unknown number of events may lead to rescheduling, so PLEXIL may need to repeat this node an arbitrary number of times. Fourth, the *interface* between PLEXIL and the scheduler is the set of function calls retrieving the variables in the activity start conditions. For this simple case, all that PLEXIL needs is the new activity start times, A_s, as recomputed by the scheduler. These values in PLEXIL are retrieved by synchronous get() commands that provide access to the A_s variables, which are maintained internally in the scheduler. The scheduler is a command called by PLEXIL, as opposed to an external function communicating with PLEXIL over a message bus, which requires Lookups. Finally, the scheduler is invoked explicitly only when new constraints arise from events, be they faults or otherwise. The Start condition in each node ensures that PLEXIL will not start new activities while rescheduling is taking place.

We now assume an arbitrary sequence of unexpected events, including start time delay and increases of duration. However, we assume every activity has *at least* one window, and that no sequence of unexpected events eliminates all windows for an activity. These assumptions certainly do not hold in practice, but let us show how the SMS scheduler and PLEXIL interact without the complications of handing an unsolvable scheduling problem. We revise the PLEXIL as follows:

```
ManageSpacecraft:Concurrence {
  Constraint NewConstraint;
  date A_est=getest(A);
  date A_lst=getlst(A);
  boolean ADone;
```

```
  ...
  boolean Reschedule;
  date now;
  StartActivityA: {
    Start
      (A_est ≤ Lookup(t) ≤ A_lst) &&
      (Other Conditions) &&
      (ScheduleOnEvent.state!=EXECUTING) &&
      (ScheduleOnTimepoint.state!=EXECUTING)
    CmdA: a();
    now =Lookup(t);
    NewConstraint = eq(A_s,now);
    Reschedule=TRUE;
  };
  EndActivityA: {
    Start
      (Lookup(ADone))
    pop(ADone);
    now =Lookup(t);
    NewConstraint = eq(A_e,now);
    Reschedule=TRUE;
  };
  ...
  ScheduleOnTimepoint: {
    Repeat TRUE;
    Start
      (Reschedule==TRUE)
    reschedule(NewConstraint);
    A_est=getest(A);
    A_lst=getlst(A);
    ...
    Reschedule=FALSE;
  };
  ScheduleOnEvent: {
  ...
  }
};
```

The SMS scheduler builds a total ordering of the activities by choosing activity windows, and when necessary, ordering activities within windows. The resulting schedule can be interpreted as a Simple Temporal Network, or STN [1]; the resulting STN can also be made *dispatchable* (Morris, Muscettola, & Tsamardinos 1998) to reduce propagation costs. Let A_{est} denote the propagated earliest start time of A, A_{lst} denote the propagated latest start time of A, A_{eft} denote the propagated earliest finish time of A, and A_{lft} denote the latest finish time of A. The plan uses these bounds in the start conditions, instead of the fixed times A_s and A_e.

When the activity can start late or end late, we must introduce explicit PLEXIL nodes for managing activity start and end. In our previous PLEXIL plan, only one PLEXIL node for each activity is needed; the Start condition of this node only needs the scheduled start time of each activity, because we assumed the activities ended 'on time'. In the new PLEXIL plan the 'other conditions' may not be satisfied at the activity's earliest start time. We assume the command starting the activity, a(), asynchronously starts a process that sets a boolean *ADone* signaling that the activity is completed. This variable is looked up by PLEXIL and used in the condition for node EndActivityA. When activities start or end late, either propagation or rescheduling is needed. We need a new node, ScheduleOnTimepoint, to reschedule once event times are known. This node passes new constraints to the scheduler based on observed start or end times. This node's start condition, the boolean *Reschedule*, is effectively a semaphore set by each activity node, and unset by ScheduleOnTimepoint. The constraints are simple; for example, execution of EndActivityA results in the constraint eq(A_e,*now*) being passed to the scheduler. This node also moves all updated bounds back into the nodes for each activity using the get() interface functions.

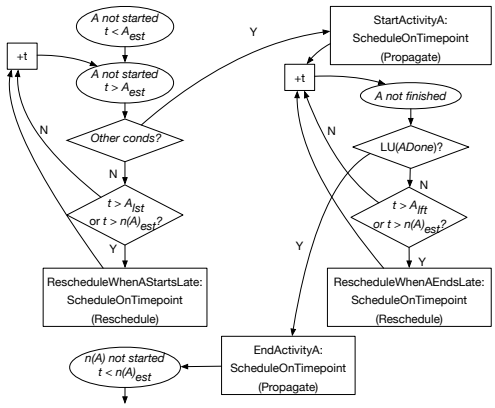


Figure 3: State machine description of PLEXIL plan with all four nodes showing the 'lifecycle' of an activity.

There are still some problems with the PLEXIL as written. Even though we have Lookup(t) $\leq A_{lst}$ as part of the Start condition for StartActivityA, there is no guarantee that A will start before A_{lst}. As written, PLEXIL would not start A if it is delayed past this time. Further, the scheduler will not be invoked to reschedule A if it is delayed past A_{lst}. Worse yet, it is possible that A and A's successor, n(A), suddenly start executing *at the same time*, violating the unary resource constraint. This can happen if $A \prec$ n(A), but both are in a large time window. In order to protect against this, we need to identify *milestones* at which time *proactive* rescheduling is necessary to prevent these undesirable consequences from happening.

The first milestone is A_{lst}, the latest time when A can start without violating some constraint. This bound could be due to the end of A^i (A's assigned window), or due to an STN constraint between A and some future activity. The second milestone is n$(A)_{est}$; if A has not started by this time, there is a risk that A and n(A) could both start. Thus, PLEXIL needs to reschedule when A has not started and $t = \min(A_{lst}$, n$(A)_{est})$. To address this, a new PLEXIL node, RescheduleWhenAStartsLate, is added for each activity; the milestone above is part of this node's Start condition. When executed, this node records the new start time constraint for A,

[1] In fact every timepoint is *uncontrollable* meaning that the schedule is really an STNU (Muscettola, Morris, & Vidal 2001).

then invokes the scheduler. We must add a similar PLEXL node, RescheduleWhenAEndsLate, to cover the case where an activity ends late, i.e. after A_{lft}, which might prompt rescheduling due to an STN constraint with some future activity. Similarly, the activity could end after n$(A)_{est}$, which introduces the danger of n$(A)_{est}$ starting while A is still executing. Both of these nodes invoke the ScheduleOnEvent node, which must retrieve the earliest start time of A's successor, n$(A)_{est}$, for each activity, as well as the bounds on activity start and end.

These new nodes are shown in the PLEXIL below: the behavior of the PLEXIL plan controlling an activity's life cycle is shown in Figure 3.

```
ManageSpacecraft:Concurrence {
  Constraint NewConstraint;
  date A_est=getest(A);
  date A_lst=getlst(A);
  date A_lft=getlft(A);
  date n(A)_est=getsucest(A);
  boolean ADone;
  ...
  date milestone;
  # StartActivityA and EndActivityA
  RescheduleWhenAStartsLate: {
    Repeat TRUE;
    milestone=min(A_lst,n(A)_est);
    Start
      (StartActivityA.state==WAITING) &&
      (Lookup(t) > milestone)
    now =Lookup(t);
    NewConstraint = geq(A_s,now);
    Reschedule=TRUE;
  };
  RescheduleWhenAEndsLate: {
    Repeat TRUE;
    milestone=min(A_lft,n(A)_est);
    Start
      (StartActivityA.state==FINISHED) &&
      (Lookup(t) > milestone)
    now =Lookup(t);
    NewConstraint = geq(A_e,now);
    Reschedule=TRUE;
  };
  # ScheduleOnTimepoint and ScheduleOnEvent
  ...
};
```

Finally, we describe the scheduler in pseudocode. Recall that the scheduler is effectively a function call, reschedule(*NewConstraint*). The parameter *NewConstraint* is updated by the PLEXIL node that invokes it. The scheduler may need to relax duration constraints after event times are fixed to ensure consistency, in addition to relaxing ordering and window commitments. At a minimum, we have to propagate if there are any execution time delays to ensure rigid STN constraints are obeyed by subsequent activities. If activities start or end too late, rescheduling is needed. This is detected in the scheduler and propagation or rescheduling is performed accordingly. Now, StartActivityA and EndActivityA will always have up-to-date bounds if a preceding activity starts or ends after a milestone, because rescheduling will be triggered by RescheduleWhenAStartsLate or RescheduleWhenAEndsLate. Similarly, the unary resource constraint is enforced; two activities will not start at the same time. The scheduler maintains variables A_{est} A_{lst} A_{eft} A_{lft} n$(A)_{est}$ that are retrieved by the PLEXIL get() interface functions.

```
reschedule(NewConstraint) {
VarId var;
Domain val;
...
while (TRUE){
    var=getvar(NewConstraint);
    val=getval(NewConstraint);
    if ((var == A_s) && (A_est ≤ val ≤ A_lst)) {
      propagate();
    }
    else if ((var == A_e) && (A_eft ≤ val ≤ A_lft)) {
      propagate();
    }
    else {
      do-reschedule(NewConstraint);
      propagate();
    }
    # A_est A_lst A_eft A_lft n(A)_est updated
  }
}
```

Rescheduling could be required multiple times when an activity starts or ends late. To see why, consider some activity A starting late with k other activities scheduled to start in its window. The rescheduling operation may move some activity scheduled later than A to a later window, rather than rescheduling A; this can happen at most k times. Similarly, multiple STN constraints on A can conspire to create a succession of A_{lst} bounds that cause rescheduling. This is why RescheduleWhenAStartsLate requires a Repeat condition.

In summary:

- There are now 4 PLEXIL nodes per activity, managing activity start, end, rescheduling when starting late, and rescheduling when ending late.
- The interface between PLEXIL and the scheduler requires retrieving the STN bounds, plus the earliest start time of the successor of the currently executing activity, or the next activity to be executed, as mediated by the get() commands.
- Propagation or re-scheduling is done as needed by evaluating the situation. To ensure activities never execute at the same time, proactive rescheduling steps may be needed, perhaps multiple times, as noted in the previous discussion.

Scheduling and Execution with an Arbitrary Number of Activities

```
ManageSpacecraft:Concurrence {
  Constraint NewConstraint;
  date S_est=getest(S);
  date S_lst=getlst(S);
  date S_lft=getlft(S);
  date n(S)_est=getsucest(S);
  boolean SDone;
  date now;
  ...
  StartUsingSystemS: {
    Repeat TRUE;
    Start
      (S_est ≤ Lookup(t) ≤ S_lst) &&
      (Other Conditions) &&
      (ScheduleOnEvent.state!=EXECUTING) &&
      (ScheduleOnTimepoint.state!=EXECUTING)
    CmdA: s();
    now =Lookup(t);
    NewConstraint = (S_s = now);
    Reschedule=TRUE;
    };
  };
  EndUsingSystemS: {
    Repeat TRUE;
    Start
      (Lookup(SDone))
    pop(SDone);
    now =Lookup(t);
    Reschedule=TRUE;
    };
  };
  ...
};
```

The PLEXIL plan and scheduler integration shown above is limiting because, in general, automated systems like spacecraft require schedules of large size. Large schedules require considerable memory allocations for the scheduler and PLEXIL, which may be precluded on embedded processors. Also, the set of activities may not be known up front; new activities may be needed during the mission. Ideally, we would like a finite sized PLEXIL structure that can execute an *arbitrary* number of activities, allowing the scheduler to schedule new activities when they arrive, and purge completed ones. We assume each activity uses only one resource, but arbitrary temporal constraints are allowed. We can use almost the same PLEXIL structure we created in the previous section. The key changes are to write PLEXIL nodes that manage a *unary resource* instead of an activity, and to change the semantics of the scheduler interface get() functions. Instead of returning the bounds for a fixed activity, these functions now feed this node the bounds for the *next unexecuted activity* using the resource. The revised PLEXIL is shown below; we have replaced the activity name A with resource S in the node names. Since the nodes StartUsingSystemS and EndUsingSystemS now execute an arbitrary number of times, they must have a Repeat condition. We omit repeating the nodes RescheduleWhenSStartsLate, RescheduleWhenSEndsLate, ScheduleOnTimepoint and ScheduleOnEvent, because they are almost identical; the function call is now reschedule(*NewConstraint, S, SDone*) to indicate which resource needs rescheduling. If the resource is free (*SDone*==TRUE), the calling node must set *SDone*=FALSE.

As before, the scheduler waits to hear that activities are completed, or that other constraints have been added. We assume rescheduling places the bounds in queues sorted by activity order. Now, the scheduler need only return the activity bounds for the next unexecuted activity on each resource S, e.g. getest(S)=est.top(). When the scheduler is notified an activity is completed, the appropriate bounds are removed from the queues. We write the scheduler as non-PLEXIL pseudocode:

```
reschedule(NewConstraint, System, Done) {
VarId var;
Domain dom;
Constr constraint;
queue estS;
queue lstS;
queue eftS;
queue lftS;
queue sucestS;
...
while (TRUE){
    #Previously described logic for reschedule vs propagate
    #Scheduler reorders bounds in queues
    if (System == S && Done==TRUE)) {
      estS.pop();
      lstS.pop();
      eftS.pop();
      lftS.pop();
      suceftS.pop();
    }
    ...
}
```

One advantage of this representation compared to the previous one is that the size of the PLEXIL plan shrinks dramatically; we now need 4 PLEXIL nodes per unary resource, as compared to 4 nodes per activity. The new design also reduces the amount of data that must move between the scheduler and PLEXIL. Not only is the amount of data to be moved limited by the number of resources; the scheduler also only must move the *next* activity's propagated bounds, instead of *every* activity's bounds. (To be fair, the activity based PLEXIL plan could be written to move a minimum of data as well, exploiting the dispatchabililty of the STN and completed activities.) This is accomplished at a modest increase in complexity on the scheduler; the scheduler must maintain queues and reimplement the get() accessors.

The PLEXIL plan above can manage multiple unary resources at the same time. Similarly, the PLEXIL representation is able to handle scheduling problems in which tasks can be rejected prior to the start of execution, which in turn, allows handling of activities with priorities.

Discussion

In this section we take a step back and look at the integration of scheduling and execution.

PLEXIL provides direct support for starting the scheduled activities, via its Start conditions. These conditions can refer to variables representing the start times, or bounds, for each activity. PLEXIL also provides commands to invoke the scheduler to change the values of those variables when unexpected events or faults occur. Finally, PLEXIL can detect the conditions when rescheduling is needed, and invoke the scheduler. Designing the PLEXIL to have the right behavior takes some work; however, some of this complexity is driven by the problem (desired behavior when a schedule is disrupted by unexpected events) and not PLEXIL itself. The PLEXIL plan can, however, be written in different ways. For instance, Start conditions for each activity could include a shared Boolean mutual exclusion variable (e.g. semaphore) to prevent two activities from executing at the same time. This design protects against some, but not all, violations of the mutex constraint; it does not facilitate scheduling when activities are late. The specific get() commands must be designed to move propagated time bounds from the scheduler into the right local PLEXIL variables, and the scheduler must manage the bounds. PLEXIL does not provide any direct support for different scheduling algorithms; the scheduler must determine whether propagation is sufficient, or whether a complete rescheduling activity is needed. Put another way, the integration of scheduling and execution requires an up-front investment in determining what information moves between the scheduler and the executive, and the milestones in the execution state of each activity drive propagation or rescheduling. While to some extent this is based on the scheduling problem and the algorithms used to respond to changes, the elementary pieces are the current activity schedule bounds, scheduling milestones, functions to move the information out of the scheduler and into the executive, and functions to invoke the correct rescheduling procedures when milestones are reached.

Previous Work

As noted, the focus of this work is on the integration of scheduling and execution, and not on scheduling algorithms. Clearly the success of an integrated approach requires incremental rescheduling and, eventually, replanning, either directly or by using formalisms such as dynamic constraint satisfaction, will be needed. Given that automated planning has already been demonstrated for applications such as high-speed manufacturing (Ruml, Do, & Fromhertz 2005) and low Earth orbit satellite planning (Tran *et al.* 2004), it is plain that this can be accomplished; however, these previous efforts have largely been engineering exercises.

PLEXIL is one of a number of executives that have been integrated with planners and schedulers. At one end of the spectrum, SMACH (Bohren & Cousins 2010) is a state-based executive (as compared to PLEXIL, which is a command driven executive). Each state contains local variables; states can be hierarchical and concurrent. SMACH states can execute arbitrary Python code, and SMACH has been integrated with Robot Operating System (ROS). While general and flexible, there is little first-class support for planning and execution. TREX (McGann *et al.* 2008) and CASPER (Chien *et al.* 1999) represent the other end of the spectrum. These systems represent plans explicitly as timelines, and planning and execution are processes that interact with the same timeline data structure. In the case of T-REX there may be different specialized planners that perform specific functions that are hand-tuned for performance. CASPER's planner, ASPEN (Fukunaga *et al.* 1997), achieves low computational time through the use of local search. Executing directly off of the timelines maintained by the planners is an advantage; CASPER and TREX do not need to move data from the scheduler into the executive's data structures. However, timelines are heavy-weight data structures, with the implied speed and memory penalties, integrated tightly in the core of the execution system.

T-REX and CASPER represent point engineering designs. More recently, ROSPlan (Cashmore *et al.* 2015) provides a general framework for PDDL-based planners integrated with ROS. ROSPlan allows integration of a wide variety of planning algorithms that operate on PDDL domains with ROS, allowing these planners to control a wide variety of real and simulated robot platforms. The detection of conditions for replanning is semi-formally described, and is similar to the integration described in this paper. However, replanning in ROSPlan still requires re-invocation of a PDDL planner; not all such planners are suitable for embedded processors and high-speed low-latency applications.

PLEXIL has previously been integrated with a PDDL planner to control a hexapod robot (Muñoz, R-Moreno, & Castaño 2010). The specific semantic integration of the PDDL plan and PLEXIL plan is not fully described, making it difficult to compare to ROSPlan. Regardless, integration of PLEXIL with a PDDL planner suffers from the same difficulty we point out for ROSPlan; it may not be suited to embedded processors and high-speed low-latency applications. PLEXIL has also been extended to include decision logic queries, also to control a robot (Moser *et al.* 2009). This increases the flexibility of the knowledge, and therefore behavior, a PLEXIL plan can exhibit during execution, but falls short of the kind of flexibility needed to reschedule actions in the presence of uncertain events.

We have deliberately taken a 'classical' rescheduling and replanning approach, as opposed to an approach based on MDPs or POMDPs, so we do not cover this fascinating area in our previous work survey. We do discuss the Simple Temporal Network with Uncertainty (STNU) as an alternative foundation for rescheduling and execution. The most important property of an STNU is whether it is dynamically controllable, that is, whether there exists a strategy for executing its time-points that will guarantee that all of its constraints will be satisfied no matter how the durations of the uncontrollable durations or event separations turn out. The problem checking whether an STNUs is dynamically controllable was first described in (Muscettola, Morris, & Vidal 2001); the problem of *executing* dynamically controllable STNUs is analyzed in (Hunsberger 2016). Writing PLEXIL for an STNU is a worthwhile next step; using these ap-

proaches require synthesizing the STNU to be executed, e.g. by solving some harder planning or scheduling problem. An alternative classical approach would use Disjunctive Temporal Network (DTN) solvers (Tsamardinos & Pollack 2003) and execution (Shah & Williams 2008). While guarantees such as strong controllability may eliminate the need for rescheduling an STNU, and reduce rescheduling, the presence of resource constraints means rescheduling will still be needed as unexpected events occur.

Philosophically, we compare the integration of planning (well, scheduling) and execution with the recently introduced notions of Planning and Acting (Ghallab, Nau, & Traverso 2016). Viewed broadly, our chosen executive, PLEXIL, may be considered an *actor*. It provides support for handling activity start and stop and invocation of rescheduling in the presence of execution-time uncertainty. Scheduling serves the acting function; the actor, encoded in PLEXIL, is in charge. However, a more hard-nosed attitude is that the philosophical argument is not important, and that it is better to crisply define the problem (if an unexpected event breaks the schedule, what is to be done?) and focus on the solution. This is the approach taken in our work.

Conclusions and Future Work

We have described how to integrate PLEXIL, a plan execution system, and a scheduler. Our motivation is the management of systems in which fast response time precludes using fully functioned, but high complexity, AI planners. We have assumed that PLEXIL manages activities with time windows, temporal constraints, and unary resource constraints; the scheduler, therefore, solves a variant of the Single Machine Scheduling (SMS) problem. Our focus is on formally defining what information passes between the scheduler and the execution system, and when rescheduling is needed. We developed two integrated systems, one assuming a known and fixed number of activities to be scheduled, and a second one assuming a fixed number of unary resources but an arbitrary number of activities.

Implicit in our discussion above is that computation time and resource performance of the integrated scheduler and executive are at a premium. Specifically, even for SMS, rescheduling may end up taking a significant amount of time. Characterization of the computation time (in real wall-clock time terms, not algorithmic complexity analysis) is generally needed to ensure that rescheduling happens in a timely manner. This will vary CPU to CPU, and problem to problem.

As noted above, we reschedule (reorder tasks, move them to other windows) only when temporal constraints are violated. Further, we propagate when possible, and reschedule when needed. Other policies could be implemented to reschedule earlier, reschedule instead of repeating propagation, and heuristics used to guide the rescheduling of tasks more likely to fail and need a second attempt. If, for example, an activity A is starting late, and waiting would push many other activities out of a shared set of windows, it may be better to simply move A later. Similarly, if later windows are not ideal for activities, waiting until the latest start times for activities may lead to worse schedules. On a related note, since computing on spacecraft is often a significant power draw, characterizing the amount of time spent rescheduling is of more than theoretical interest. As noted in previous work, the runtime (and hence power consumption) of executing STNUs has been analyzed. In the very general setting we describe in this paper, it may be much more difficult to analyze the runtime of rescheduling during execution completely. Certain limited cases can be analyzed directly. For example, if A is continually delayed until it must be scheduled in a later window, the number of successor activities k in the current window can be counted, and the amount of rescheduling quantified; similarly, if A is constrained by STN constraints to k successors, each of which imposing a different value of A_{lft} in isolation, the amount of rescheduling in this case can also be quantified. Notably this depends on both the schedule and the set of unexpected events.

In the SMS problem, determining the milestones when rescheduling is needed is straightforward. When we move to more complex scheduling problems with multi-capacity resources, deciding when to reschedule becomes more complex. Consider, for instance, rescheduling activities with power and energy constraints, as described in (May *et al.* 2014). Activities now use and return a multi-capacity resource such as energy. Relaxing the times at which events occur leads to a Resource Temporal Network (Laborie 2003). New algorithms are needed to determine the milestones at which rescheduling must be done if activities do not occur when they are scheduled. Another avenue of investigation must address activities that use *multiple* resources. Arguably, if all resources are unary resources, the proposed integration still holds (windows indicate when all resources are available) but activities using *heterogeneous* resources may pose a challenge.

As noted, rescheduling will generally not be sufficient to address all the complexities of autonomous systems; eventually, we will need planning. Introducing the idea of a scheduler tightly integrated with the execution system leads to an interesting design question for traditional AI planning systems: these planners now only need to make enough decisions to induce a *scheduling problem* to feed to the fast-response scheduler-execution system. This is a different style of planning than the AI community is accustomed to solving.

We have short-circuited the issue of co-development of the PLEXIL plan and representation used in the scheduler. From a theoretical point of view, the key issue is ensuring that PLEXIL activity node representation matches the variables retrieved by the scheduler's get() functions. It should be straightforward to auto-generate the PLEXIL from some higher-order language description that, in turn, configures the scheduler. As the problems get more complex, this is no longer an obvious conclusion, and should be investigated.

Acknowledgements We are grateful for fruitful discussions and contributions of Chuck Fry, Richard Levinson, J Benton, Michael Iatauro, Thomas Stucky, and Paul Morris to this paper. This work was funded by the NASA Advanced Exploration Systems program.

References

Bohren, J., and Cousins, S. 2010. The SMACH high-level executive. In *IEEE Robotics and Automation Magazine*.

Brücker, P. 1998. *Scheduling Algorithms*. Springer.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carreraa, A.; Palomeras, N.; Hurtós, N.; and Carreras, M. 2015. ROSplan: Planning in the Robot Operating System. In *Proceedings of the* 25^{th} *International Conference on Artificial Intelligence Planning and Scheduling (ICAPS)*.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 1999. Integrated planning and execution for autonomous spacecraft. In *Proceedings of the IEEE Aerospace Conference*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–94.

Fukunaga, A.; Rabideau, G.; Chien, S.; and Yan, D. 1997. Toward an application framework for automated planning and scheduling. In *Proceedings of the* 15^{th} *International Joint Conference on Artificial Intelligence*.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Hunsberger, L. 2016. Efficient execution of dynamically controllable simple temporal networks with uncertainty. *Acta Informatica* 53(2):89 – 147.

Laborie, P. 2003. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Artificial Intelligence* 143:151–188.

May, R.; Soeder, J. F.; Beach, R. F.; George, P. J.; Frank, J.; Schwabacher, M. A.; Wang, L.; and Lawler, D. 2014. An architecture to enable autonomous control of spacecraft. In *AIAA Propulsion and Energy Conference*.

McGann, C.; Py, F.; Rajan, K.; Thomas, H.; Henthorn, R.; and McEwen, R. 2008. A deliberative architecture for AUV control. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

Morris, P.; Muscettola, N.; and Tsamardinos, I. 1998. Reformulating temporal plans for efficient execution. In *Proceedings of the* 15^{th} *National Conference on Artificial Intelligence*.

Moser, H.; Reichelt, T.; Oswald, N.; and Förster, S. 2009. PLEXIL-DL: Language and runtime for context-aware robot behaviour. 179 – 186.

Muñoz, P.; R-Moreno, M. D.; and Castaño, B. 2010. Integrating a PDDL-based planner and a PLEXIL-executor into the ptinto robot. In *Proceedings of the 23rd international conference on Industrial engineering and other applications of applied intelligent systems*.

Muscettola, N.; Morris, P.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proceedings of the* 17^{th} *International Joint Conference on Artificial Intelligence*.

Ruml, W.; Do, M. B.; and Fromhertz, M. 2005. On-line planning and scheduling for high speed manufacturing. In *Proceedings of the* 15^{th} *International Conference on Automated Planning and Scheduling*, 30 – 39.

Shah, J., and Williams, B. 2008. Fast dynamic scheduling of disjunctive temporal constraint networks through incremental compilation. In *Proceedings of the* 18^{th} *International Conference on Artificial Intelligence Planning and Scheduling (ICAPS)*.

Tran, D.; Chien, S.; Sherwood, R.; Castaño, R.; Cichy, B.; Davies, A.; and Rabbideau, G. 2004. The autonomous sciencecraft experiment onboard the eo-1 spacecraft. In *Proceedings of the* 19^{th} *National Conference on Artificial Intelligence*, 1040 – 1045.

Tsamardinos, I., and Pollack, M. 2003. Efficient so- lution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence* 151(1-2):43–90.

Verma, V.; Jónsson, A.; Pasareanu, C.; and Iatauro, M. 2006. Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations. In *Proceedings of the AIAA Space Conference*.

Autonomous Search-Detect-Track for Small UAVs

Robert Morris[1], **Anjan Chakrabarty**[1], **Joshua Baculi**[2], **Xavier Bouyssounouse**[1] and **Rusty Hunt**[1]
1 Intelligent Systems Division
NASA Ames Research Center, Moffett Field, CA 94035
2. Department of Mechanical Engineering, Santa Clara University

Abstract

A system is described for autonomously searching, detecting, and tracking an object of interest with a small unmanned aerial vehicle (sUAV). The vehicle is given one or more areas to search. If the object is detected, the sUAV follows the target while maintaining a fixed distance and centered on its image plane. If the object is lost, the sUAV reverts to search. This problem presents many challenges in integrating planning capabilities with sensing and control. This paper describes an architecture for autonomous search and track for sUAVs. The components of the architecture include planning, image classification and image-based control for tracking. The system has been implemented in the Robot Operating System (ROS) framework using the Parrot ARDrone platform, using ROSPlan for goal planning and re-planning.

Introduction

The role of planning in autonomous robotic systems has been extensively documented. Planning frameworks allow for representing actions and goals at multiple levels of abstraction, enabling modularity and hierarchical control. Task planning is needed to transform mission goals into sequences of tasks that will accomplish them and keep the system safe. Task planners must be combined with motion planners, manipulation planners, and sensing and control systems for actuation. In this manner deliberative systems are integrated with reactive systems for dealing with noisy sensors and dynamic environments.

One major challenge in developing autonomous robotic systems is to ensure an effective interaction between the deliberative and reactive sub-systems. Specifically, the deliberative level needs to be always 'doing the right thing', given the information it currently possesses about the state of the robot and the world it inhabits. This is a core capability of autonomy and one that offers challenges in design and development.

The main contribution of this paper is a framework that provides an interface between high level planning and low level sensing and control for solving the problem of autonomously searching, detecting, and tracking (henceforth SD&T) an object of interest by a small UAV (sUAV). To motivate, consider the problem of finding, locating and tracking poachers of rare white rhinos in Africa(SaveTheRhino 2016). sUAVs with night and day sensors are deployed to search over a large area and locate poachers in a timely manner so that rangers can be dispatched to stop poachers before they strike. Some of the technical challenges in sensing, navigation and decision-making for this application include choosing where to search, distinguishing poachers from animals based on movement, and tracking an unfriendly target trying to elude capture. Uncertainty of target location makes it necessary to plan a path that guarantees sufficient coverage of an area while using data such as previous known target location to focus the search, and does not violate resource constraints. At execution time, the vehicle must rely on robust, real time image classification to detect a potential target, as well as rely on an image-based visual controller to keep the target in view. The integrated SD&T system will also involve precise communication of location to operators, a ground control system to enable human supervision and situational awareness of the sUAV activities, and potentially more than one sUAV in a coordinated operation.

The following sections of this paper describe an architecture, component capabilities, and implementation of an integrated autonomous SD&T system for a sUAV, as well as highlighting tests in simulation and in indoor field tests.

Autonomy Architecture and Component Capabilities

An operational SD&T system for a sUAV will consist of a strategic combination of human and machine intelligence. A 'fully' manual SD&T system might consist of a human sitting at a console. The search stage would consist of the human using a joystick to navigate a path for the sUAV through a promising area for search. The vehicle sensors would allow the human to identify the target of interest, and tracking could then also be performed manually.

The focus here is an integrated system where capabilities of search, identify and track are automated. The requirements for *autonomous SD&T* encapsulate a sort of 'set it and forget it' approach: at the most abstract level, the sUAV is provided with a goal to locate an object (candidate rhino poacher) and then track it until some terminating event occurs (the poacher is captured) or the sUAV is close to consuming its power resources and is required to return to base.

There are of course many variations of an SD&T problem, but this is the nominal scenario we will use here.

No assumptions are made here about whether the machine autonomy resides on the sUAV or on a ground computer (we assume here that human autonomy resides fully on the ground). Constraints on platform size and payload weight will to a large extent determine whether the component capability can be on board. It suffices here to assume that to some degree the autonomy architecture is distributed: the capabilities for autonomy reside partly on the vehicle itself and partially on a ground computer.

Figure 1 summarizes the components of the framework for autonomous SD&T described in this paper. A high level task planner alternates between two mission goals: search and track. The search goal triggers a trajectory-based control system combined with a detection system for identifying an object of interest. The track goal triggers a combined image-based visual servoing system (IBVS) and a tracker for following the target. The high level plan dispatcher ensures that the system will continuously alternate between searching as long as the object is not in view and tracking as long as it is.

The figure also shows the components of the implementation of the framework. Building upon previous work by the authors and others, the IBVS and CMT tracker are implemented as modules with ROS (Quigley et al. 2009). ROS provides the middleware for building robotic applications (primarily, in the ability to publish or subscribe to "topics" that provide state information or control lower-level behaviors). A Parrot AR.Drone quadrotor is commanded from a computer via WiFi link using the AR.Drone Autonomy ROS package (Monajjemi 2012), or in simulation using Gazebo (Koenig and Howard 2004). All simulations are run under Ubuntu 14.04 LTS 64-bit and an Intel Xeon E5-2630 @ 2.60 GHz x 17 CPU, a NVIDIA Quadro K5000 GPU, and 32 GB of RAM. Software components include ROSPlan for task planning and execution; the integrated CMT tracker and image-based controller; and the HOG-HAAR detector combined with a simple path planner. These components are described in more detail below.

Previous Work

The relevant literature on the component capabilities required for integrated SD&T is too large to adequately survey here. Rather we briefly summarize the relevant subproblems and key elements of major approaches to solving the problems.

Planning for Search

Planning for search and track (and variants such as search and rescue (SAR)) is one of the oldest problems in Operations Research. The foundations of the theory of search are Koopman's formulation (Koopman 1957) which divides the problem into two sub-problems: optimal allocation of effort (i.e. what percentage of time to spend in a given subregion); and optimal rescue track. From this core formulation models of how a target can move in an environment (e.g. a lost swimmer drifting at sea) and other environmental models, such as weather forecasts, are added, that aid in generating an optimal search path. Search trajectories are evaluated based on maximizing the probability of finding a target of interest at minimum cost (including time, manpower expended, fuel or other resources, etc.).

Planning for search is potentially challenging because it is assumed that the target is located within an area that is too large to search exhaustively; the target's location is represented as a probability distribution over subregions of the search area; and the target may or may not be moving. A typical planning cycle involves the production of a probability distribution for the object's location at the time of the next search. A trajectory uses this distribution along with a list of assigned search assets to produce operationally feasible search plans that maximize the increase in probability of detecting the object. If the search is unsuccessful, a posterior probability map for object location that accounts for the unsuccessful search and the possible motion of the object is generated, providing the basis for planning the next increment of search (Kratzke, Stone, and Frost 2010).

The work that most resembles the over all approach to search presented here is found in (Bernardini, Fox, and Long 2014). The work described in that paper is mostly complimentary to the work here. There the problem to be solved requires a search over the space of possible patterns to find one that has the best chance of finding a target. By contrast, for this paper patterns are provided as inputs to the problem being solved, and the effort here is on applying continuous re-planning to enable hierarchical control.

Vision and Control for Search and Track

SD&T is an application of object detection systems. The goal of object detection is to detect all instances of objects from a known class, such as people, cars or faces in an image. Each detection is reported with some form of pose information; here, we assume the system returns a bounding box around the object.

Object detection systems build a model from a set of training examples. Methods for inferring models are either *generative* or *discriminative*. The former create probabilistic models of pose variability and appearance; the latter create classifiers that can distinguish between images that contain an object from those that do not. Here we use a person detector based on a classification technique using grids of histograms of oriented gradients (HOG) descriptors and a linear Support Vector Machine (SVM) to classify images as person/not person; the system is described in (Dalal and Triggs 2005).

Many visual servo systems, sometimes classified as *dynamic look-and-move* (Campoy et al. 2008), use a hierarchical approach, in which a vision-based controller provides set-point inputs to a lower level position controller. The result is a visual processing loop, comprised of a feature extraction component and a visual controller, and an internal loop, comprised of a state estimator and a flight controller. The feature extractor processes the tracker data and outputs a feature vector that provide feedback to the visual controller, related to the location and size of the object on the image plane. The difference between the feature vector values and

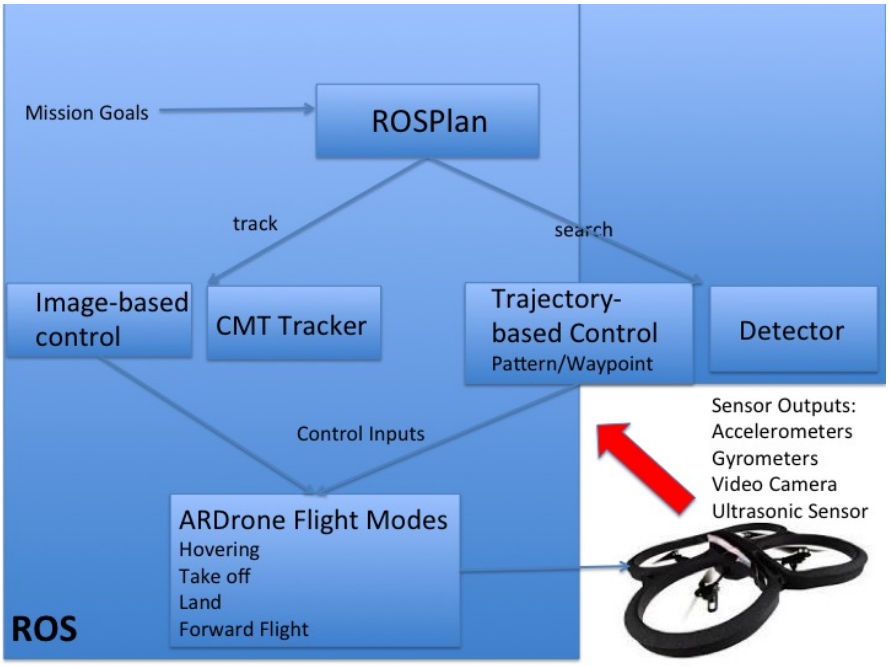


Figure 1: Hierarchical Autonomous Control using ROSPlan

the desired position and size of the features provides the input to the visual controller, which outputs velocity information to the flight controller. Meanwhile, an optical flow state estimation algorithm processes internal sensor data to maintain stability and position. In this architecture, the internal control cycle is on board the UAV, while the visual processing is performed remotely on a laptop, and communicated via a wifi connection.

Tracking is the problem of detecting an object of interest in a field of view of a camera over a period of time. More precisely, the input to the problem is a bounding box defining the set of features of an object of interest. Given a sequence of images, the problem is to identify a set of matches between features defined in the bounding box and those of the current image. These matches constitute the representation of the object of interest.

The CMT tracker (Nebehay and Pflugfelder 2015) used in the SD&T system in this paper, represents a tracked object as a set of feature correspondences of key points, relating the current position of a feature to its position in the original image. Furthermore, CMT distinguishes between static correspondences (between the current image and the original image) and adaptive correspondences (between successive images), and uses both kinds of correspondence to update its object model. Adaptive correspondence is better at handling different object appearances due to deformation, whereas the static model is better at handling the reappearance of the object after occlusion. Second, CMT introduces a tolerance parameter in order to allow the set of correspondences between frames to be robust to changes due to deformations. The CMT tracker uses heuristic estimates to generate values related to scale and rotation of the bounding box. These values, in addition to an estimate of the center of the tracked object, are the outputs of CMT.

For more on the IBVS controller and tracker used in the system described here, see (Chakrabarty et al. 2016) or (Pestana et al. 2014).

SD&T Planning and Execution Using ROSPlan

As stressed in (Nau, Ghallab, and Traverso 2015), carrying out actions in a plan in robotic applications requires continuous on-line planning and deliberation and involves a hierarchically ordered collection of modules to carry out specialized tasks. In SD&T, search, detection, tracking and navigation are specialized behaviors that need to be integrated into an autonomous system that stays safe and accomplishes high-level mission goals.

As noted earlier, planning for search is potentially a computationally challenging optimization problem involving the efficient use of resources to maximize the probability of finding a target of interest. To formulate a search planning problem, the domain model must contain a way to specify one or more subareas to be searched. In addition, the domain should contain a collection of actions that describe a comprehensive search of an areas of interest, as discussed in (Bernardini, Fox, and Long 2015).

In addition to these planning requirements, there are a number of requirements for effective execution of plans. When a target of interest is found, search must stop and tracking started. From a planning perspective, this transition can be viewed as a change in plans, from one consisting of following a trajectory to one consisting of keeping a target in view (tracking). This transition from trajectory-

based to tracker-based control of the vehicle should be done in a timely manner to avoid logins the object. More generally, the system should support an "alternating behavior" between search and track, in which the target is repeatedly lost and found.

Planning and execution using ROSPlan

We use ROSPlan (Cashmore et al. 2015) to automate the planning process for SD&T. ROSPlan supports PDDL activity planning, consisting of a domain defining the actions used for planning, and a problem file that contains a description of the initial state and goal(s) of the plan. ROSPlan supports different planning solvers; for this work we use the temporal planner POPF, that is included in the ROSPlan installation. ROSPlan contains a plan executive that dispatches plans. ROSPlan uses the ROS message passing infrastructure to dispatch actions and receive feedback from the low-level controllers. The PDDL model and problem instance are also stored as a ROS nodes called the Knowledge Base (KB) and the Problem Generation nodes respectively. The KB is continuously updated from sensor data through the ROS interface. ROSPlan validates the current plan against the current model and allows replanning to occur in the case of action failure.

ROSPLan model for SD&T

The PDDL model developed for SD&T contains predicates *found* and *trackdone* which correspond to the goals of the stages of the activity. These goals are effects of actions *search* and *track*. Intuitively, the goal *found* is true if a target is found, the area of search has been exhausted, or some other condition holds (like running out of battery charge). The precondition for search is *airborne*, and the effect is *found*; conversely, the precondition for track is *found* and the effect is *trackdone*. Following previous formulations of the problem, the search phase is interleaved with trajectory planning whereby the drone can be dispatched to a a region based on predictive models of where the target might be. This phase is accomplished through a *goto-waypoint* action. (The difference between executing a search action and executing a goto-waypoint action is that the latter does not assume the detection system is turned on.)

PDDL actions map directly to ROSPlan action components that refine the actions into low-level commands. We use a procedure-based approach to action refinement, with hand-written procedures written in C++, reminiscent of systems for reactive planning such as RAP (Firby 1987). The PDDL domain actions map to the following two methods:

procedure SEARCH(path,object)
  while object not detected and battery OK **do**
    follow path
  Stop and hover
  Add found fact to KB
  if battery not OK **then**
    Add home waypoint goal to KB
  else // object is found
    Add tracked goal to KB
    Re-Plan to track

procedure TRACK(object)
  Start tracker
  Start IBVS controller
  while confidence $>$ threshold and battery OK **do**
    follow object
  Add trackdone fact to KB
  Stop IBVS controller
  if battery not OK **then**
    Add home waypoint goal to KB
  else // object is lost
    Add found goal to KB
    Re-Plan to search

We distinguish among 4 types of low-level actions that collectively implement a refinement of *search* and *track*:

- Accessing state information: the conditions on the while loop access state information (via the implementation of publishers or subscribers to ROS Topics) to determine whether the object has been found or lost and whether power resources are sufficient to continue the mission. Specifically, two state variables *PersonDetected*, set by the detector, and *Confidence*, set by the tracker, are monitored by the action components. If *PersonDetected* is true and *Confidence* is above a certain threshold, then the system is in a tracking state; otherwise, the system is searching.
- Initializing sub-systems for sensing or control: starting and stopping the tracker and IBVS controller.
- Drive the sUAV: following a path constitutes a high level trajectory planner and controller
- Re-planning: If the object is found then re-planning for tracking is started by adding and removing goals. The Track method is complementary.

In this way, manipulation of the knowledge base by adding or removing goals enables continuous alternating behavior required by SD&T.

Experiments

In this section we summarize the experiments conducted on the ROSPlan-based SD&T system described in this paper. The tests here are meant to verify the feasibility of the ROSPlan approach to continuous planning for SD&T. The performance metrics for feasibility include correctness (the planner always responds properly to the presence or absence of the target in its image plane by placing the system into a search or track behavior mode), and timeliness (ROSPlan responds to a change in the sUAV's sensor outputs quickly enough to ensure continuous, stable operations).

Experiments in simulation using Gazebo (Figure 2) and in an indoor testing facility at Ames Research Center (Figure 3) were conducted. The simulation allowed for quick debugging and adjustment of system parameters that influence performance. Some of these parameters include the expected size and distance of the target (used by the IBVS controller) and the sampling rate of the state variables within the ROSPlan action interface code.

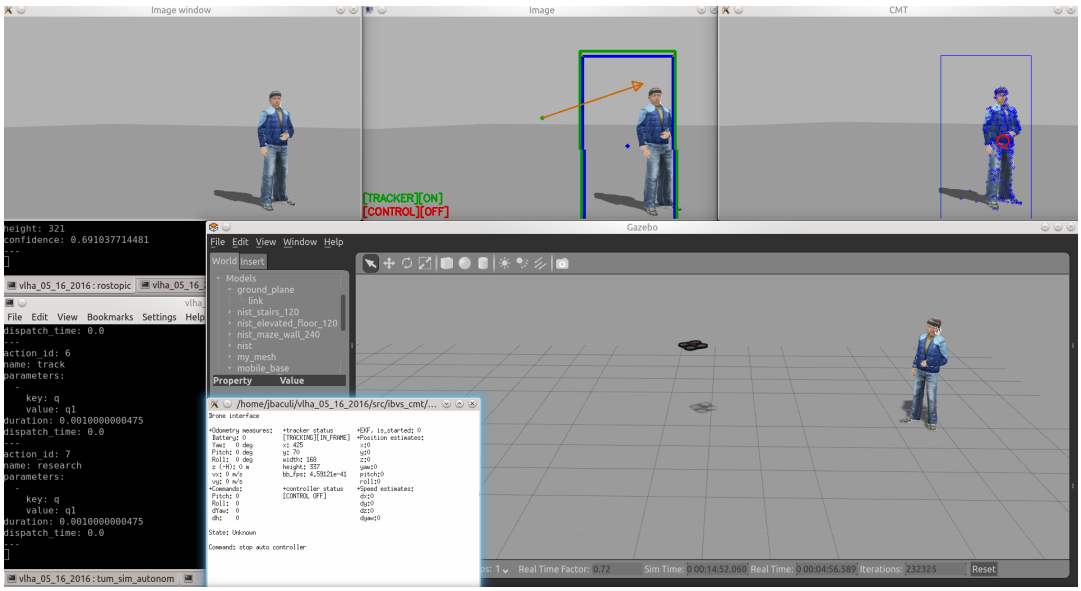


Figure 2: Simple Gazebo environment for testing SD&T in simulation. The upper middle panel shows the bounding box around the person target maintained by the CMT tracker, and control input (orange arrow) to visual controller. The right panel shows the bounding box constructed by the HOG detector. The black windows on the left stream the outputs of the relevant ROS topics. The larger panel shows the sUAV and target.

Figure 3: Testing ROSPlan for SD&T in an indoor facility at NASA Ames Research Center.

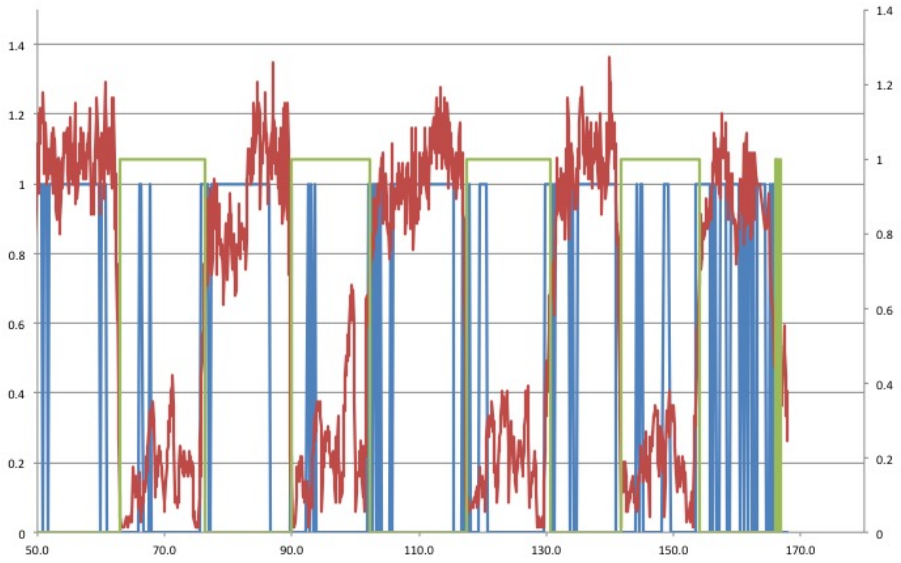


Figure 4: Interactions between ROSPlan Action Dispatching and Sensing Outputs. ROSPlan Action Dispatch Variable (Green Line) alternates between posting a search goal (1 on right y axis) or a track goal (0) on the ROSPlan KB, based on values obtained by the tracker (Confidence, red line) and the detector (PersonDetected, blue line, where 1 (left y axis) = PersonDetected true).

We deliberately chose a fairly controlled indoor environment in order to isolate the interactions between the continuous planning framework and the underlying sensing and control frameworks, thereby demonstrating the feasibility of a hierarchical autonomy architecture based on ROSPlan. Consequently, however, there is little in the way of assurance that the system is robust to realistic dynamic environments or noisy sensors. Future outdoor field tests are planned, using more powerful platforms than the AR Drone, in order to improve robustness.

In a typical run of the system, a human target would begin out of sight of the sUAV. The sUAV would take off and begin a pattern maneuver (such as a square pattern) in search mode. Once the human target is found, we studied the response time of the system to transition into track mode. Once in track mode, the target would first move in a slow linear walk to demonstrate simple tracking behavior. At some point, the target would take evasive maneuvers until he was out of sight of the sUAV. We then could observe the transition back into search mode, which consisted of following another pattern search (typically a simple rotation). This alternating behavior of search and track was typically repeated many times in a single run.

Figure 4 illustrates the run time performance in the indoor facility, illustrating the alternating search-track behavior induced by ROSPlan. In the figure, changes in the state of three ROS topic variables over time (in seconds) are displayed: *confidence*, published by the tracker, *person detected*, published by the detector, and *action dispatch*, used within ROSPlan. The action dispatch variable is represented as a step function with two values: 0 (for track) and 1 (for search); depending on the value of this variable, one or the other of the procedures described above are executing.

In the execution snippet in the figure, the sUAV starts in track mode (ActionDispatch=0, green line). Recall that ROSPlan triggers a new plan for tracking by adding a goal to the KB, while invoking the CMT tracker and IBVS controller. The CMT tracker inherits the bounding box generated by the the detector, and initializes the *confidence* state variable, which it updates during tracking. If the confidence variable goes below a certain threshold (0.6 in the experiments here) ROSPlan concludes that the object has been lost, and a new search goal is placed on the KB. For example, at roughly the 62nd second of this run, the ActionDispatch variable is set to 1 as a response to the *confidence* variable reaching its threshold.

To aid in robustness and stable behavior, we found through experiments that it is best to re-trigger tracking when *both* the detector finds the human figure and the confidence variable exceeds the desired threshold. For example, roughly around the 95th second, the tracker has a confidence value greater than the threshold, but the detector has not found a target. The tracker has in fact falsely identified a target to track in the image plane at this point. The plan dispatcher, however remains in search mode, waiting for the person detector to register the target. Conversely, there are

times (such as around second 135) in which the detector shows uncertainty between target and no target, and yet the tracker shows confidence in its target. In this case we allow the tracker to direct the plan dispatcher.

To summarize, the experiments conducted validated the feasibility of using ROSPlan as the basis for a hierarchical architecture for autonomous control. We were able through successive refinements of operational parameters to achieve stable and correct behavior in a controlled environment using an AR Drone. SD&T is a difficult challenge for autonomy because of the need for high-level management and control of subsystems that are themselves comprised of complex sense-control loops. We showed that a high-level decision-maker based on continuous planning fulfills the requirements for an effective system for SD&T.

Future Work

We are exploring improvements to the autonomous SD&T framework in parallel on a number of fronts. On the platform side, we are in the process with replacing the ARDrone with higher-cost platforms with better flight stability and more advanced sensing units. This would allow us to incorporate different sensors that would allow searching in a wider range of realistic outdoor environments. In turn, this would allow us to pursue a more phased approach to search, whereby, say, a motion detector is used to find a moving target, which triggers the vehicle to move towards the object in order to identify it.

Secondly, we are in general interested in more rigorous testing in more realistic environments, whether actual or in simulation. On the simulation side, we are exploring the use of the Modular OpenRobots Simulation Engine (MORSE) combined with the Blender Game Engine for more realistic environments. An outdoor testing facility at Ames will also be used. In general, more scenarios for demonstrating autonomy in more realistic conditions are needed.

Finally, on the planning side, we are interested in expanding the current model and planning system in a number of ways. First, we are interested in incorporating various optimization criteria into planning, following traditional methods for solving Search-and-rescue problems. Secondly, we would like to devise ways to handle some of the uncertainty at planning time, for example, through contingent planning, which is supported within ROSPlan.

Conclusion

An approach was presented for combining simple task planning with sensing, motion planning and reactive control to achieve autonomous SD&T. The developed system is based on ROS and the ROSPlan planning framework, integrating state-of-the-art algorithms for object recognition and object tracking. The ability to coordinate complex components to enable continuous search and tracking using a sUAV illustrates the challenges and rewards of autonomy. In both simulation and field experiments using the AR Drone the ability of deliberative and reactive systems to work together to achieve high-level goals was demonstrated.

Analogous to the way toy problems like Blocks World provided testbeds for the development of search algorithms which could solve real world problems, ROS and the AR Drone together provide a toy testbed for designing architectures for robotic autonomy that have the potential for solving real world problems like SD&T. In addition, as shown here, ROSplan adds further infrastructure for enabling complex behaviors through hierarchical control.

Acknowledgements

The authors thank Michael Cashmore, Sara Bernardini, Jindrich Vodrazk and Jesus Pestana for helpful discussions. Also thanks to the reviewers to helping identify gaps in the discussion in the original draft.

References

Bernardini, S.; Fox, M.; and Long, D. 2014. Planning the Behaviour of Low-Cost Quadcopters for Surveillance Missions. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS-14)*.

Bernardini, S.; Fox, M.; and Long, D. 2015. Combining temporal planning with probabilistic reasoning for autonomous surveillance missions. *Autonomous Robots* 1–29.

Campoy, P.; Correa, J. F.; Mondragón, I.; Martínez, C.; Olivares, M.; Mejías, L.; and Artieda, J. 2008. Computer vision onboard uavs for civilian tasks. In *Unmanned Aircraft Systems*. Springer. 105–135.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Huros, N.; and Carreras, M. 2015. Rosplan: Planning in the robot operating system. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, 333–341. ICAPS.

Chakrabarty, A.; Morris, R.; Bouyssounouse, X.; and Hunt, R. 2016. Autonomous indoor object tracking with the parrot ar. drone. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, 25–30. IEEE.

Dalal, N., and Triggs, B. 2005. Histograms of oriented gradients for human detection. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1.

Firby, R. 1987. An investigation into reactive planning in complex domains. *Proceedings of AAAI*.

Koenig, N., and Howard, A. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2149–2154.

Koopman, B. 1957. The theory of search, part iii: the optimum distribution of searching effort. *Operations Research* 5:613626.

Kratzke, T. M.; Stone, L.; and Frost, J. R. 2010. Search and rescue optimal planning system. *Proceedings of the 13th international conference on information fusion*.

Monajjemi, M. 2012. Ardrone autonomy : A ros driver for ardrone 1.0 & 2.0.

Nau, D.; Ghallab, M.; and Traverso, P. 2015. Blended planning and acting: Preliminary approach, research challenges. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Nebehay, G., and Pflugfelder, R. 2015. Clustering of static-adaptive correspondences for deformable object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2784–2791.

Pestana, J.; Sanchez-Lopez, J. L.; Saripalli, S.; and Campoy, P. 2014. Computer vision based general object following for gps-denied multirotor unmanned vehicles. In *American Control Conference (ACC)*.

Quigley, M.; Gerkey, B.; Conley, K.; Faust, J.; Foote, T.; Leibs, J.; Berger, E.; Wheeler, R.; and Ng, A. 2009. Ros : an open-source robot operating system. In *EEE International Conference on Robotics and Automation (ICRA 2009)*.

SaveTheRhino. 2016. The use of drones in rhino conservation. *https://www.savetherhino.org*.

An Architecture for Integrated Timeline Planning and Model-based Execution

Tiago Nogueira*, **Simone Fratini**
European Space Agency, ESA/ESOC
Darmstadt, Germany
name.lastname@esa.int

Abstract

Integration of planning and execution, or acting, is a crucial issue in practical applications. This paper presents a system that integrates timeline-based planning with a model-based executive. The executive exploits both the plan's flexibility and the structure of the planner's model to robustify the execution, reducing the need for continuous interaction with the planner. The integration extends also the planner's modeling language to entail the automatic generation, directly from the planning model, of the controllers used at execution time. The possibilities of the approach in terms of modeling, planning and execution are exemplified in a logistic scenario where a UAV has to be operated to move objects in a warehouse.

Introduction

Research and deployment for architectures to support autonomy and automation for space missions has increased mainly to support new generation of missions for Earth observation, space station operations and planetary robotic exploration. In fact, technology developments coupled with more ambitious missions led to the need for autonomy capabilities for operations of spacecrafts and rovers. The reasons for autonomy vary across missions, but main factors that contributed to this escalation are technical and organizational: communication delays and high environmental uncertainty for deep space missions, operational costs reduction and need of increasing missions' scientific return.

Design and implementation of such advanced systems to support autonomy is an activity that involves a certain amount of developing effort and risk. For this reason space has been often a fertile field for the introduction of novel AI based planning and scheduling technologies. In fact, the AI model-based approach allows reusing of software modules across different missions because of the great flexibility introduced by the symbolic representation of goals, constraints, logic and parameters to be optimized, for example. This makes the software deployment and test substantially independent from the specific mission, reducing costs and risks.

*This work has been co-funded by the European Space Agency Networking/Partnering Initiative (NPI) between ESA-ESOC and the Center for Telematics (Zentrum für Telematik e.V.), and by the European Research Council (ERC) Advanced Grant "NETSAT" under the Grant Agreement No. 320377.

As an effect, significant efforts have been so far dedicated to build software development environments for rapid prototyping, test and synthesis of new planning and scheduling applications at NASA (EUROPA (EUROPA 2008), ASPEN (Chien et al. 2000)). The European Space Agency (ESA) concurs in this area of advanced research by promoting the development of APSI (Advanced Planning and Scheduling Initiative) and APSI-related activities (Cesta et al. 2011), its use for on-board autonomy with the Goal Oriented Autonomous Controller (GOAC) (Ceballos et al. 2011) and its application on teleoperations (Fratini et al. 2013). Regarding practical applications of autonomy, the flagship missions for planning and execution technologies are still nowadays the Remote Agent Experiment (RAX) on Deep Space 1 (DS-1) (Muscettola et al. 1998) and the Autonomous Sciencecraft Experiment (ASE) on Earth Observing 1 (EO-1) (Chien et al. 2005). These two missions have pioneered and proven the value of AI planning and scheduling for injecting autonomy in space applications. Lately, NASA has launched the IPEX CubeSat to validate new technologies for on-board image processing and autonomous operations (Chien et al. 2016).

The maturity and domain independence of the planning and scheduling engines, as well as the clear distinction made between decision making and execution, a need driven by the differences between the heterogeneous scenarios to be addressed, from satellites to robotics, suggests the possible reusability of these technologies out of the specific scope for which they have been originally designed.

In this paper we present an integration of a model-based executive into the ESA APSI planning platform. We have extended the planner's modeling language to define directly into the planner's model the rules and procedures to apply for monitoring and controlling the plan execution on the target system. This model is then used both by the planner and by the executive, that exploits the knowledge in the plan's domain theory to cope, to some extent, with unexpected behavior at runtime. As an example of applicability (out of the space segment) we use a test domain and a simulated execution environment inspired by a scenario where a UAV has to be operated to move objects in a warehouse.

Planning Technology

The ESA APSI platform is a Java architecture for rapid prototyping of planning and scheduling applications. The plat-

form is designed for constraint-based temporal planning and scheduling. Constraint-based temporal planning, often referred to as "timeline-based planning", is an approach to temporal planning which has been applied to the solution of several space planning problems – e.g., (Muscettola 1994; Jonsson et al. 2000; Smith, Frank, and Jonsson 2000; Frank and Jonsson 2003; Chien et al. 2010; Cesta et al. 2011). This approach pursues the general idea that planning and scheduling for controlling complex physical systems consist in the synthesis of a set of desired temporal behaviors, named *timelines*, for system features that vary over time. In this approach, problem solving consists of controlling components by means of external inputs in order to achieve a desired behavior. Hence different types of problems (e.g., planning, scheduling, execution or more specific tasks) can be modeled by identifying a set of inputs and relations among them that, together with the model of the components and a given initial set of possible temporal evolutions, will lead to a set of final behaviors which satisfy the requested properties; for instance, feasible sequences of states or feasible resource consumption[1]. The APSI Framework provides a Domain Definition Language (DDL) and a Problem Definition Language (PDL) based on two classes of modeling primitives: state variables and resources. These components and their possible evolutions are then "connected" by means of temporal and logical synchronizations.

State Variables State variables represent components that can take sequences of symbolic states subject to various (possibly temporal) transition constraints. This primitive allows the definition of *timed automata* as the one represented in Figure 1. Here the automaton represents the constraints that specify the allowed logical and temporal transitions of a timeline. A timeline for a state variable is valid if it represents a *timed word* accepted by the automaton. In the example in Figure 1, `P(?x)`, `Q(?y)` and `R(?z)` are the set of possible symbolic states. Transition between the states is subject to value constraints (`?x>?z`) or temporal constrains (`@t<2`), requiring in the latter that the transition from `R(?z)` to `P(?x)` is allowed only if `R(?z)` has been maintained for less than 2 time units

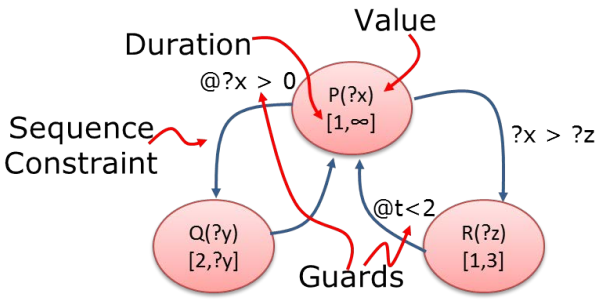


Figure 1: State variable.

[1]A detailed description of the planning approach, state of the art and basic concepts is out of the scope of this paper. More information can be found, for example, in (Muscettola 1994; Frank and Jonsson 2003; Fratini and Cesta 2012).

Resources The second APSI primitive is the resource. This can be used to model any physical or virtual entity of limited availability, such that its timeline (or profile) represents its availability over time whereas a decision on the resource models a quantitative use/production/consumption of the resource over a time interval. Three types of resources are currently available in the APSI Framework: *reusable* resources abstract any real subsystem with a limited capacity, where an *activity* uses a quantity of resource during a limited interval and then releases it at the end. *Consumable* resources abstract any subsystem with a minimum and a maximum capacity, where *consumptions* and *productions* consume and restore a quantity of the resource in specific time instants. *Reservoir* resources do not have a stepwise constant profile of consumption like reusable and consumable ones, but the activities specify the amount of production and consumption per time, namely *slope*, resulting in a profile of resource that is linear in time. As a consequence the amount of resource available at each transition of the timeline depends on the duration of the time intervals over which this production or consumption has been performed. This distinguishes reservoir from the other type of resources where the profile of the resource availability at each transition depends only on when and how much is produced/consumed and not on the duration of the production/consumption.

Synchronizations In timeline-based modeling the physical and technical constraints that influence the interaction between subsystems (modeled either as state variables or resources) are represented by means of temporal and logical synchronizations among the values taken by the automata and/or resource allocations on the timelines. Languages for timeline-based planning have constructs, called *synchronization* in DDL, to represent the interaction among the different timelines that model the domain. Conceptually these constructs define valid schema of values allowed on timelines and link the values of the timelines with resource allocations. Despite the syntactic differences, they allow the definition of Allen's relations (Allen 1983) like quantitative temporal relations among time points and time intervals as well as constraints on the parameters of the related values.

Problem Solving An application in the APSI framework, being a generic planner and/or scheduler or a domain specific deployed application is designed as a collection of *solvers*. Based on the constraint-based paradigm, the search space of an APSI solver is made of planning and scheduling statements on timelines and temporal and data relations among them. Available solvers and applications include state-of-the-art binary and multi-capacity schedulers as well as integrated planners and schedulers like PLASMA (PLAn Space Multi-solver Application) (Fratini et al. 2015).

PLASMA is a planner designed as a collection of solvers implementing a *flaw-based* solving process. Solvers are chosen and activated on a flaw detection base. When a flaw is detected on a timeline the planner activates the corresponding solver to fix the problem. A flaw is any type of violation in a plan. It can be a logical flaw, when unsupported actions are added to the plan, or a resource violation flaw, when a resource is over or under used, or any other impair-

ment of temporal allocation on the values over a timeline. PLASMA incorporates the principles of Partial Order Planning (POP, (Weld 1994)), like plan-space search and the least commitment approach. Starting from an initial partial plan only made of partially specified timelines and goals to be achieved, the planner iteratively refines it into a final plan that is compliant with all the requirements expressed by the goals.

The set of solvers currently available in PLASMA include: (a) a Partial Order Scheduler (POS), supporting the scheduling process resulting from planning to guarantee temporal flexibility[2]; (b) a resource activity generator that makes sure that linear resources can be adequately managed by avoiding any over-consumption; (c) a MaxFlow resource profile bounder whose task is to bound position and duration of activities to assure that all the resource constraints and/or requirements are consistent; (d) an FF-type (Hoffman and Nebel 2001) PDDL planner (JAVAFF (Pattison 2017)) to solve agent-centric combinatorial task allocation problems and (e) a path planner based on the RRT algorithm (LaValle 1998).

A plan produced by PLASMA is composed of a set of *flexible* timelines. A flexible timeline is defined by means of a sequence of values occurring at ordered transition points. Both the transition points and associated values are not grounded, but only bounded by the planning process. Flexible plans represent then an envelope of possible plans instead of a single completely specified plan. This flexibility can be exploited by an executive system for robust on-line execution.

Integrated Planner and Executive

Planning encompasses the capabilities required to expand high-level mission goals into sequence of actions (a plan) while respecting system and environment constraints. Execution, on the other hand, deals with the problem of putting a plan into practice while guaranteeing real-time response.

In space applications these two processes are typically separated into two different applications (or engines). Such split is usually driven by the need to run the executive independently from the planner to execute commands directly issued from ground, or oversee failure detection, identification and recovery activities. Also, the planner and executive are, most of the times, developed by separate groups and rely on different modeling and implementation technologies.

If this separation on the one hand simplifies the design and increases the independence of the executive from the planning technology, on the other hand it can pose problems of controllability when not all the planned event are under the control of the executive (Vidal and Fargier 1999). In fact, integration of planning and execution engines is still nowadays a crucial issue in practical applications.

In our proposal we exploit the structure of the planner's model to derive the architecture of an executive based on the distinction between *system* and *component* controllers. Figure 2 depicts the integrated planning and execution model where a batch planner receives goals as input and produces time-flexible plans. An executive, composed of a system controller, component controllers and observers, ingests flexible plans and issues low-level commands and monitors the components on the target system so as to execute the plan. To close the loop, the planner interfaces with the executive to monitor the execution status and to derive the current system state as input to the planning process.

The rationale behind this architecture is given by the properties of the timeline-based model. Being the system modeled as a set of subsystems evolving in time as parallel threads, with internal evolution constraints and external occasional synchronizations with each other, the model itself identifies the control structure. Each subsystem is operated by an independent controller that knows only its private control law. A system controller, then, *supervises* the individual subsystem (or component) controllers by monitoring and restricting their behavior such as to enforce the plan synchronization points. In the next sections we describe in more detail the control approach, architecture and implementation.

Model-based Executive

The flexible plan is executed in a dynamic environment with hard real-time constraints and some level of uncertainty. Uncertainty at execution time can usually be attributed to either: (a) incomplete domain modeling, when for example an activity takes longer or a resource is consumed faster than anticipated or (b) unexpected changes to the domain, including for example failures in the system and unexpected changes to the environment in which the system operates. As planning, and in particular batch planning as used in our system, is a computationally expensive process, the executive should try to explore the inherent flexibility of the plan to minimize the need for replanning.

Robustness and effectiveness are two typical properties desired in an executive (Tsamardinos, Muscettola, and Morris 1998). Robustness reflects the ability to manage uncertainty such as to control the plan to the desired state in the presence of perturbations. A perturbation, in this context, is any deviation to the initially modeled domain or problem. To be robust a plan must be flexible, in that it must encompass a set of possible alternative behaviors. This is provided by the plan's inherent time flexibility. To be effective an executive should only have to process constraints in the vicinity of the current execution step. That is, we don't want the executive to have to propagate, at runtime, over the full set of constraints to decide which action to take next, as this could introduce considerable delays in execution.

As depicted in Figure 2, we distribute the control responsibility among two types of controllers:

- System Controller (SC), that controls the state of the system, by enforcing inter-component constraints and ensuring that state transitions are triggered at the correct time.
- Component Controller (CC), that controls the state of a component (or subsystem), by guiding its behavior to match the one of the corresponding timeline.

Observers provide an estimate of the state of the system or of a given component from acquired telemetry.

[2]The POS (Policella et al. 2007) is a set of activities partially ordered such that any possible complete order that is consistent with the initial partial order is a resource and time feasible schedule.

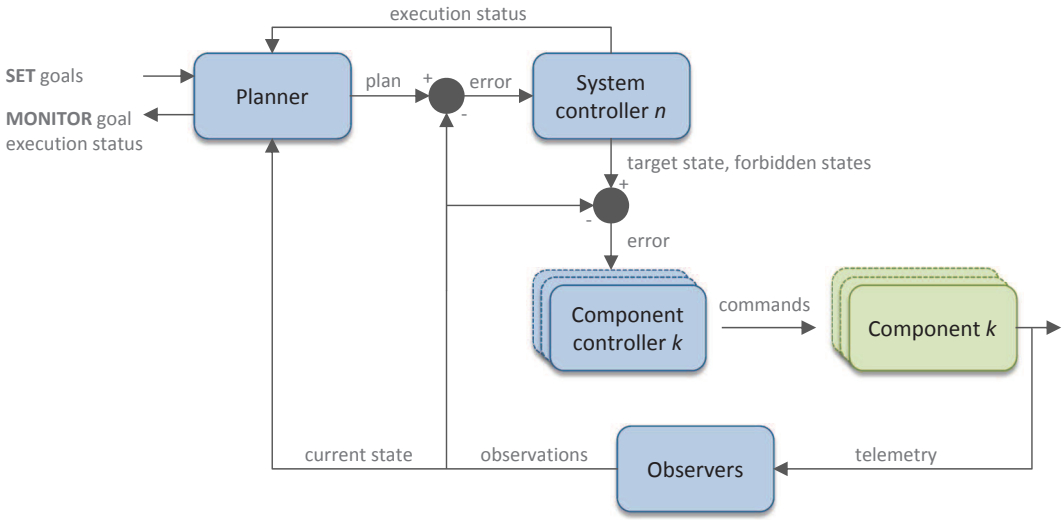


Figure 2: Goal-based integrated planning and execution model.

Splitting the concerns between system and component-level controllers rather than following a centralized approach is mainly driven by two factors. First, the need to simplify the implementation and reduce the memory footprint. Second, the possibilities such an approach offers to distribute the actual deployment and execution of the individual component controllers across various subsystems. This split, however, introduces a problem. Full knowledge about the domain is only held by the system controller. Knowledge within a component controller is local, in that it is limited to the component it controls. This implies that we need to define clear boundaries within which the component controller can operate, such as to guarantee that the decisions it takes do not interfere with the other timelines being controlled. To guarantee that the plan remains sound we must make sure that the component controller can only cross states that are either not synchronized at all to any other states of the timelines being controlled or, if synchronized, are only synchronized to a state currently being controlled for. In other words, the component controller cannot cross states that would move the other timelines away from their target states. This is addressed by, at each transition and for each component controller, introducing a set of forbidden states that cannot be traversed by the local component controller. This set of forbidden states is automatically derived from the synchronizations in the planning domain model and added to the logic of the system controllers. For a detailed definition of the system and component controller algorithms refer to (Nogueira, Fratini, and Schilling 2017).

System Controller The SC is responsible for monitoring and controlling the state of the whole system by guaranteeing that all timelines reach and maintain the required state while respecting the planning constraints. The SC continuously monitors the state of all timelines and triggers the corresponding CC if the state of the timeline deviates from the desired. The SC is the guardian of time flexibility. Rather than instantiating a plan as a sequence of fixed-time actions or commands, the planning temporal constraints are kept at execution time. These constraints are encoded as a Simple Temporal Network (STN) which is used by the SC to lookup and propagate temporal constraints. By propagating the STN the controller can infer the current validity of the plan and abort its execution in case it becomes invalid. If this happens the SC will then trigger the planner to produce a new plan.

Component Controller A CC is responsible for monitoring and controlling the behavior of a particular component type, as defined by the corresponding timeline. Several timelines can share the same controller. A CC can be defined as either a Weighted Finite State Automaton (WFSA) or a user-defined script. The states modeled by a component controller are a superset of the states allowed by the corresponding component model used by the planner. Extra states can be added such as to monitor and control failure scenarios and unexpected states, making the state of the component fully observable. This inherently provides the capability to model failure identification and recovery mechanisms together with nominal behavior, allowing thus to embed part of the Failure Detection Isolation and Recovery (FDIR) logic directly in the executive layer. The option to specify a controller using a user-defined script is there to allow to model controllers whose behavior cannot be encoded as a WFSA. This is typically the case for many domain-specific controllers like an attitude controller in a satellite, for example. By encoding the behavior of the components as WFSA that extend the planning state variable primitives, we are able to bring into the executive the component-level planning constraints used at planning time. This makes it possible for the controller to, at runtime, choose between alternative paths between any two states (as long as this is allowed by the model). Once a CC is given a target value (i.e. state) by the SC, the controller will try to achieve the target state using all the available nominal and recovery paths modeled in its WFSA. If, and after exhausting all allowed attempts, the current state still deviates from the target one the CC returns an error. If

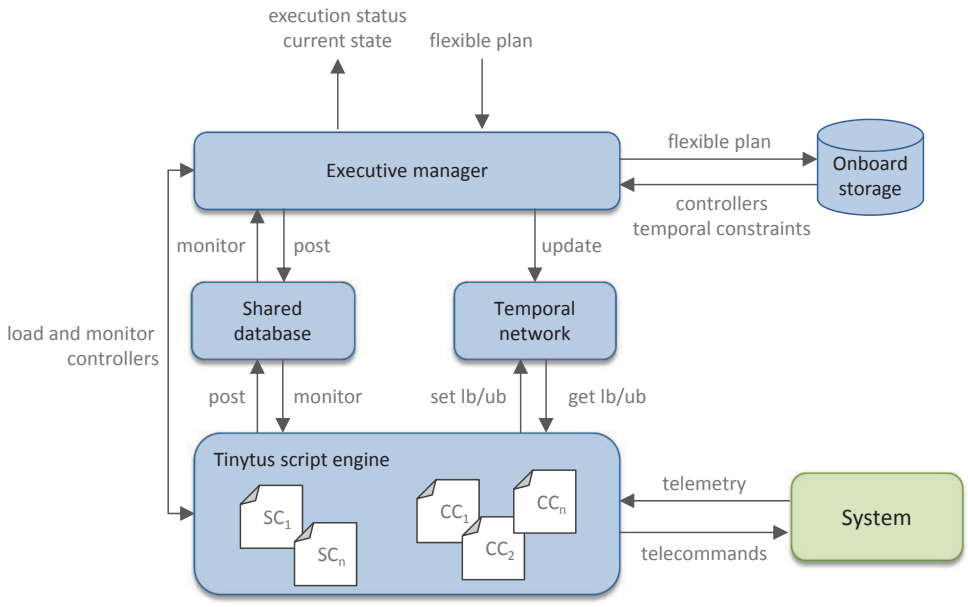


Figure 3: Tiny Executive (TEX) main components and interfaces.

the timeline value being controlled to by this CC temporally or causally constraints other timelines, then the SC triggers a replan, as the plan has become invalid. If, on the other hand, the timeline is completely independent then the SC will continue with the execution. Note that it is up to the modeler, and when designing a particular controller, to guarantee that in case of component-level failure the component will be placed in a safe configuration.

TEX Architecture and Implementation

The Tiny Executive, or TEX, is an implementation of the control model described in the previous section. Our target deployment platform is the 16 bit MSP430 microcontroller, with very limited memory and processing power. Figure 3 depicts TEX's main components and interfaces.

Executive Manager The executive manager oversees the overall execution, interfaces with the planner to receive new plans and report the execution status, and interfaces with the onboard storage to store newly received plans and to load controllers as required.

Shared Database A database is used to share runtime information among all running controllers and the executive manager. Such information includes the current plan origin and horizon, the current execution elapsed time, the current active transition and the status of the current system state (achieved / not achieved). The database is also used to post and monitor event messages, if any, that need to be passed between controllers.

Tinytus Script Engine The system and component controllers run in a sandbox environment, the Tinytus script engine, and interact with the underlying system by issuing commands and monitoring telemetry. Tinytus is a script language and interpreter for embedded systems that provides an onboard sandbox environment for safe software execution (Dombrovski and Bangert 2015). The Tinytus script language uses Polish prefix notation and offers the basic constructs and expressiveness typical of imperative programming language. This includes: (a) arithmetic and logic operations; (b) flow control primitives; (c) declaration, access and casting of numerical variables and arrays; (d) function calls.

Temporal Network The temporal network holds the plan temporal information. The controllers use the network to retrieve the lower and upper temporal bounds for their transitions, and to update the temporal network with the actual transition time once it occurs. The temporal information is encoded as a Simple Temporal Network (STN) formulated as a distance graph. The vertices correspond to the (temporal) events and the edges to the temporal constraints between events. The edges are labeled with the lower and upper temporal bounds for the duration constraint between events (Dechter 1991). At runtime the network must be checked for consistency to ensure the plan's validity. Once a transition is confirmed, the corresponding lower and upper bounds in the STN are updated with the actual execution time. The STN is then propagated to adjust the temporal constraints of future events accordingly, while maintaining a consistent network. If, after a propagation, the network is found inconsistent the plan is considered invalid, triggering a replan. If the STN is consistent than it is guaranteed that it is possible to pick any time point within the allowed time range for an event, and still find valid times for all other events such that the plan is valid (Muscettola, Morris, and Tsamardinos 1998).

The component controllers, the system controllers and the temporal network are autogenerated at the end of the planning process and stored in the onboard storage.

Related Work

While our approach has some similarities with existing and past systems used in space applications, it deviates from those in some key aspects. Contrary to Apex (Freed 1998), PROPEL (Levinson 2005) or PRS (Ingrand et al. 1996) that are based on procedures, we use declarative monitoring and control (action) representations. In this aspect our implementation is closer to the IDEA (Muscettola et al. 2002) system. Also on the control approach we share some similarities with the IDEA system and with the RAX executive (Rajan et al. 2000), in that we implement a limited internal (reactive) planning at component (subsystem) level coupled with deliberative planning implemented by an external planning engine. On the modeling side, and contrary to systems like the RAX executive (Rajan et al. 2000) or ASE SCL (Chien et al. 2005) that use dedicated executive languages, our approach combines modeling for planning and execution. This allows us to explore the information in the planning models when generating the execution logic.

Warehouse Domain and Execution Environment

Our approach is being tested in a scenario where a UAV has to be operated to move objects in a warehouse. This domain, first presented in (Nogueira, Fratini, and Schilling 2017), introduces several planning and execution problems that are common to other domains in space robotics, and we use it as a representative scenario to test our implementation.

Domain and Problem Definition

We take the Blocks World (BW) problem as the basis to devise a more realistic domain that, while maintaining most of BW's original features, extends it with new ones relevant for a scenario where the plan has actually to be executed:

- Time. We need to take into account the time we need to process the boxes and we want to specify temporal windows within which we want a box to be processed.
- Resources. We have a battery on the UAV, that discharges as the UAV moves and that has to be recharged from time to time.
- Navigation. The UAV must take into account the 3D positions of the boxes and any obstacles in the warehouse when flying around.
- Uncertainty at runtime. The plan needs temporal flexibility to handle uncertainty at execution time.

The domain, depicted in Figure 4, is composed of one warehouse of finite dimensions containing: (a) one storage area with a number *n* of boxes of the same shape and size; (b) one loading dock; (c) one UAV with a rechargeable battery and an arm that can carry one box at a time; (d) one charging station.

The warehouse storage area has pre-defined finite dimensions, constraining the maximum number of boxes that can be placed on the floor and the maximum number of boxes that can be stacked. A UAV is used to move the boxes from the storage area to the loading dock, for posterior loading for distribution. The UAV has a battery of limited capacity and an arm with a grip to pick up the boxes. The battery discharges as the UAV moves and picks up boxes. The warehouse has a charging station used by the UAV to recharge its battery as required. Finally, the warehouse has in store a number *n* of boxes up to a maximum number limited by the warehouse dimensions disposed in an initial configuration

The task at hand involves moving a given set of boxes from the storage area to the loading dock for transport within a given time window. As they arrive in the warehouse the boxes are first placed in the storage area. The boxes must then be moved to the loading dock and arranged in a specific configuration such as to facilitate their posterior loading and delivery. The initial number of boxes, their starting and final positions as well as the time window allotted to move the boxes are given by the initial problem. The UAV initial position and battery state-of-charge are also set at the start.

We have modeled this domain using 5 timelines (see Figure 5):

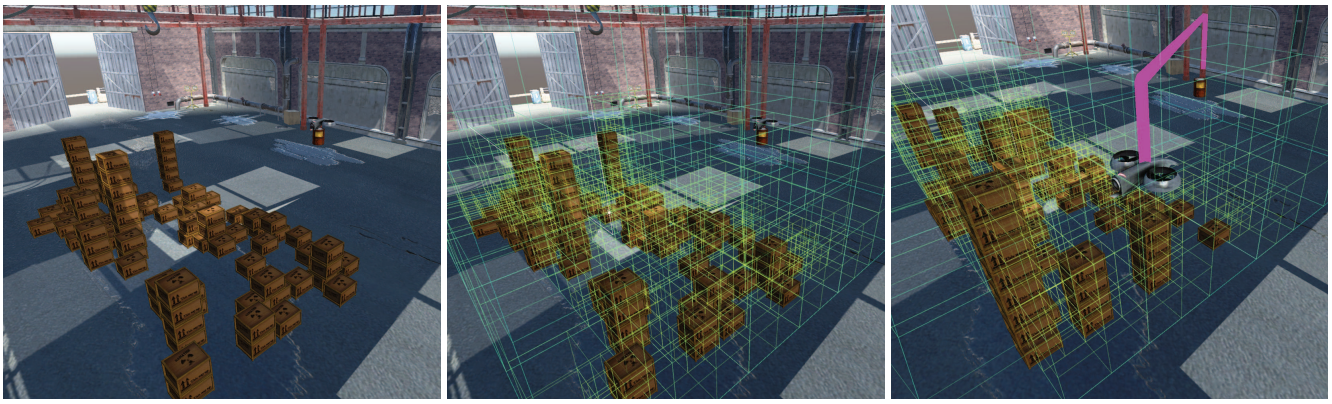
- The timeline WAREHOUSE encapsulates an octree representation of the spatial configuration of the warehouse with the position of the boxes and any obstacles.
- The timeline ACTIONS represents the actions being performed by the UAV. This timeline can take the value CHARGE($?s$), when the UAV is at the charging station $?s$, and two values UNSTACK($?box, ?position$) and STACK($?box, ?position$) when the UAV is picking or putting down a box.
- The timeline ARM represents the status of the UAV arm. The timeline can take the values ARM-EMPTY(), when the arm is not holding any box, and HOLDING($?box$), when it is holding a box.
- The timeline BATTERY models the UAV battery level as a reservoir resource.
- The timeline PATH models the actual position of the UAV and the path being followed.

A similar domain but limited to planning and using grunts and no battery resources was introduced in (Hamilton 2009).

Execution Environment

We implemented a simulation environment for the Warehouse domain that allow us to jointly evaluate the planner and the executive. The environment is implemented using the Unity game engine (UNITY 2016) that interfaces with the executive through a UDP/IP socket (Figure 6).

The initial domain was extended to allow to inject the following perturbations at runtime: (a) add and remove a box; (b) move a box to a different position; (c) switch the tag between two boxes and (d) modify the battery charge or discharge rates. By adding and removing boxes we can simulate scenarios where activities are added or removed as a plan is being executed. We can evaluate, among other things, how the executive handles changes to the plan, how it interfaces with the planner and replanning algorithms. By moving a box from its initial position we introduce delays in execution and unplanned battery depletion. As the boxes are placed in the loading dock they are checked to verify that the correct



(a) Planning domain. (b) Overlaid octree mesh. (c) UAV path.

Figure 4: Warehouse planning domain and execution environment: (4a) Planning domain showing the storage area with the stacked boxes, the loading dock on the top left, and the UAV on the charging station on the right; (4b) Octree used for path planning overlaid on the domain; (4c) UAV path on the way to pick up a box.

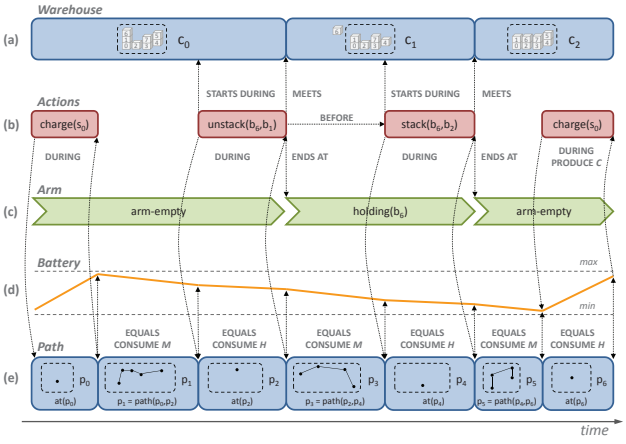


Figure 5: Timeline-based plan.

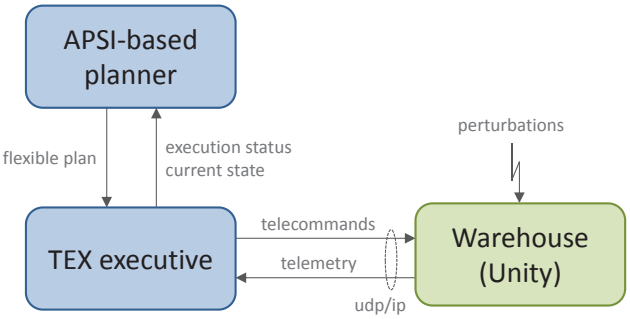


Figure 6: Planning and execution setup.

box has been collected and that it has been put in the correct place for loading. By switching the tags on the boxes we can simulate scenarios where the outcome of an action is not as intended. By modifying the battery charge and discharge rates we can simulate scenarios where resources are not consumed or produced as modeled.

For interfacing with the executive, the simulation environment provides a set of telemetry in real-time, including the current simulation time, the UAV battery state-of-charge, the UAV position and arm status, the current position of each of the boxes and the currently assigned tag for each of the boxes. The executive controls the UAV and its arm by issuing commands, including: *go-to(x,y,z)* to command the UAV to move from its current position to a new position given by the coordinates *(x,y,z)* (component controller for the timeline PATH in the model), *arm-open()* to open the UAV arm grip such as to release a block and *arm-close()* to close the UAV arm grip such as to pick up a block (component controller for the timeline ARM in the model).

Modeling for Planning and Execution

Modeling for planning and execution has traditionally been handled separately. In this work we try to harmonize the two models by extending APSI's DDL. The execution model is specified together with the planning model, by defining the set of rules to monitor and control the behavior of a given component in the target system. At runtime the domain model is used, together with the planner's output in the form of a time-flexible plan, to autogenerate the controllers, the observers and the temporal network required to execute the plan.

We extended DDL with an executive grammar, the Tinytus Executive Language (TEL), that builds on the constructs provided by DDL and the Tinytus script language, providing added capabilities to model component and system controllers, observers, actions, telemetry checks, telecommands, guards and mathematical and logical expressions.

Component Controller A component controller is modeled as a WFSA. The automaton encodes the monitoring and control logic needed to move a component (or subsystem) between states. This controller extends the DDL's state variable with execution logic. The user defines: (a) the input parameters; (b) a set of allowed states (values) with the corre-

sponding *expressions* allowing to evaluate the current component state (the automaton is in a given state if the corresponding expression evaluates to true); (c) a set of transitions with the corresponding *edges*. An edge is any combination of actions and guards that tells the controller how to move between states. The values and transitions are a superset of the ones used by the planner. An additional UNKNOWN() state is added to the original model used by the planner as a catch all for any behavior that is inconsistent with all known/modeled behavior. Other states to handle non-nominal behavior could also be defined.

In our warehouse domain we use, for example, a state variable to model the behavior of the UAV arm. This simple model used by the planner describes whether or not the arm is holding a box, and if yes, which.

```
COMP_TYPE IMPULSIVE_STATE_VARIABLE arm
  VALUES
  {
    arm_empty();
    holding(box ?b);
  }
  TRANSITIONS
  {
    arm_empty() TO
    {
      holding(?b);
    }

    holding(?b) TO
    {
      arm_empty();
    }
  }
```

So that the executive can then monitor and control the UAV arm, we need to define the corresponding controller. The controller model attaches expressions to the values and actions to the transitions. In addition, it adds extra INIT() and UNKNOWN() values to mirror the physical component behavior and to make its state fully observable.

```
CONTROLLER COMPONENT_TIMELINE arm.tl
  PARAMETERS
  {
    target: U8;
  }
  VALUES
  {
    arm_empty(): status_arm() == 0;
    holding(?b): status_arm() == 1;
    init(): status_arm() == 2;
    unknown(): status_arm() != 0 &&
               status_arm() != 1 &&
               status_arm() != 2;
  }
  TRANSITIONS
  {
    arm_empty() TO holding(?b): close();
    holding(?b) TO arm_empty(): open();
    unknown() TO init(): reset();
  }
```

Where STATUS_ARM() is an *observer* that returns an integer that is then used as part of an expression to derive the current status of the arm, and CLOSE(), OPEN() and RESET() are the *actions* used by the controller to control the transitions. Note that in this particular example, and once the arm initialization procedure (triggered by the transition to INIT()) has finished, the component will automatically transition to either ARM-EMPTY() or HOLDING($?box$). These automatic transitions are not explicitly modeled.

System Controller A system controller simply defines the component timelines that need to be actively monitored and controlled.

```
CONTROLLER SYSTEM_TIMELINE
  VALUES
  {
    position.tl;
    arm.tl;
  }
```

Observer A state observer provides an estimate of the state of the component or system. An observer is any valid expression and wraps the logic required to derive the state of a component from telemetry. In its simplest form an observer simply wraps a *telemetry check*.

```
OBSERVER U8 status_arm() {
  tm(ARM_STATUS);
}

OBSERVER FLT position_error(x, y, z) {
  sqrt((tm(POSITION_X) - x)^2 +
       (tm(POSITION_Y) - y)^2 +
       (tm(POSITION_Z) - z)^2);
}
```

Action An action implements a control directive. It is any valid expression and wraps the logic required to implement the control directive in the target system. In its simplest form an action simply wraps a *telecommand*.

```
ACTION open() {
  tc(ARM_OPEN);
}
```

Telemetry Check A telemetry check allows a user to retrieve telemetry values from within the model. This could be used within an expression to check the value of a parameter, or within a *guard* to implement a value constraint.

Telecommand A telecommand allows the user to invoke a command on the target system from within the model.

Guard A guard is used to model temporal and value guard conditions. Guards are used to dynamically enable or disable actions in a transition.

Expression An expression is any combination of arithmetic and logical expressions and can include observers, telemetry checks, actions and telecommands.

Conclusions and Future Work

This contribution addresses the problem of integrating planning and execution at three levels. First, we describe a system that integrates timeline-based planning with a model-based executive that explores the plan's time flexibility and the component-level models to minimize the need for re-planning. Second, we introduce a new domain and execution environment, representative of a robotics domain, that we use to jointly evaluate planning and execution engine implementations. Finally, we describe an extension to APSI's DDL planning language as an attempt to bring planning and execution modeling closer together. The proposed approach, though initially conceived for space robotics, could see applications in other domains.

Controllability, and in particular dynamic controllability, is a problem that we have addressed only partially so far. In the current implementation all temporal information is encoded as an STN. To properly handle situations where the

duration of certain activities or the timing of certain events cannot be controlled, we are in the process of extending our approach to make use of the formalisms provided by the Simple Temporal Network with Uncertainty (STNU) (Morris and Muscettola 2005).

References

Allen, J. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM* 26(11):832–843.

Ceballos, A.; Bensalem, S.; Cesta, A.; de Silva, L.; Fratini, S.; Ingrand, F.; Ocón, J.; Orlandini, A.; Py, F.; Rajan, K.; Rasconi, R.; and van Winnendael, M. 2011. A Goal-Oriented Autonomous Controller for Space Exploration. In *Proceedings of the ASTRA 2011, 11^{th} Symposium on Advanced Space Technologies in Robotics and Automation*.

Cesta, A.; Cortellessa, G.; Fratini, S.; Oddi, A.; and Bernardi, G. 2011. Deploying Interactive Mission Planning Tools - Experiences and Lessons Learned. *JACIII* 15(8):1149–1158.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated Planning and Scheduling for Space Mission Operations. In *SpaceOps*.

Chien, S.; Sherwood, R.; Tran, D.; Cichy, B.; Rabideau, G.; Castano, R.; Davis, A.; Mandl, D.; Frye, S.; Trout, B.; and Shulman, S. 2005. Using Autonomy Flight Software to Improve Science Return on Earth Observing One. *Journal of Aerospace Computing, Information and Communication* 2(April):196–216.

Chien, S.; Tran, D.; Rabideau, G.; Schaffer, S.; Mandl, D.; and Frye, S. 2010. Timeline-Based Space Operations Scheduling with External Constraints. In *ICAPS-10. Proc. of the 20th International Conference on Automated Planning and Scheduling*.

Chien, S.; Doubleday, J.; Thompson, D. R.; Wagstaff, K. L.; Bellardo, J.; Francis, C.; Baumgarten, E.; Williams, A.; Yee, E.; Stanton, E.; et al. 2016. Onboard Autonomy on the Intelligent Payload EXperiment CubeSat Mission. *Journal of Aerospace Information Systems* 1–9.

Dechter, R. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49(1-3):61–95.

Dombrovski, S., and Bangert, P. 2015. Introduction of a New Sandbox Interpreter Approach for Advanced Satellite Operations and Safe On-board Code Execution. In *66th International Astronautical Congress*.

EUROPA. 2008. Europa Software Distribution Web Site. https://babelfish.arc.nasa.gov/trac/europa/.

Frank, J., and Jonsson, A. 2003. Constraint Based Attribute and Interval Planning. *Journal of Constraints* 8(4):339–364.

Fratini, S., and Cesta, A. 2012. The APSI Framework: A Platform for Timeline Synthesis. In *Proceedings of the 1st Workshops on Planning and Scheduling with Timelines at ICAPS-12 (PSTL-12), Atibaia, Brazil*.

Fratini, S.; Martin, S.; Policella, N.; and Donati, A. 2013. Planning-Based Controllers for Increased Levels of Autonomous Operations. In *ASTRA 2013. 12^{th} Symposium on Advanced Space Technologies in Robotics and Automation*.

Fratini, S.; Policella, N.; Faerber, N.; De Maio, A.; Donati, A.; and Sousa, B. 2015. Resource Driven Timeline-Based Planning for Space Applications. In *Proceedings of the 9^{th} International Workshop on Planning and Scheduling for Space, IWPSS15*.

Freed, M. 1998. Managing Multiple Tasks in Complex, Dynamic Environments. *AAAI-98 , American Association for Artificial Intelligence* 921–927.

Hamilton, P. A. 2009. A Composite Architecture for a Realistic Blocks World Domain. Technical report, University of Maryland, Baltimore, Maryland.

Hoffman, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14(27):253–302.

Ingrand, F.; Chatila, R.; Alami, R.; Rober, F.; Prs; and A. 1996. PRS: High Level Supervision and Control Language for Autonomous Mobile Robots. *in: Proc. ICRA-96* 1:43–49.

Jonsson, A.; Morris, P.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in Interplanetary Space: Theory and Practice. In *AIPS-00. Proc. of the Fifth Int. Conf. on Artificial Intelligence Planning and Scheduling*, 177–186.

LaValle, S. M. 1998. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, Computer Science Dept., Iowa State University.

Levinson, R. 2005. Unified Planning and Execution for Autonomous Software Repair. In *Workshop on Plan Execution: A Reality Check*, number January.

Morris, P. H., and Muscettola, N. 2005. Temporal Dynamic Controllability Revisited. *Aaai* 94043:1193–1198.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5–47.

Muscettola, N.; Dorais, G. A.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. IDEA : Planning at the Core of Autonomous Agents. In *AAAI Eighteenth National Conference on Artificial Intelligence*.

Muscettola, N.; Morris, P. H.; and Tsamardinos, I. 1998. Reformulating Temporal Plans for Efficient Execution. *6th International Conference on Principles of Knowledge Representation and Reasoning (KR 98)* 444–452.

Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., ed., *Intelligent Scheduling*. Morgan Kauffmann.

Nogueira, T.; Fratini, S.; and Schilling, K. 2017. Autonomously Controlling Flexible Timelines: From Domain-independent Planning to Robust Execution. In *IEEE Aerospace Conference*.

Pattison, D. 2017. JavaFF. JavaFF distribution site: http://personal.strath.ac.uk/david.pattison/#software.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. F. 2007. From Precedence Constraint Posting to Partial Order Schedules. *AI Communications* 20(3):163–180.

Rajan, K.; Bernard, D.; Dorais, G.; and Gamble, E. 2000. Remote Agent: An Autonomous Control System for the New Millennium. *ECAI 2000: 14th European Conference on Artificial Intelligence*.

Smith, D.; Frank, J.; and Jonsson, A. 2000. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review* 15(1):47–83.

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast Transformation of Temporal Plans for Efficient Execution. *Proceedings of the 15th National Conference on Artificial Intelli-gence (AAAI'98)* 254–261.

UNITY. 2016. Unity Software Distribution Web Site. https://unity3d.com.

Vidal, T., and Fargier, H. 1999. Handling Contingency in Temporal Constraint Networks: from Consistency to Controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence* 11:23–45.

Weld, D. S. 1994. An Introduction to Least Commitment Planning. *AI Magazine* 15(4):27–61.

Goal Reasoning as Multilevel Planning

Alison Paredes and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
alison, ruml at cs.unh.edu

Abstract

There has been much recent interest in the topic of goal reasoning: where do an agent's goals come from and how is it decided which to pursue? Previous work has described goal reasoning as a unique and separate process apart from previously studied AI functionalities. In this paper, we argue an alternative view: that goal reasoning can be thought of as multilevel planning. We demonstrate that scenarios previously argued to support the need for goal reasoning can be handled easily by an on-line planner, and we sketch a view of how more complex situations might be handled by multiple planners working at different levels of abstraction. By considering goal reasoning as a form of planning, we simplify the AI research agenda and highlight promising avenues for future planning research.

Introduction

It is widely understood that plan synthesis is only part of the functionality that an agent needs with respect to taking intelligent action. For example, Molineaux, Klenk, and Aha (2010) posit a capacity for goal reasoning, which creates the goals that the agent's planner might attempt to achieve, determines which goals the agent will pursue at any particular moment, and monitors goal achievement. (They also include a sophisticated component for estimating the state of the world and updating the agent's model, given its actions and their observed consequences, but this is not our concern here.) In this paper, we consider whether goal reasoning is best thought of as a capability distinct from plan synthesis and action selection, or whether it might be possible to unify these two functionalities, thereby simplifying the AI research agenda. After reviewing the proposals of Molineaux, Klenk, and Aha (2010) regarding goal reasoning and their experimental benchmark scenarios, we introduce a new benchmark domain, called Harvester World, that captures many of the features of prior benchmarks. We then demonstrate a relatively simple planner, called GROH-wOW, that achieves high performance in Harvester World. We argue that this result undercuts the empirical support for goal reasoning claimed by Molineaux, Klenk, and Aha (2010) on the basis of their experiments. We then speculate about how goal reasoning functionality might be exhibited by multiple planners working together in various relationships.

Goal Reasoning as a Separate Module

A planner takes a model of the environment, a state and a goal and returns either an action or an entire plan. A goal reasoner takes a model of the environment, a current state and a goal, and returns either the same goal or a new one (Klenk, Molineaux, and Aha 2013). It also uses a sequence of expected states which should result from each action taken in the plan and the observed result of the previous action in the plan to: 1) detect discrepancies between expected states and the observed current state, 2) explain these discrepancies, which may modify the current model of the environment, 3) generate new goals which might be applicable in this new understanding of the environment, and 4) finally, decide which among the current goals and any newly generated goals should be pursued next.

For example, in a real-time strategy game such as Bos Wars (formally known as Battle of Survival), a game very similar to Starcraft, a planner might be given the goal to move a friendly harvester unit to its home base. The planner returns a sequence of actions such as, move the harvester from point A to B to C to D around an known obstacle until it reaches the base. For the use of the goal reasoner, the planner might also return a sequence of expected states such as after moving the harvester from point A to point B, the harvester is at point B. After each action is executed in a plan, the goal reasoner should get the game's current state, which in this case would be the observation that the harvester is at point B. Given this information, the goal reasoner might conclude that it should continue to pursue its current goal to move the harvester to its home base. In a partially observable environment, however, it is possible that the goal to move the harvester to its home base is not the best one to pursue in the long run, and it is part of the goal reasoner's responsibility to figure that out. For example, it might make more sense to first deploy some kind of defenses before sending the harvester home because we have reason to believe that an enemy may be lurking nearby.

Klenk, Molineaux, and Aha (2013) implement a planner with a goal reasoner and test it in comparison to planning without one. In these experiments, the goal reasoner performed better than either planning once off-line or replanning to the initial goal. In their scenarios, the most successful behavior involved responding to unobserved factors in a partially observable environment. The tests used two dif-

ferent simulations, an instance of the open source Battle of Survival (an older version of Bos Wars) and a proprietary Navy training simulation; in all scenarios there was some element of partial observability.

In this paper we focus on the three scenarios run in Battle for Survival (BoS): 1) Resource Gathering, 2) Escort, and 3) Exploration. In all of these scenarios the current state of the game could be described in terms of the units, such as a friendly harvester or not so friendly enemy, and whether harvestable things like titanium deposits have been gathered.

Actions in the BoS scenarios describe low level movements of units such as moving a harvester one step in some direction. Goals are represented as high level tasks such as moving the harvester to its home base rather than moving the harvester one step north. Consequently, planning involves figuring out which low level actions to string together to accomplish the high level action.

In the resource gathering scenario the location of harvestables may be only partially known. For example, it is possible that there could be a more conveniently located titanium deposit than the one the agent knows about. In the escort scenario, the location of enemy units is initially unknown until an enemy attacks, by which time it is too late to deploy defenses. In the exploration scenario, some paths may be impossible, making some goals unachievable. In all of these scenarios, planning with a goal reasoner performed better than without one. Klenk, Molineaux, and Aha interpreted this success as providing support for goal reasoning as a separate functionality alongside planning.

Goal Reasoning as Planning

But we hypothesize that the right type of planner might do just as well without an explicit goal reasoner. To test this hypothesis, we synthesized a new domain, called Harvester World, that captures the essential features of the benchmarks in Klenk, Molineaux, and Aha (2013) and implemented a planner for it. The previous benchmarks share the following important features: 1) partial observability: the agent does not necessarily see every aspect of the state, such as the locations of harvestables; 2) open world: the agent does not necessarily know about all the objects that exist; 3) on-line sensing: the agent learns more about the world as it takes actions; 4) multi-unit: the agent controls multiple moving objects in the environment which might move simultaneously; and 5) adversarial: there may be other agents whose goals conflict with that of the agent. Harvester World is a single domain that captures all of these features.

Harvester World

Harvester World takes place on a two-dimensional grid similar to BoS (see Figure 1) and supports test cases similar to the three scenarios described in the previous section. While both time, states and actions are discrete, the world state is only partially observable, there may be adversaries, and the agent may need to control multiple movable units. A simulator controls the ground truth of an instance of Harvester World, and at each time step, the simulator requests an action from the agent, attempts the action in the ground truth

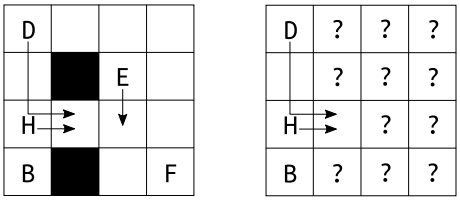


Figure 1: An example Harvester World instance: B = base, D = defender, E = enemy, H = harvester, F = food. Left map represents ground truth; right map represents the agent's current belief state.

representation and moves other factors of the environment forward, and returns an observation to the agent which the agent then uses to update its belief state.

Individual grid cells can allow movement or be fixed impassable obstacles. The harvester's home base (B in Figure 1) is also fixed. The grid also contains harvestable food (F). Food can be harvested by the agent when it is in the same cell. As soon as one food is harvested, another will sprout somewhere else in the map (the law of conservation of food). The agent controls the harvester (H) and a defender unit (D), and the grid may also contain an enemy unit (E). The agent can sense the state of the cell it is in, but cannot see food elsewhere and cannot see an enemy further than one cell away. For example, if an enemy is adjacent to the harvester then the harvester immediately discovers it without any uncertainty about whether or not it is an enemy.[1]

The harvester can move in the four cardinal directions or stay still. It may also choose whether or not a defender should move to the location of the harvester. Hence there are a total of eight actions which may be considered at each step. In BoS it is possible to move a defender independently of the harvester with the intention of strategically placing it in a way that would prevent the harvester from taking damage from an enemy unit. Our simplification does not preclude this effect, it only simplifies the number of actions needed to achieve it. Other actions were hard-wired into the agent and do not require deliberation: if the harvester is not carrying food and enters a cell with food, the food is harvested; if the harvester reaches the base while carrying food, the food is delivered.

The enemy's policy is also hard-wired and known to the agent. An enemy unit will immediately move out of a cell containing the defender. Otherwise, at every timestep the enemy will attempt to move to the next location, either North, South, East or West, along the shortest path toward the harvester's current location, taking obstacles and the defender into account. We compute this via Dijkstra's algorithm outward from the agent.

The objective of the game is to maximize accumulated reward. As in BoS, the successful player should try to maximize the amount of resources it collects while minimizing

[1]For those familiar with BoS, we omit the scout from our domain because we have tried to keep our problem succinct; a scout only increases the range at which an enemy is discovered and thus the size of the problem needed to demonstrate interesting behavior.

Figure 2: An example Resource Gathering scenario

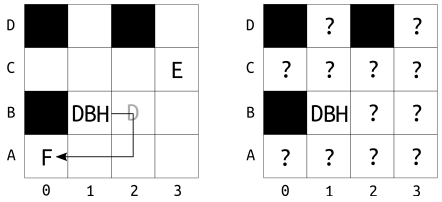


Figure 3: An example Escort scenario

the damage it takes from enemies and the amount of resources it spends to play. Every move of the harvester costs -1, even if the action does not complete because an obstacle is in the target location. Moving a defender costs -1 for each cell it traverses on its way to the harvester. If the harvester delivers food to the base, this earns +50. If the harvester is in the same location as an enemy, this penalizes the agent -10.

Using Harvester World, we were able to reproduce the three BoS scenarios described in the previous section by varying which features should be hidden from our agent. To model Resource Gathering, we configured a world that contained a base, a harvester initially located at the base, and two food sources located in two different locations some distance from the base (see Figure 2 for an illustration). Initially only the base, the harvester and the location of one of the food are known to the agent, making the current state of the world only partially known. An efficient plan should be able to maximize the amount of food it can collect in limited amount of time by harvesting the food closest to the base, including recognizing when newly discovered food is closer than the previously known food.

To model Escort we configured a world that contained a base, a harvester and a defender both initially located at the base, an set of obstacles which partition the world into defensible regions, and one food located some distance from the base (see Figure 3 for an illustration). Initially only the base, the harvester, the defender and the obstacles are known to our agent. The belief state does not include the location of the enemy as well as the location of food. An efficient plan should be able to maximize the amount of food it can collect in a limited amount of time while minimizing the damage it takes from enemies and the resources it spends to defend against them.

To model Explore we configured a world that contained a base, a harvester, a set of obstacles and two food (see Figure 4 for an illustration). Initially only the obstacles are hidden from the planner. An efficient plan should be able to get both food, detouring around any discovered obstacles, if there is an open path to both, or give up on one if it turns out all paths to it are impassable.

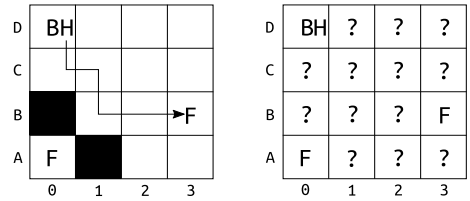


Figure 4: An example Explore scenario

HOP(s)
1. for i from 1 to N do
2. $\quad w_i \leftarrow$ sample world consistent with current belief
3. foreach action a applicable in s
4. $\quad s' \leftarrow a(s)$
5. $\quad c \leftarrow (\sum_i^N plancost(s', w_i))/N$
6. $\quad Q(s,a) \leftarrow C(s,a) + c$
7. return $\mathrm{argmin}_a\, Q(s, aA)$

Figure 5: Sketch of hindsight optimization

A Planner for Harvester World

Harvester World exhibits partially observable state, an adversary, and multiple units that need to be controlled by the planner. This is well beyond the scope of classical planning tasks. To handle this problem, we turn to an approximate method for planning in large partially-observable Markov decision processes: hindsight optimization (Yoon et al. 2008; 2010). Hindsight optimization planning has been shown capable of performing better than reactive planning in a partially observable environment when we are unsure if and when new goals may arrive (Burns et al. 2012). In their planner OH-wOW, Kiesel et al. (2013) showed that hindsight optimization can be used to plan on-line in open worlds where the existence of important objects is unknown. We extend the work of Kiesel et al. (2013), hence our planner for Goal Reasoning with Optimization in Highsight in Open Words is called GROH-wOW.

Fundamentally, hindsight optimization works by sampling from a model of all possible realizations of the uncertain features in the current state, and then finding the best deterministic plan in each of these sampled world, and using the values of these plans to estimate the value of the agent's possible successor states and hence determine what the best next action is. See Figure 5 for a pseudocode sketch ($C(s,a)$ is the cost of taking action a in state s). Sampling avoids explicitly representing a belief distribution over a large number of possible states, and the fully specified worlds allow conventional fast deterministic planning techniques to be used. Despite its practicality, unlike methods such as UCT (Kocsis and Szepesvári 2006), hindsight optimization is well-known to not be guaranteed to converge to the optimal action in the limit of infinite sampling. We use the hindsight optimization strategy within a receding horizon paradigm: the deterministic planner looks ahead only to a limited horizon while finding the maximum reward plan for each sampled world, then the agent takes a single action and the cycle repeats.

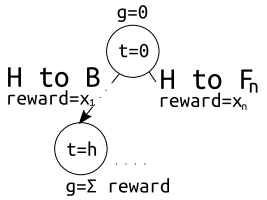


Figure 6: Search tree

Following the algorithm sketched above, GROH-wOW has two main parts: the agent's action planner, which takes a model of Harvester World, samples from it and passes these samples on to the deterministic planner, and the deterministic planner subroutine, which will then find the best plan in each of these samples and return to the action planner the value of each. The hindsight optimization planner will then use these values to find the expected value of each of the immediate successors of the current state and return the action that leads to the next state with the highest expected reward. The action is executed and a set of observations is returned which may be used to update the agent's belief model, which will be given to the hindsight optimization planner at the next timestep to use when determining the next action.

Sampling over the remaining uncertainty in the belief state given to the planner is intended to resolve all ambiguity in the current state in order to leverage deterministic planning. In Harvester World this means resolving two different types of uncertainty and partial observability such as where the enemy is located and stochastic effects such as where food may grow next. We model this uncertainty as a uniform distribution over unknown factors in a belief state. For example, to model Resource Gathering the belief state given to the sampling function might omit the location of food. It might assume that there are at most two food in the world, but the belief state does not provide the location of both of them. Sampling a world from this belief would result in one that contains both foods at a specific locations, e.g. food at positions 2 and 3 in Figure 2. We can then plan to move the harvester to the food at 2 and back to the base for a reward of $+50 - 3$. The belief state might also contain ambiguity about future states. Since in Harvester World the amount of food in the world never diminishes, gathering food implies that new food must have grown somewhere. Hence when we sample a world, we also determine exactly where food might grow in the future, resolving both the uncertainty regarding where food is located now but also how its location might possibly change. Sampling resolves all ambiguity into a set of possible worlds to pass along to the deterministic planner.

Deterministic Planner

The deterministic planner uses a basic breadth-first search from each of the now complete models of the current state out to a given time horizon, enumerating all states which can be reached within $t = horizon - 1$ time steps. It then considers the plan with the highest reward among all of the states reachable at $t = horizon$, and returns this value to the hindsight optimization planner to aggregate.

To make deterministic planning faster, we note that there are only a few 'macro actions' worth considering. In the Resource Gathering scenario there are three macro actions that it makes sense for the deterministic planner to consider: move the harvester to the base, move the harvester to the known food at a specific location, or move it to the unknown food at a different location. This would have been impractical without having first sampled a world where the location of the unknown food can be known, but in hindsight optimization it becomes a simple matter of planning the shortest path from where ever the harvester is now to the closest food and back to the base for the optimal reward, assuming this can be completed within the given horizon. If a goal would take longer to complete than the remaining time, then we consider the macro action incomplete and its result is the state of the world up to the horizon. These macros reduce the branching factor and search depth that the deterministic planner must search. Additional strategies, such as designing a heuristic reward-to-go function, are also possible.

In the Escort scenario the deterministic planner has four macro actions to reason about after sampling determinizes the unknown attributes of the model: move the harvester to a food, move both the harvester and the defender to a food, move the harvester to the base, or move both the harvester and the defender to the base. Recall at each time step we simulate the movement of the enemy. We can do this during planning as well since we assume that, while we do not know exactly where our enemy is, we have a reasonable model of how it moves. (In the absence of a good model, we would just sample possible enemy actions.) If the enemy, wherever sampling has imagined it might be, would interrupt the completion of any of the macro actions above, then we consider the macro action incomplete, like we do when we have run out of time. Its result is the state of the world when encountering the enemy.

In the Exploring scenario, the deterministic planner must reason about the same three macro actions used in Resource Gathering but with some nuance. While the existence of obstacles are initially unknown in the model given to the hindsight optimization planner, for simplicity we do not speculate about where they could be. Instead as obstacles are discovered, macro actions like moving harvester to food at position (A, 0) in Figure 4 potentially become more expensive to complete. The deterministic planner is able to take these changes into consideration when deciding if it should first get the food at (A, 0) and drop it off at the base before getting the food at (B, 3). If the shortest path to (A, 0) becomes too long or even impossible because of obstacles then it should choose to get the food at (B, 3) first.

Experimental Results

The central empirical question at issue in this paper is whether a planner that lacks an explicit goal reasoning component can perform well in Harvester World instances similar to the problems considered by Klenk, Molineaux, and Aha (2013). We addressed this question by running GROH-wOW on each of the BoS scenarios.

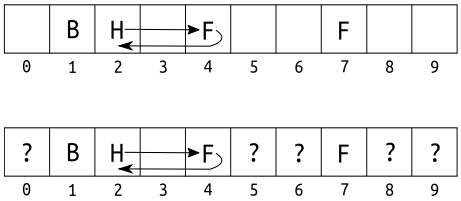


Figure 7: Example sequence in a Resource Gathering instance

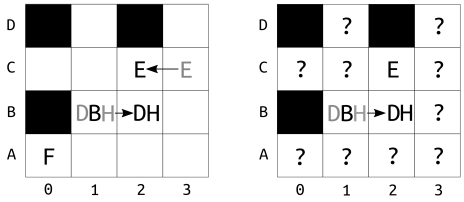


Figure 8: Example step in an Escort instance

Experiment 1: To demonstrate Resource Gathering we ran hindsight optimization on a single instance of Harvester World configured for Resource Gathering as described above for 10 time steps on a 10 x 1 grid, with a sample size of 10 and a horizon of 20, which would allow enough time to foresee the value of getting food located as far away as position 9 and bringing it back to the base. Figure 7 illustrates our results. The top map describes an intermediate state from the perspective of the simulator; it is perfectly known. The bottom map describes the agent's belief state at that time. In the simulation's initial state the harvester (H) is at the base (B) and the planner knows about the food (F) at position 7. At each time step the simulator gives our planner an incomplete model of the world, and the agent returns a single action to the simulator. In this test, the harvester discovered food at position 4 and thereafter the planner determined it should return to base instead of continuing to the original food at position 7. Thus on-line planning can recognize an opportunity as well as a system using a goal reasoner.

Experiment 2: To demonstrate our planner in the Escort scenario, we configured 10 instances of the Escort scenario, varying the initial location of the base, where the harvester and defender start, the hidden enemy and a hidden food. The number of obstacles in each scenario was constant and their location was known to the planner. We ran each of these 10 scenarios for 100 time steps, allowing the hindsight optimization planner to sample 10 worlds at each time step, and gave the deterministic planner a horizon of 10 time steps. Figure 8 illustrates one time step in one problem instance. In this example the agent decided to move the harvester (H) along with the defender (D) East away from the base (B) before the location of the enemy was ever revealed, illustrating the ability of hindsight optimization to consider unobserved factors such as the possible location of the enemy. All problem instances exhibited similar behavior as the agent explored the map and collected food.

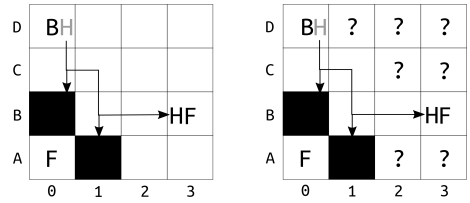


Figure 9: Example sequence in an Explore instance

Experiment 3: To demonstrate hindsight optimization in the Explore scenario, we configured an instance of Harvester World with two food, one of which was unreachable due to obstacles. We made the location of all of the food known but not the location of obstacles, and allowed the deterministic planner a horizon of 10. We did not need to sample more than one possible world since the location of the food was known. As figure 9 illustrates, the agent initially attempted to get the food in the lower left corner by exploring the cells along the left side of the grid where it discovered two obstacles which precluded its ability to get the food it was initially after; it then went for the food on the right side of the grid. This behavior highlights the influence of the objective function in goal reasoning. Once the obstacles at (A, 1) and (B, 0) have been discovered there is no plan which sends the harvester to the food at (A, 0) that could improve the total expected reward measured by the objective function so the food at (A, 0) is abandoned.

Discussion

So far we have shown that a hindsight optimization planner can perform well in the same class of problems as was previously thought to require goal reasoning. That is not to suggest that these problems do not require goal reasoning, just that a discrete goal reasoner is not necessarily required to achieve the desired functionality. We propose that our planner demonstrates many of the aspects of goal reasoning. We explore this with reference to Johnson et al. (2016)'s goal lifecycle.

In previous work goal reasoning has been organized into discrete steps defining a goal lifecycle. A goal must be formulated, selected, expanded, committed, dispatched, and then execution and resolved (Johnson et al. 2016; Roberts et al. 2016). We consider these in turn. If one were to interpret existing food as goals for planning with low-level actions, then we could view grounding a belief state into a set of completely known worlds as formulating a set of goals. For example in Figure 7, in the belief state the location of only one food is known; there is one known goal, retrieving the food at position 7. Sampling however could result in a world in which there is a food at position 0, in which case sampling generates the additional goal, retrieving the food at position 0. Selection is then the process by which our planner decides which food to pursue first; the low-level planner will recommend the order that gets us the most expected reward within the planning horizon. Expan-

sion is finding the action that best follows this longterm plan. Goal evaluation metrics used in previous work like principles and intensity levels (Johnson et al. 2016; Cox, Dannenhauer, and Kondrakunta 2017) are encapsulated in the idea of a reward. For example, the ordering of macro actions in optimal plans represent the most efficient ordering of goals as high-level tasks to achieve the best possible total reward without having to be explicitly prioritized.

In these ways, one can view our planner as engaging in the elements of goal reasoning. However, there is also a more subtle interpretation. One can also find goal reasoning in how hindsight optimization uses the objective function to estimate the expected reward and choose the next low-level action toward this 'estimated goal.' Goal generation in this sense could be seen as the process of finding the number of times we should be able to put the harvester at the base with food within the planning horizon. Selecting one of these goals (or abstract plans) is easy when the world is completely known; there is only one optimal number of times harvester can be at the base with food. Goal expansion then is deciding which low-level action is the best next step in making the harvester arrive at the base with food the expected number of times. This is simply the next action along the plan that puts the world in a state with the best possible reward within the planning horizon according to the objective function.

The selection step in the goal lifecycle becomes more interesting in a partially known world because the true goal, the number of times we can make the harvester at the base with food true, is unknown as well, but with hindsight optimization we can estimate this quantity and estimate the true goal; the expected value of the true goal is the expected value of the optimal plans across all possible worlds. Selection can be interpreted as the process by which hindsight optimization finds this estimate. Given the current belief state, the selected goal is the estimated number of times we could expect to put the harvester at the base with food. Because each grounded version of the belief state has the potential to change the costs involved in achieving the goal state and so change the number of times it can be achieved within the planning horizon, the goal in each possible world may vary, but sampling from the belief space allows us to quickly find an estimate of the true goal. Then in the expansion step of the goal lifecycle we can use this estimate of the true goal to compare how much each low level action is expected to move the agent closer to the estimated goal.

After selection and expansion our agent immediately dispatches the next action with the promise of achieving the selected goal. The effect of the next action, along with any new information about the environment, is then incorporated in the next round of goal reasoning. Because every new observation has the potential to dramatically change the expected value of the goal, such as we saw in the Explore scenario when the discovery of obstacles surrounding the food at (A, 0) made one of the food unreachable, an on-line planer can be said to repeat goal reasoning at every time step, beginning with the evaluation of the state resulting from the previous action and possibly selecting a new goal.

Goal Reasoning as Multilevel Planning

Although we have demonstrated how an appropriate planner like GROH-wOW can handle the types of goal reasoning tasks present in Harvester World and simple scenarios of its type, we do not mean to claim that a single planner can give high performance in truly huge domains such as dealt with by human emergency responders and battalion commanders. It seems clear that for planning over very long horizons or with huge action spaces, hierarchical techniques will be necessary. But again, it does not necessarily follow that a separate goal reasoning component is necessary. Instead, we speculate that multiple levels of planning may be appropriate. These planners might coordinate their computation in various ways — we will sketch three. In the most classic approach, a high-level planner might generate a plan of very high-level actions, such as establish communications network, rescue survivors, and treat wounded, while a low-level planner might treat just one of the high-level actions as a goal to achieve through the generation and coordination of many lower-level actions. In this way, subgoals are created that allow low-level planning to be limited to subproblems with shorter solutions, speeding search. In this sense, the high-level plan is a form of loose guidance using landmarks for the low-level planning.

This arrangement raises the question of domain and goal representation for the high-level planner. Using the concept of reward, we believe that a high-level planner can reason about even abstract concepts such as keeping friends safe by postulating the existence of unseen adversaries, reasoning about their behavior, and selecting actions such as defensive patrolling. As we have shown, there is nothing inherent in a partially observable stochastic adversarial domain that demands a goal reasoner.

This traditional concept of multi-level planning, in which the actions of one level are goals for the next, is just one way in which multiple planners can be coordinated to solve large problems. A second way is by viewing a plan at the higher more abstract level as establishing constraints on the possible state space to be considered by the lower-level planner. The lower-level planner considers only concrete states that project into the abstract states visited by the abstract plan. In this way, the high-level planner specifies a kind of tunnel in the state space that constrains the search of the lower-level planner (Gochev et al. 2013). In this view, the states spaces of the two planners must be similar enough to be aligned with projection, with one just being more abstract than the other.

Finally, a third method of coordination is to use the high-level planner to guide the lower-level planner in a more flexible way. The previous tunnel method can be seen as the high-level planner pruning away concrete states that map to abstract states that are not in the plan. In a sense, such concrete states are given heuristic values of infinity (Holte 1995). A more flexible method is to use the high-level planner directly as a heuristic function, allowing it to return values smaller than infinity and allowing the low-level planner to eventually expand states that project outside of the initial high-level plan if necessary. Planning in an abstract representation of the problem has been shown to be an effec-

tive heuristic (Hoffmann and Nebel 2001). One can regard our Harvester World planner as being of this type, as the deterministic planner finds more abstract plans, which are used to evaluate states for the low-level action planner.

Conclusion

Despite agreement on the observable capabilities of agents, such as dealing with multiple goals in a dynamic partially-observable world with adversaries, it is not obvious how those capabilities might be implemented in various computational modules. Locating goal reasoning in a specialized module is certainly one way to go. However, our results show that the empirical evidence presented for that architectural commitment may not uniquely support that choice. While Harvester World is certainly beyond the classical planning problem setting, we have demonstrated that modern planning technology can handle partially-observable open worlds with multiple units and adversaries. We have also speculated about three possible ways in which multiple levels of planners might be arranged to handle more challenging domains.

Acknowledgements

We would like to thank Mak Roberts, Will Doyle, Tianyi Gu, Jordan Ramsdell and our anonymous reviewers for their thoughtful comments. We also gratefully acknowledge support from NSF (grant 1150068).

References

Burns, E.; Benton, J.; Ruml, W.; Do, M. B.; and Yoon, S. 2012. Anticipatory on-line planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Cox, M. T.; Dannenhauer, D.; and Kondrakunta, S. 2017. Goal operations for cognitive systems. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*.

Gochev, K.; Cohen, B.; Butzke, J.; Safonova, A.; and Likhachev, M. 2013. Path planning with adaptive dimensionality. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-13)*.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Holte, R. C. 1995. The tradeoff between speed and optimality in hierarchical search. Technical Report 95-19, University of Ottawa Computer Science.

Johnson, B.; Roberts, M.; Apker, T.; and Aha, D. W. 2016. Goal reasoning with information measures. In *Proceedings of the Fourth Annual Conference on Advances in Cognitive Systems (ACS-16)*.

Kiesel, S.; Burns, E.; Ruml, W.; Benton, J.; and Kreimendahl, F. 2013. Open world planning for robots via hindsight optimization. In *Proceedings of the ICAPS Workshop on Planning and Robotics*.

Klenk, M.; Molineaux, M.; and Aha, D. W. 2013. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence* 29(2):187–206.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML-06)*, 282–293.

Molineaux, M.; Klenk, M.; and Aha, D. W. 2010. Goal-driven autonomy in a navy strategy simulation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*.

Roberts, M.; Shivashankar, V.; Alford, R.; Leece, M.; Gupta, S.; and Aha, D. W. 2016. Goal reasoning, planning, and acting with actorsim, the actor simulator. In *Proceedings of the Fourth Annual Conference on Advances in Cognitive Systems (ACS-16)*.

Yoon, S.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *Proceedings of AAAI*.

Yoon, S.; Ruml, W.; Benton, J.; and Do, M. B. 2010. Improving determinization in hindsight for on-line probabilistic planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*.

Automated Planning with Goal Reasoning in Minecraft

Mark Roberts[1] **Wiktor Piotrowski**[2] **Pryce Bevan**[3] **David Aha**[1] **Maria Fox**[2] **Derek Long**[2] **Daniele Magazzeni**[2]
[1]Naval Research Laboratory, Code 5514; Washington, DC, USA | {first.last}@nrl.navy.mil
[2]Department of Informatics, King's College London, United Kingdom | {first.last}@kcl.ac.uk
[3]Georgetown University, Washington, USA | pwb8@georgetown.edu

Abstract

We combine PDDL/PDDL+ planning with goal reasoning to leverage the strengths of both and succeed in a limited variant of Minecraft. Automated planners have long been able to reason about numeric fluents and exogenous events, but remain largely confined to closed worlds with full observability. Goal reasoning can respond to dynamic, open worlds and partial observability, but must rely on an effective planner. We demonstrate that combining goal reasoning with automated planning reduces the overall computational effort to achieve goals while succeeding at multiple domain specific metrics. We highlight important design decisions in PDDL1.2, PDDL2.1, and PDDL+, including the use of PDDL+ events to model opportunistic goals. We close with a discussion of trade-offs associated with choosing the modeling features and identify a number of challenges for the next generation of planning systems.

1 Alex's Quest

Consider an agent, Alex, at the end of a 300-meter hallway with entries to dozens of rooms along each side containing randomly placed resources (e.g., wood, diamonds, iron ore, and coal), necessary crafting equipment (e.g., a workbench for crafting and a furnace for smelting ore), and randomly spawned zombies which can harm Alex. Alex can only observe the world directly nearby and must reach the far end of the hallway with a crafted diamond sword in hand. After this hallway, Alex faces a dungeon filled with enemies, therefore it would be best to also have a full complement of armor, a stack of torches, some ladders, etc.

This quest is derived from Minecraft, an open-world sandbox game where players choose their own objectives such as building structures, collecting resources, crafting, fighting enemies, or exploring. The world consists of 1 meter voxels (i.e., cubes). Minecraft has many properties of an ideal testbed for designing planning and acting techniques. It includes such issues as multiple agents (both cooperative and adversarial), partial observability, complex tasks (e.g. crafting tools, building), to name a few. The quest, and similar Minecraft challenges, can be easily solved by automated planning systems using replanning with two additions: opportunistic goals provided by PDDL+ events and goal reasoning.

PDDL+ (Fox and Long 2006) is an extension of the standardized modeling language in automated planning, PDDL (IPC Committee 1998). In conjunction with previous versions, PDDL+ introduces independent processes and exogenous events. Processes have time-dependent continuous effects, whereas events bring about instantaneous discrete change. Both elements they are triggered *by the environment* as soon as their preconditions are satisfied; the planner does not control processes/events. Events are ideal to model the appearance of entities and resources and enable the modeling of opportunistic goals.

Goal reasoning allows an agent to deliberate about its own goals during execution. This allows an agent to be more adaptable to changes in the world by determining, for example, when it should replan or when it should adjust its sensing. A recent implementation of goal reasoning, called ACTORSIM (Roberts et al. 2016b), has been applied to Minecraft, but ACTORSIM lacked a connection to PDDL-based planners and some features needed to support observations.

We extend ACTORSIM to translate Minecraft into PDDL and examine trade-offs of this combination. The contributions of this paper include: (1) an extension of ACTORSIM to include mobs, resources, and the ability to observe them; (2) PDDL/PDDL+ models of Minecraft; to our knowledge, these are the first PDDL models of Minecraft that include randomly placed resources and zombies; (3) a discussion of how the domain modeling evolved, highlighting the strengths and shortcomings of each approach; (4) an effective mechanism for managing an open, dynamic world via opportunistic goals implemented as PDDL+ events; and, (5) an application of goal reasoning to reduce the cumulative search effort of the planner whilst maintaining domain-specific metrics such as resources collected. In the remainder of the paper, we describe the planning models we constructed, how we leverage them during goal reasoning, our evaluation of the system, and related work. We close with a discussion of future work.

2 Interacting with Minecraft

To interact with the Minecraft game, we use the publicly available tool called ACTORSIM (Roberts et al. 2016b). This connector exposes state information about the world and discrete motion primitives for controlling character. It also provides a way to construct challenge problems like the introductory quest. The previous version of ACTORSIM only supports actions to move forward, mine, or build a bridge and only reports the blocks directly around the player.

We extended ACTORSIM considerably to support our study. We added actions to move the character north, east, south, or west and actions to collect resources. We extended the sections available to include zombies, diamonds, bread, and wood as well as observations of entities and items around the player. Finally, we added a PlanManager to convert this state to PDDL and run a planner. Figure 1 shows the relationships between the core components for our study.

The PDDL planner accepts input files and produces a single plan; not shown is the PlanManager component that assists with this process by creating the PDDL files, running the planner process, and parsing the plan output. The 'Alex Controller' ensures that all actions taken are safe to execute – that is, that the character doesn't walk off a cliff or into lava. It also abstracts the game state (e.g., blocks, observations, inventory) into data structures that can be read by other components. Finally, the 'Goal Reasoner' sets up challenges, monitors experiments, and manages the character's goals.

3 Modeling Minecraft

All planning modeling languages have been designed for particular classes of domains. With a multitude of domain definition languages available, consideration of all features of the scenario and its future extensions is necessary to capture the model accurately. The choice of modeling language is crucial as it affects the size of the search space, branching factor, and generally how closely the domain represents the corresponding real world problem. A domain language not well-suited to a given problem can result in an inaccurate representation of the essence of the real-world scenario, as well as a convoluted and bloated domain. A more expressive model allows a closer resemblance to the Minecraft world but can increase search cost. We aim to arrive at a model that is expressive enough while still solvable within a few seconds. We modeled the Minecraft domain incrementally, adjusting our choice of modeling languages as the desired set of features expanded over time.

Observation and Plan Zones All the models share the concept of zones. Minecraft coordinates are given in X (east-west), Y (up-down), and Z (north-south) coordinates where north, west, and down are negative. In this paper, we focus on the X-Z grid of obstacles around the player and plan to add height in future work. Even without height, the challenges we present are 20 blocks wide by 150 blocks long. The full obstacle course takes 90-180 seconds (or more) to plan, far too slow for reactive execution. So we must consider how much of the world to reason with when encoding the local state around the player into a problem file for the planner. We define *observation and plan zones* around the player given by the distance (front,back,left,right) away from the player in each of these directions. An *observation zone* is the larger of the two, and defines the distance away from Alex that items are observed. While this reduces the level of detail, planning for the entire observation zone can still take tens of seconds or longer and is too slow for a tight execution loop. So we use a much smaller *plan zone* that the planner can solve quickly.

Consider Figure 2 which captures the abstract state from the image in Figure 1. Alex (shown as Y) is about to walk into

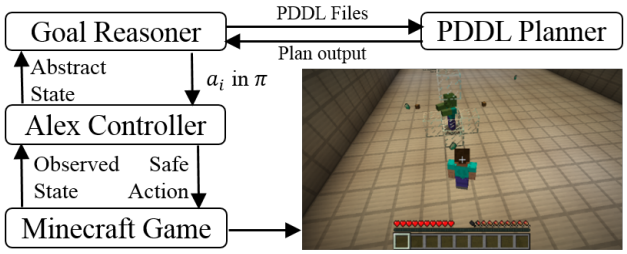


Figure 1: Overview of the planning and acting components.

```
      x3 X4 X5 X6 X7 X8 x9
z-9   D  W                W
z-8      .  .  T  .  .
Z-7      .  X  X  X  .
Z-6      .  X  X  X  .
Z-5      .  X  X  X  .
Z-4      .  .  .D .  .
Z-3      .  .  .  .  .
Z-2      .  .  .Y .  .
Z-1      .  .  .  .  .
            PDDL2.1
```

```
      x3 X4 X5 X6 X7 X8 x9
z-9   D  W                W
z-8      .  .  T  .  .
Z-7      .  .  =  .  .
Z-6      .  =  Z  =  .
Z-5      .  .  =  .  .
Z-4      .  .  .D .  .
Z-3      .  .  .  .  .
Z-2      .  .  .Y .  .
Z-1      .  .  .  .  .
            PDDL+
```

Figure 2: Sample problem and observation zones around Alex (denoted Y) containing walkable cells (.), unsafe cells (X), the intermediate target (T), glass blocks (=), diamonds (D), wood (W), and a zombie (Z). The left plot shows the PDDL2.1 representation, while the right plot shows the PDDL+ version.

a zombie (Z) noted in the PDDL 2.1 model as unsafe (X) or in the PDDL+ model as surrounded by glass (=). We cage the zombies in glass because we lack an effective way to defend against zombies; in future work, we discuss our plans for using reactive strategies for defense. Walking in any corner squares around the zombie, results in health damage, so Alex must avoid these positions if possible. The PDDL+ model encodes this knowledge but the PDDL2.1 model does not. Thus, the goal reasoner marks as unsafe all blocks around a zombie, effectively enforcing safety. The observation zone in this small example includes the entire area. The plan zone of (6,1,2,2) is six cells in front of Alex, one cell behind Alex, and two cells to the left and right. Walkable cells (.) in the plan zone are safe spots where Alex can stand. An intermediate target (T) creates a feasible subproblem for the planner. Just outside the plan zone are resources of wood (W) and diamond (D). As the end of the hallway is always north of the character, the intermediate target is the northernmost walkable cell closest to the hallway's end.

A propositional model Our first model employed PDDL1.2 (IPC Committee 1998) and relied on propositional variables to represent Alex and the surrounding area. Each cell in the grid was defined in relation to its neighboring cells (i.e. cell_i "is north of" cell_j). Discrete change and purely propositional set of variables served well as a proof of concept and the character could achieve the end goal. Research in propositional encodings is the most mature, and every

planner we tried succeeded at the model for small problem sizes. The character was able to move through hallway and overcome most obstacles, but this required replanning every 3-5 steps. The planner could sometimes stall if the planning zone wasn't big enough to get around an obstacle.

Though simple to construct, the model scaled poorly. This version only included the goal of reaching a specified target cell and extending the model to include additional tasks was particularly difficult. The key to reducing the number of replanning episodes and eliminating stalls was to increase the problem size, allowing the planner to plan further each episode. However, the problem could not scale or include height because each cell and its n-way connections must be explicitly stated in the problem file.

A Numeric PDDL2.1 model To increase the problem zone, we turned to PDDL2.1 (Fox and Long 2003) which adds numeric fluents, durative actions and continuous action effects. Expanding the model to account for entities and resources required major changes in the domain.

Instead of predicates between cells, we represented each cell with a numeric function for each dimension (e.g., x_cell, z_cell) and a type (e.g., walkable, unsafe). Each cell in the grid only takes four statements: the declaration, the x and y values, and the cell type (walkable, unsafe, etc.). Queries such as "is-north-of" are easily calculated and transitive. This formulation provided information about distances between cells, reduced the problem size, and eased scaling.

An added benefit of moving to PDDL2.1 was the ease in representing inventory and resource collection. Resource location, type, and quantity could easily be specified using numbers without adding new predicates or functions.

The PDDL 2.1 model heavily relies on the goal reasoner to add the goal conditions forcing Alex to visit cells with resources and avoid cells with zombies. Extending the domain to consider collecting resources is complicated by the fact that these are randomized. A simple approach is to add a goal condition to "collect" all resources, but this makes the problem unsolvable if no resources exist in the current plan zone. To solve this, the PDDL2.1 model uses a "visited" predicate to force the planner to collect resources. Similarly, avoiding zombies requires an "unsafe" annotation for cells around a zombie. The goal reasoner and PlanManager include these predicates for resources/zombies within the plan zone.

A Numeric PDDL2.1 model with a plan metric To facilitate collecting resources without relying on "visited" predicates, our next model employed plan metrics that maximized wood and diamond resources in the inventory, through `collect` actions. However, the metric alone does not encourage the planner to collect the drop, and instead the planner focuses on a direct trajectory to the target cell at the end of the corridor. We believe this behavior results from the bias of a planner toward the shortest plan. This approach completely ignores the resources, a focal point of this work, so we exclude this model from further discussion and evaluation. Instead, we employ PDDL+ events with what we call opportunistic goals.

A Numeric PDDL+ model with opportunistic goals PDDL+ (Fox and Long 2006) extends PDDL2.1 with independent processes and exogenous events. It has mostly been used to define hybrid systems, i.e. models exhibiting both discrete and continuous behavior, often with non-linear system dynamics. Indeed, some of the continuous behavior in Minecraft can only be represented by independent processes (e.g., health regeneration cannot be directly manipulated by Alex because Alex needs to eat, which in turn activates a continuous process increasing his health level at a steady rate.) On the other hand, events are best suited to model some of the innate features of Minecraft, such as resources or hostile entities entering Alex's field of view. In addition, PDDL+ events can also be compiled to act as opportunistic goals/plan preferences which allow us to maximize the collected resources, as described in Section 4.

Opportunistic goals enabled by PDDL+ events are triggered when such resources are observed and need to be collected. Unlike soft goals, which can be ignored by the planner, opportunistic goals must be satisfied when encountered (e.g. collect resources when seen along Alex's path or avoid zombies). Moreover, opportunistic goals are only ever considered when required (e.g. if Alex has not collected a sufficient amount of a given resource), otherwise they are ignored to avoid unnecessary computational effort.

For each type of resource, a propositional fact is added to the goal condition and falsified by events. For example, to trigger wood collection we falsify `(wood_resource_collected)`:

```
(:event wood_resource_appears
  :parameters (?start_cell - cell ?resource - cell)
  :precondition (and (alex_at ?start_cell)
    (wood_resource_collected)
    (< (wood_in_inventory) (wood_goal))
    (< (- (z_coord ?resource) (z_coord ?start_cell)) 5)
    (= (resource_type ?resource) 3))
  :effect (and (not (wood_resource_collected))))
```

This event triggers when wood is close to Alex.

There are cells that Alex should avoid at all costs. Zombies are hostile entities which attack Alex when nearby. Each strike from a zombie decreases Alex's health. To trigger zombie avoidance we falsify `(alive)`:

```
(:event zombie_damage
  :parameters ( ?start - cell ?zombie_cell - cell )
  :precondition (and (alive) (alex_at ?start)
    (= (entity_type ?zombie_cell) 100)
    (>= (- (z_cell ?zombie_cell) (z_cell ?start)) -1)
    (<= (z_cell ?zombie_cell) (z_cell ?start))
    (<= (- (x_cell ?start) (x_cell ?zombie_cell)) 1)
    (<= (- (x_cell ?zombie_cell) (x_cell ?start)) 1) )
  :effect (and (not (alive)) ))
```

This event triggers when a zombie appears in the plan zone and when Alex is in striking distance from the zombie.

These goal conditions are satisfied in the initial state (i.e. set to true by default) and falsified by the event, forcing Alex to collect the resource or avoid the zombie before continuing to the target cell. In the absence of resources or zombies, the resource-related goal conditions remain satisfied throughout.

Overall, the PDDL+ domain allows finding plans which prioritize immediate acting in the local space, while assuming

the subsequent sections of the search space are restricted to movement only. As mentioned before, the triggered events also drive the replanning strategy, specifying when to replan, to efficiently catch changes in the local plan zone without redundant effort. However, problems with large numbers of resources and zombies, even in a restricted subproblem, can quickly become intractable. So we turn to goal reasoning.

4 Goal Reasoning

Goal reasoning enables an actor to deliberate online about its goals. Roberts et al. (Roberts et al. 2016b) formalize goal reasoning as progression of goals according to a goal lifecycle. A goal is *formulated* and *selected* before planning; these two stages can filter an open world by only formulating or selecting relevant goals. Let a plan $\pi = \langle a_i..a_n \rangle$ be a sequence of actions $a_i..a_n$. A goal is then *expanded* into one or more plans Π and a single plan $\pi \in \Pi$ is *committed* for execution; goal reasoning naturally extends to approaches that find more than a single plan. When π is sent for execution the goal is *dispatched*; the simplest system sequentially executes each action $a_i \in \pi$. A goal impacted during execution is *evaluated* for the best course of action and *resolved* appropriately. Example *resolve* strategies include repairing π, replanning to create a new π, regoalling, etc. We focus on replanning in this work.

Goal Reasoning Example Our study focuses on three goals from the introductory paragraph: get to the end of the hallway (rooms excluded for the moment), avoid zombies, and collect resources. These goals are formulated in a combined 'Complete Hallway' goal that is selected once the hallway is constructed. The PlanManager expands a plan for this goal by converting the goal to PDDL and calling an automated planner; the zombie and collect goals are translated to the correct PDDL model, as described earlier. The planner automatically commits to the first plan it finds.

The 'Complete Hallway' goal is dispatched by preparing the full plan for execution. But because the Controller only executes one command at a time, the goal reasoner creates subgoals of 'Complete Hallway' so each action of the plan can be tracked independently. For example, a move-north action is converted to a subgoal to be at the location north of the player. The goal reasoning system automatically selects the subgoal, skips planning since it is not needed, and dispatches the single action to the controller. The controller completes the action and reports back to the subgoal. When the subgoal completes, 'Complete Hallway' is notified and can check to see if any of its goals are impacted. Changes during execution may result in the need to replan. Further, Alex will clearly need to replan when all actions of the current plan are completed. Alex might also replan when zombies, wood, or diamonds appear. In all cases, the goal reasoner determines when replanning is needed.

Goal Reasoning with Planning So far, we have explained how opportunistic goals in PDDL+ allow a planner to synthesize a plan under partial observability and how a goal reasoning system adjusts goals based on Alex's context. Consider again Figure 2. The resulting plan for either plot moves Alex north two blocks to collect the diamond, east and north around the zombie, and west toward the intermediate target. But there is a problem with the resources in row z-9 that lie outside the plan zone: the planner is blind to them because they are not reported in the PDDL model. It could replan at every step but this may waste computational effort or needlessly slow down Alex. There are several more reasonable choices for the goal reasoner that we explore: (1) It could use a Small plan zone to identify opportunities close to Alex and force replanning at fixed intervals. (2) It could use a Large plan zone, which could result in increased planning cost when there are no opportunities available. (3) It could use a Dynamic plan zone by starting with a Small plan zone when there are no nearby opportunities but increase the zone size when opportunities surface in the observation window. We next explore the tradeoffs of these three choices.

5 Evaluation

In this section, we assess three research hypotheses concerning the use of an expressive planning model and a goal reasoning system: (1) The use of a Dynamic planning zone by the goal reasoner will significantly reduce planning effort *when compared with always using a large planning zone*; (2) the use of PDDL+ to collect resources via events will significantly reduce planning effort over the numeric PDDL model for any zone size; and (3) the use of the Dynamic plan zone or PDDL+ will not significantly reduce resource collection.

Similar to previous ACTORSIM experiments for Minecraft, we randomly generate hallways consisting of 10 sections. Each section is 20 blocks wide and 15 blocks long, and contains one randomly placed element: a wood drop, a diamond drop, or a caged zombie.

The goal reasoning system selects for the planner a planning zone, defined as the number of cells away from the player in each of the following directions: (front, back, left, right). The *Small plan zone* of (5,1,3,3) has the benefit of a fast planning time at the expense of frequent replanning and possibly missing opportunities to collect resources. The *Large plan zone* of (8,3,8,8) reduces the number of replanning episodes at the cost of increasing the overall computational effort and planning time; however, it is less likely to miss opportunities. The *Dynamic plan zone* defaults to the Small zone until a resource is within 8 blocks, at which point it enlarges the plan zone in the direction needed to reach that drop. In effect, the goal reasoner is varying the replanning strategy as imposed by the plan zone. We leave more sophisticated zone-selection strategies for future work.

For this study, we limit the models and planning systems to focus on our research questions. For planning models, we focus on the numeric PDDL2.1 model that uses "visited" and "unsafe" predicates as well as the PDDL+ model that uses opportunistic goals. For a planner, we used the POPF planner extended for reasoning with PDDL+ processes and events (Coles and Coles 2014). This choice was motivated by the capabilities of the planner which considers all features of our models. Using the same planner for all models guarantees fair comparison between the different variants.

Table 1 summarizes 180 runs: 30 runs for each of the six zone/model combinations. The *S(N)*, *L(N)*, and *D(N)*

rows show the numeric PDDL2.1 runs, whereas rows denoted *S(+)*, *L(+)*, and *D(+)* summarize the PDDL+ runs for *Small*, *Large*, and *Dynamic* planning areas, respectively. The columns summarize the mean and standard deviation for each of the response variables: cumulative CPU time, number of nodes evaluated, and memory as well as the ratio of items collected versus those available in the hallway. A two-factor, paired-sample ANOVA for zone and course across each metric reveals that zone size significantly effects the results of each metric ($p \approx 0$ for all five metrics), justifying pairwise comparisons between the zone results. We now make a number of pairwise comparisons to examine how strongly the evidence supports our three research hypotheses. We report p-values for the Tukey Honest Significant Difference test ($\alpha = 0.05$) but we verified these results using Scheffe's method, which is more robust to potential effects of heteroscedasticity.

A dynamic planning zone reduces planning effort. The evidence suggests that the Dynamic plan zone can significantly reduce planning time while not using more memory. It is evident that in either the numeric or PDDL+ models there is 70-90 second difference in the CPU time between the Small and Large planning zones. As expected, the Dynamic plan zone is significantly different from the Large planning zone ($p < 0.0001$) and significantly similar to the Small planning zone ($p \approx 0.84$ for the numeric PDDL and $p \approx 0.99$ for PDDL+). In terms of nodes evaluated and memory, the Dynamic uses more nodes but less (or similar) memory than Small zone. However, the Dynamic zone never uses significantly more nodes or memory than the Large zone.

A PDDL+ model is less computationally intensive than a numeric PDDL2.1 model. The evidence is mixed. A PDDL+ model uses a statistically similar number of nodes as a PDDL2.1 model. It uses significantly more CPU time for the large zone but is similar for the other two zones. Finally, it uses significantly less memory for the Small zones and similar time for the Large and Dynamic zones. This is seen by comparing the Time, Nodes, or Memory usage for S(N) with S(+), L(N) with L(+), or D(N) with D(+).

The dynamic plan zone (as selected by goal reasoning) or PDDL+ do not reduce resource collection The evidence suggests that neither goal reasoning nor PDDL+ significantly reduces Alex's ability to collect resources. This question is answered by examining the ratios of wood and diamonds collected. At first glance, it may appear that these ratios are very different because the means vary so much. Examining the pairwise comparisons reveals a significant difference for the Small zone. Because there exists no significant difference between collection results for most of the zone/model combinations, it is clear that adding goal reasoning or PDDL+ to the system allowed it to maintain resource collection at the Large zone level.

Discussion The results show that goal reasoning can significantly reduce the planning time by a factor of 15-20. The results did not show that PDDL+ provided a further reduction in planning effort, but rather both models exhibited similar performance. Further, neither approach fared worse at resource collection than the Large zone, which was a baseline upper bound. However, the PDDL+ provides a much easier and more encompassing model for resource collection, which reduces the computational effort exerted by the goal reasoning system.

	Time		**Nodes**		**Memory**		**Wood**		**Diamond**	
	$\bar{x}$	s	$\bar{x}$	s	$\bar{x}$	s	$\bar{x}$	s	$\bar{x}$	s
S(N)	2.0	0.1	304	50	134	2	0.47	0.47	0.27	0.32
S(+)	3.4	0.5	312	44	110	9	0.25	0.28	0.28	0.29
L(N)	74.2	12.3	12439	21870	120	40	0.96	1.01	0.94	0.87
L(+)	93.4	13.0	7508	16175	104	31	0.65	0.33	0.64	0.38
D(N)	4.3	2.2	1804	3339	124	5	0.66	0.64	0.75	0.78
D(+)	3.5	0.8	446	204	124	9	0.68	0.77	0.64	0.74

Table 1: Summary of statistics for the hallway study.

6 Related Work

Perhaps the closest planning work related to our use of opportunistic/soft goals in an open world is the Open World Quantified Goals of Talamadupala et al. (2010), where quantified goals allow the planner to expand on goals that may appear during execution. An earlier work by Etzioni et al. (1997) used Local Closed-World statements to integrate an open world with a closed-world planner. In contrast to employing quantification or locality, a PDDL+ opportunistic goal is always in the problem's goal conjunction and a conditional event enables the goal.

A variety of research systems have been built to study Minecraft. The first, called BurlapCraft, by Abel et al. (2015) integrated the BURLAP machine learning platform[1] and examined how to use knowledge to select actions. More recently, Microsoft has released an open-source platform called Malmo[2] that provides extensive support for multiple programming languages, the ability to set up experiments, as well as support for reinforcement learning. ACTORSIM is primarily distinguished from these other systems in its use of goal reasoning, existing experiments using deep learning (Roberts et al. 2016c), and integration with other simulators including robotics platforms. Each system has merits depending on the task at hand, but none of these systems supported observations when we first examined them.

Recently, AI research in game-playing concentrated on exploiting Deep Learning techniques, particularly deep Q-networks (DQN) which beat expert human players on a range of ATARI games (Mnih et al. 2015). ATARI games have also been tacked using classical planning (Lipovetzky, Ramirez, and Geffner 2015). More advanced games such as Starcraft and Minecraft require a non-trivial transition to a much more complex environment. Recent work (e.g., (Bonanno et al. 2016; Tessler et al. 2016; Abel et al. 2015; Usunier et al. 2016)) shows promise in this area, though these studies examine simplified models or restricted subtasks. Massive training data sets, sparse rewards, vast state-action spaces, and difficult-to-define evaluation functions significantly limit the scaling potential and efficiency of DQNs.

[1] http://burlap.cs.brown.edu/

[2] https://github.com/Microsoft/malmo

In fact, Starcraft and Minecraft games are both prime examples of domains from automated planning (e.g. the Settlers domain (Long and Fox 2003)) where long-term, high-level goals need to be achieved.

Prior work has explored the use of PDDL to represent Minecraft (Branavan et al. 2012). This paper presents the first PDDL+ domains of Minecraft. Similar domains were previously defined in either purely propositional PDDL (IPC Committee 1998) or non-temporal PDDL2.1 with numeric fluents (Fox and Long 2003). PDDL+ is designed to compactly represent hybrid systems with mixed discrete/continuous behavior through processes and events. In recent years, PDDL+ planning has become a rising trend in AI and multiple approaches have been proposed to deal with PDDL+ domains (Shin and Davis 2005; Cashmore et al. 2016; Coles and Coles 2014; Della Penna et al. 2009; Piotrowski et al. 2016).

In the past PDDL+ was combined with Hierarchical Task Networks (HTNs) for goal-driven autonomy, implemented in the $\text{SHOP2}_{\text{PDDL+}}$ planner (Klenk et al. 2013; Molineaux et al. 2010). In contrast to hierarchical approaches, our work focuses on the first PDDL representation. We plan to incorporate more recent developments in hierarchical planning (e.g., (Ghallab, Nau, and Traverso 2016; Alford et al. 2016; Dvorak et al. 2014; Shivashankar et al. 2012)).

Soft goals are hard to express in PDDL. PDDL2.1 (Fox and Long 2003) introduced the notion of plan metrics enabling specification of soft goals and enhancing the quality of solutions. They could be useful in the context of assessing alternative plans. However, few planners actually employ plan metrics and they are limited to one specified optimizable function. For example, suppose each action modified the resources used or acquired, the character's change in health or food, and the number of steps to completion. Then, a planner could focus on producing diverse plans that span the trade offs. But plan metrics can only minimize or maximize the quality function which can exert unnecessary computational effort by collecting resources well beyond Alex's needs (and overloading Alex's inventory in the process).

On the other hand, PDDL3.0 (Gerevini and Long 2005) also incorporated plan metrics and combined it with the concept of *planning with preferences* which enable better reasoning with multiple objects and a more accurate method for specifying the desired plan characteristics. Planning with preferences is largely based on Linear Temporal Logic (LTL) (Pnueli 1977). PDDL3.0 provides a strong feature base for representing the Minecraft scenario. Numerical variables, plan metrics, and preferences are well-suited to build a concise and expressive Minecraft planning domain including maximizing collected resources when available. However, defining preferences in this manner can inflate the size of the domain and state variables sets, and significantly increase planning time.

These advanced features – plan metrics from PDDL2.1 and preferences from PDDL3.0 – could provide interesting and successful variants extending the modeling in this paper. The numeric PDDL model in this paper provides a solid foundation for assessing these features in Minecraft.

7 Closing Remarks

We have presented the first formulation of the Minecraft domain in PDDL+ extended with resource collection and zombie avoidance tasks. We also showed how events, a native feature of PDDL+, can be used to model opportunistic goals. Finally, we presented a goal reasoning approach to reducing the computational effort for finding a viable plan by selecting the zone size and partitioning the original problem into planning and observation zones. To the best of our knowledge this is the most extended model of Minecraft in AI, though it is only a preliminary stage of a larger project.

Future work will focus on modeling and solving the motivating example. It will include a revision of the plan zone and visibility, not only adjusting the size of the zone but also the shape which prunes areas of no interest while including areas with desirable resources and entities. We will also aim to expand the PDDL+ model to account for the continuous behavior in Minecraft, such as health management. Health decreases when attacked by hostile mobs, but regenerates slowly when resting. We will expand the list of happenings, actions, and goals to reason with and manage the agent's health. PDDL+ events are a natural fit for this modeling.

Our approach grew from a desire to accurately model health, food, resources, and entities in Minecraft, for which PDDL+ events and processes are best suited. It also provides a foundation for incorporating mixed discrete/continuous dynamics should this be desired in future domains. External happenings often modify the state of the environment without interference from the agent; we believe there is a great deal to learn from extending the model in this way.

A long-term focus of future work will enable the goal reasoning for long-duration autonomy. While it may be possible to manage short-term goals such as those in the motivating quest, we are interested in leveraging goal reasoning and automated planning for an agent that *perpetually learns* (Roberts et al. 2016a). Such an agent will need to manage its own learning agenda to master new tasks, revise previously learned tasks, and halt learning for already mastered tasks.

A final area of future work is in extending the model to incorporate Deep Learning. Minecraft was previously attempted using Deep Reinforcement Learning. We plan to compare the two approaches to identify their relative strengths. Deep Learning could manage short-term reactive behavior (i.e. self-defense) while planning with goal reasoning could manage long-term deliberative behavior.

Authors are sometimes circumspect about the actual development, design choices, and computational requirements of using a particular 'brand' of planning. In contrast, we have set out in this work to identify exactly our representational and design choices to the greatest extent possible. While it is unreasonable to expect a laymen to understand the nuances of automated planning, PDDL, or of effective domain modeling, we can at the least point to the active goal of the goal reasoning system, examine the domain or problem files produced, and examine the search trace of the planner to explain its decision. The transparency of the approach we have outlined is especially noteworthy in an era where AI systems are being called to arrive at sensible and correct output while also making transparent their decision-making process.

Minecraft presents worthwhile challenges for studying planning with respect to a simulated environment. As we moved from a propositional representation to a more detailed PDDL+ representation, the planners available to us diminished considerably – from close to a hundred to less than a few. Were we to move in the direction of PDDL3.0 features, a similar problem would occur. Similarly, few planners support advanced PDDL2.1 features such as metrics beyond action cost. Our findings underscore the need for the continued advance of planning systems – perhaps through the competitions – to better support the range of PDDL features used by applications.

Acknowledgments

We thank the anonymous reviewers whose comments improved this paper. MR, DA, and PB thank NRL for supporting this research.

References

Abel, D.; Hershkowitz, D. E.; Barth-Maron, G.; Brawner, S.; O'Farrell, K.; MacGlashan, J.; and Tellex, S. 2015. Goal-based action priors. In *ICAPS*, 306–314.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *IJCAI*. AAAI Press.

Bonanno, D.; Roberts, M.; Smith, L.; and Aha, D. W. 2016. Selecting Subgoals using Deep Learning in Minecraft: A Preliminary Report. In *IJCAI Workshop on Deep Learning for Artificial Intelligence*.

Branavan, S.; Kushman, N.; Lei, T.; and Barzilay, R. 2012. Learning high-level planning from text. In *Proc. of the Annual Meeting of the Assoc. for Comput. Linguistics*, 126–135.

Cashmore, M.; Fox, M.; Long, D.; and Magazzeni, D. 2016. A Compilation of the Full PDDL+ Language into SMT. In *ICAPS*, 583–591.

Coles, A. J., and Coles, A. I. 2014. PDDL+ Planning with Events and Linear Processes. In *ICAPS*, 74–82.

Della Penna, G.; Magazzeni, D.; Mercorio, F.; and Intrigila, B. 2009. UPMurphi: A Tool for Universal Planning on PDDL+ Problems. In *ICAPS*, 106–113. AAAI.

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A flexible ANML actor and planner in robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS), Portsmouth, United States*.

Etzioni, O.; Golden, K.; and Weld, D. S. 1997. Sound and efficient closed-world reasoning for planning. *AIJ* 89(1):113 – 148.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.

Fox, M., and Long, D. 2006. Modelling Mixed Discrete-Continuous Domains for Planning. *JAIR* 27:235–297.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. In *The Language of the 5th IPC. Technical Report, Department of Electronics for Automation, University of Brescia, Italy*, volume 75.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge Univ. Press.

IPC Committee. 1998. PDDL : the planning domain defnition language. Technical report, Yale Center for Computational Vision and Control. The committee consisted of: M. Ghallab and A. Howe and C. Knoblock and D. McDermott and A. Ram and M. Veloso and D. Weld and D. Wilkins.

Klenk, M.; Molineaux, M.; and Aha, D. 2013. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence* 29(2):187–206.

Lipovetzky, N.; Ramirez, M.; and Geffner, H. 2015. Classical planning with simulators: results on the atari video games. In *IJCAI*, 1610–1616.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *JAIR* 20:1–59.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; and Ostrovski, G. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Molineaux, M.; Klenk, M.; and Aha, D. W. 2010. Goal-driven autonomy in a navy strategy simulation. Technical report, DTIC Document.

Piotrowski, W.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic Planning for PDDL+ Domains. In *IJCAI*, 3213–3219.

Pnueli, A. 1977. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, 46–57. IEEE.

Roberts, M.; Hiatt, L. M.; Coman, A.; Choi, D.; Johnson, B.; and Aha, D. 2016a. Actorsim, a toolkit for studying cross-disciplinary challenges in autonomy. In *Fall Symposium on Cross-Disciplinary Challenges in Autonomy*.

Roberts, M.; Shivashankar, V.; Alford, R.; Leece, M.; Gupta, S.; and Aha, D. 2016b. Goal reasoning, planning, and acting with ActorSim, the actor simulator. In *Proceedings of ACS*.

Roberts, M.; Alford, R.; Shivashankar, V.; Leece, M.; Gupta, S.; and Aha, D. W. 2016c. ACTORSIM: A toolkit for studying goal reasoning, planning, and acting. In *Working notes of the ICAPS PlanRob Workshop*.

Shin, J.-A., and Davis, E. 2005. Processes and Continuous Change in a SAT-based Planner. *AIJ* 166(1):194–253.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, volume 2, 981–988. Int. Found. for AAMAS.

Talamadupula, K.; Benton, J.; Kambhampati, S.; Schermerhorn, P.; and Scheutz, M. 2010. Planning for human-robot teaming in open worlds. *ACM TIST* 1(2):14:1–14:24.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D. J.; and Mannor, S. 2016. A Deep Hierarchical Approach to Lifelong Learning in Minecraft. *arXiv preprint arXiv:1604.07255*.

Usunier, N.; Synnaeve, G.; Lin, Z.; and Chintala, S. 2016. Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks. *arXiv preprint arXiv:1609.02993*.

Towards Planning With Hierarchies of Learned Markov Decision Processes

John Winder, Shawn Squire, Matthew Landen, Stephanie Milani and **Marie desJardins**
{jwinder1, ssquire1, mlanden, stemila1, mariedj}@umbc.edu
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD 21250

Abstract

Decision-making agents face immensely challenging planning problems when operating in large environments to solve complex tasks. A hierarchy of abstract Markov decision processes (AMDPs) provides a framework for decomposing such problems into distinct, related subtasks or subgoals. AMDP hierarchies (Gopalan et al. 2017) grant considerable speedup over related recursively and hierarchically optimal methods such as MAXQ and options. Each AMDP acts as a subgoal, and each is itself a planning problem with a local model and state space abstracted from a ground MDP. Currently, agents are able to plan more efficiently by using a reduced state space at the appropriate level of abstraction; however, they require their subtask models to be specified by a human expert (Gopalan et al. 2017). We describe an approach for automating model estimation by combining the R-MAX algorithm with AMDPs. We compare the resulting structures, R-AMDPs, with a similar approach, R-MAXQ (Jong and Stone 2008), and motivate its advantages. Ultimately, R-AMDPs represent the first step in learning AMDP hierarchies dynamically, completely from an agent's experience.

1 Introduction

For decision-making agents operating in large, rich environments such as factory floors or kitchens, the resulting planning problems are often extremely challenging due to the immense number of states and the need to achieve a complex, precise sequence of actions (Bollini et al. 2012; Knepper et al. 2013). Typically, the state–action space of these domains grows combinatorially with the number of objects present in the environment. Standard planning algorithms require an agent to explore this space at its lowest level, resulting in a difficult search over long sequences of actions.

For example, a taxi-driving agent might be required to navigate to pick up a passenger and then drop them off at a specific landmark of the passenger's choosing. Planning optimally over the lowest level of actions to deliver the passenger to their desired landmark could necessitate searching over all possible states and considering all of the navigational actions needed to reach each possible landmark. As the number of landmarks and size of the world increases, the complexity of action sequences likewise increases combinatorially, resulting in an increasingly difficult search problem.

The concept of hierarchical planning and reinforcement learning aims to partition such problems into subtasks, allowing agents to reason about relevant objects and actions at the appropriate level of abstraction and context. This idea is motivated by the ways humans tend to think of and describe solutions for complex tasks, especially for the eventual goal of human-agent interaction through natural language commands. For the taxi example, when provided with only a general instruction like "take this passenger to that destination," the agent ought to execute the optimal path to solve that goal. It is unappealing for the agent to require unnecessarily detailed instructions, such as "go north one step, go west one step" to solve the task. Instead, the global goal can be decomposed hierarchically into subtasks (e.g., "get the passenger" and "put the passenger to the destination"), which in turn consist of atomic, grounded actions. The agent can take advantage of these intermediate representations to more readily repeat certain useful patterns of behaviors. These subtasks serve as reusable skills that help the agent traverse state space more effectively and can transfer to new tasks in a related domain. With a hierarchical decomposition, any given subtask is smaller in state-action space than the original; often, this translates into a decision-making problem that is easier to solve overall. Moreover, learned subtask policies may be shared among parent tasks, immediately facilitating knowledge transfer. Employing hierarchies for planning, thus, dually aids the issues of representation and computation, allowing agents to store and reuse learned behaviors to solve new domains more rapidly.

A recently developed method of planning with a hierarchy of abstract Markov decision processes (AMDPs) presents a promising approach to decomposing a domain into a graph of independent subgoals (Gopalan et al. 2017). While AMDPs are appealing for improved planning in complex domains, they require a large amount of expert knowledge to design: the models must be fully specified at every level by a designer, which is not always feasible or suitable. Ultimately, agents should learn the structure and models of these hierarchies from data and experience without relying on a designer. To that end, we present a novel extension of AMDPs called R-AMDPs that uses the R-MAX method of optimistic model estimation to approximate the transition

and reward functions for all subtasks.

2 Background

A Markov decision process (MDP) is a standard structure for describing a decision-making problem. An MDP is represented as a five-tuple, $\{\mathcal{S}, \mathcal{A}, \mathrm{T}, \mathrm{R}, \mathcal{E}\}$, respectively consisting of a set of states, a set of actions, the transition probability function ($\mathrm{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$), a reward function ($\mathrm{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$), and a set of terminal states ($\mathcal{E} \subset \mathcal{S}$) from which there are no further actions. Reinforcement learning (RL) is a paradigm where an agent lacks direct access to T or R, while planning problems typically assume an agent knows these precisely. A *domain* describes a distribution of related MDPs from which a specific task MDP is drawn.

An object-oriented MDP (OO-MDP) (Diuk, Cohen, and Littman 2008) is designed to facilitate learning in domains with large state spaces. It extends the traditional MDP model by representing the domain as a set of object classes, $\mathcal{C}$, each with a set of predefined attributes, $Att(C)$. An object, o, belongs to a single object class and has values assigned for each attribute associated with the class. The state, s, of a domain is defined as the union of the attribute values for all objects in the domain, $s = \bigcup_{i=1}^{\mathcal{O}} \text{state}\,(o_i)$ where $O = \{o_1, o_2, \ldots, o_n\}$ for n objects in the domain.

This factored representation gives several benefits over an MDP that represents states symbolically or as a vector of features alone. First, OO-MDPs can more easily represent the relevance of objects and classes to specific goals and actions. Second, objects instantiated differently but with equal values (i.e., given objects o_1 and o_2 of object class c, $o_1 = o_2 \iff \forall a \in Att(c) : o_1.a = o_2.a$) are considered equivalent, so they may be handled interchangeably. Additionally, propositional functions, while not required, can be declared over objects to capture relations among them or global properties of states. Finally, as a standardization of state representation, the OO-MDP framework allows for greater extensibility and ease of implementation.

The **R-MAX** algorithm (Brafman and Tennenholtz 2002) describes a model-based reinforcement learning approach where an agent maintains an approximate representation of both T and R that is iteratively updated and improved. The agent plans with this learned model, following the principle of optimism under uncertainty. R-MAX encourages exploration by initializing unknown rewards with a high positive value (R_{MAX}, the upper bound of R), and the transition approximations as having a probability of zero. An R-MAX agent continues to behave optimistically, recording the occurrences of observed rewards and transitions, until some threshold is reached and the respective reward or transition is considered "known." R-MAX typically finds an optimal or near-optimal policy and allows agents to generalize more rapidly to similar MDPs from the same domain.

MAXQ (Dietterich 2000) is a hierarchical reinforcement learning method for decomposing a value function for a policy into subtasks corresponding to subgoals identified by the human designer. Given an MDP M, MAXQ decomposes M into a set of n MDPs $\{M_0, M_1, \ldots, M_n\}$, one for each subtask. These MDPs are organized into a "task hierarchy," a directed acyclic graph which shows the relationship from a root (global) goal to child tasks down to leaf tasks that use the primitive actions of M. MAXQ uses the idea of a completion function to represent the expected discounted reward of current task i after completing subtask (or action) a while in state s, $C(i, s, a)$. Then, action-value is decomposed recursively, $Q(i, s, a) = V(a, s) + C(i, s, a)$, in terms of the value of the state for the child a (which depends on Q at that lower level) plus the completion function of the respective task. MAXQ achieves a recursively optimal solution, where the policy is optimal at each level of the hierarchy. Other frameworks, such as SMDPs (options) (Sutton, Precup, and Singh 1999), provide a hierarchically optimal solution that achieves the maximum reward at the base level (Dietterich 2000). Therefore, plans in MAXQ can be suboptimal from a global perspective. However, in trading total optimality for near-optimal solutions, recursively optimal methods can offer significantly faster planning times.

R-MAXQ (Jong and Stone 2008) combines the hierarchical decomposition of MAXQ with the optimistic model-based exploration of R-MAX. The algorithm simultaneously estimates the reward and transition functions of a ground MDP, computing the model dynamics of higher-level tasks recursively up from that task's descendants. After the knowledge of the ground MDP is propagated up the task hierarchy, the agent is able to plan at an abstract level using a Greedy-Q policy. After a bounded amount of exploration, the agent's hierarchical model will converge to a nearly optimal solution. This learning algorithm will serve as a baseline to compare the new approach described in Section 3.

Algorithm 1 Planning with an AMDP hierarchy

```
function SOLVE(H)
    GROUND(H, ROOT(H))
function GROUND(H, i)
    if i is primitive then                          ▷ recursive base case
        s'_0 ← EXECUTE(i)                           ▷ obtain next ground state
    else
        s'_0 ← AMDP-PLAN(H, i)
return s'_0
function AMDP-PLAN(H, i)
    s_i ← F_i(s)                                    ▷ project the environment state s
    π ← PLAN(s_i, i)
    while s_i ∉ E_i do                              ▷ execute until termination
        a ← π(s_i)
        j ← LINK(H, i, a)                           ▷ a links to child task j
        s'_0 ← GROUND(H, j)
        s_i ← F_i(s'_0)
return s'_0
```

An abstract Markov decision process (**AMDP**) (Gopalan et al. 2017) hierarchy is a framework for decomposing a complex planning problem into a series of actionable subtasks. Given some ground MDP to solve, an AMDP hierarchy is represented as a graph similar to MAXQ. The root corresponds to the global goal of the MDP, the leaves repre-

sent primitive actions to be executed in the ground MDP, and all other nodes are subtasks that function as the actions of the parent nodes. A key difference when compared to MAXQ tasks is that each AMDP task node is itself a complete MDP possessing its own locally (not recursively) defined reward and transition functions.

An AMDP is an MDP where each state is an abstracted representation of the ground MDP. Formally, an AMDP is a six-tuple $\{\tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \tilde{\mathrm{T}}, \tilde{\mathrm{R}}, \tilde{\mathcal{E}}, F\}$, consisting of the OO-MDP components with the addition of a *state projection* function of $F : \mathcal{S} \rightarrow \tilde{\mathcal{S}}$ for mapping states from the ground MDP into the abstract state space of the AMDP. $\tilde{\mathrm{T}}, \tilde{\mathrm{R}}, \tilde{\mathcal{E}}$ are unique to the given AMDP, and $\tilde{\mathcal{A}}$ consists of the AMDP's child subtasks (either primitive actions or other AMDPs). We define the hierarchy of AMDPs as $H = (V, E)$, a directed acyclic graph in which the set of vertices V contains the AMDPs and primitive actions of the ground MDP, and the edges E express membership in an AMDP's action set. Pseudocode for planning with AMDP hierarchies is given in Algorithm 1

AMDP hierarchies produce optimal policies at each AMDP. They are analogous to MAXQ hierarchies, and thus share the benefit of being recursively optimal under the condition that the local state abstraction, reward, and transition functions are correct. AMDP hierarchies enable the agent to plan only for subgoals that help achieve the main task without computing plans for irrelevant subgoals, and allow planning in stochastic environments. The biggest limitation of AMDPs has been that the structure and components (the hierarchy, task nodes, mapping function, task reward functions and transition probabilities) must be specified by an expert. Thus, we are working towards the first step in learning AMDP hierarchies: R-AMDPs.

3 Approach

AMDPs with R-MAX (R-AMDPs) are a natural, model-based extension to AMDP hierarchies, analogous to R-MAXQ with respect to MAXQ. For this technique, each R-AMDP in the hierarchy follows the R-MAX approach of building a model using a maximum-likelihood estimation of its $\tilde{\mathrm{T}}$ and $\tilde{\mathrm{R}}$. Thus, R-AMDPs learn the (unseen) model dynamics at abstract levels, and use these to plan normally, as described in Section 2

Following from R-MAX, every R-AMDP maintains approximations of $\tilde{\mathrm{T}}$ and $\tilde{\mathrm{R}}$ that are updated after the GROUND subroutine in the AMDP-PLAN function of Algorithm 1. For R-AMDP i, each time a transition (s, a, r, s') is obtained, the reward estimation $\tilde{r}_i : \tilde{\mathcal{S}}_i \times \tilde{\mathcal{A}}_i$ is updated by $\tilde{r}_i(s, a) \leftarrow \tilde{r}_i(s, a) + r$, the approximate transition probability $\tilde{t}_i : \tilde{\mathcal{S}}_i \times \tilde{\mathcal{A}}_i \times \tilde{\mathcal{S}}_i$ is incremented, and the counter of state-action pair observations $n_i : \tilde{\mathcal{S}}_i \times \tilde{\mathcal{A}}_i$ is also incremented. When $n_i(s, a)$ reaches a threshold of state-pair visits m_i, the rewards and transitions are considered "known," and thereafter the model is updated with each transition:

$$\tilde{\mathrm{R}}_i(s, a) \leftarrow \tfrac{\tilde{r}_i(s,a)}{n_i(s,a)} \quad \text{and} \quad \tilde{\mathrm{T}}_i(s'|s, a) \leftarrow \tfrac{\tilde{t}_i(s,a,s')}{n_i(s,a)}.$$

Until the model is known, optimism is used for rewards and all transitions are assumed to have zero probability*:

$$\tilde{\mathrm{R}} = R_{\text{MAX}} \ \ \forall\, s \in \tilde{\mathcal{S}}, a \in \tilde{\mathcal{A}},$$
$$\tilde{\mathrm{T}}_i(s'|s, a) = 0 \ \ \forall\, s, s' \in \tilde{\mathcal{S}}, a \in \tilde{\mathcal{A}}.$$

Thus, each R-AMDP will initially plan by favoring to explore unknown rewards and transitions until they are learned.

We hypothesize that the R-AMDP algorithm effectively trades space (storing the task models) for the favorable properties of AMDP hierarchies, including faster planning and pruning of the search space. Additionally, R-AMDPs learn the abstract rewards and transitions without needing to recurse through multiple layers of abstraction to reach the ground MDP, as R-MAXQ necessitates. Moreover, by computing explicit models for each task, there is greater encouragement of exploration at all levels. That is, R-AMDPs are not just guided to infrequently visited transitions defined at the lowest level, but are directed to explore optimistically for every abstract subgoal.

Note that while the mapping functions F may eventually be learned, for now they are assumed to be provided by an expert. This requirement is still reasonable because each F typically consists solely of aggregating, removing, or otherwise reducing the base MDP state space, a task much less demanding on designers than specifying the full dynamics of $\tilde{\mathrm{T}}$ and $\tilde{\mathrm{R}}$ for all AMDPs. Ultimately, our goal is to learn the hierarchies completely from data, where an agent explores and discovers subtasks or is trained to solve subtasks separately and learns to compose them in an (R-)AMDP hierarchy on its own. R-AMDPs, thus, are preferable to AMDPs, and are a step closer to learning abstract task hierarchies for complex tasks from data.

Towards Scoring Hierarchies

To learn hierarchies from data, metrics for ranking will be needed. The optimal metric to rank the success of a R-AMDP hierarchy versus another hierarchy with an identical root goal condition is the error between the learned policy and the optimal policy; however, measuring this loss is non-trivial because it requires knowledge of the optimal policy, which is impractical and intractable in complex domains. Instead, we compare two hierarchies using the number of states explored as a metric. More state exploration can indicate a hierarchy is suboptimal because unnecessary actions result in an increased number of states explored.

Another important aspect of R-AMDP hierarchies is generalization: one hierarchy ought to be sufficient for solving many goals in the environment domain. Testing generalizability involves producing multiple goal conditions that may be solved in the base MDP without any hierarchy and applying the goal conditions to the R-AMDP. A properly designed R-AMDP should be capable of solving all tasks possible in the ground MDP, so failure to complete some or all tasks marks it as an incorrect hierarchy. Since R-AMDPs are learned, it is possible to over-fit the hierarchy, which reduces effectiveness for solving general problems.

*In implementation, a hypothetical terminal state is added to state space. All transitions lead to it until m_i is reached.

Finally, a simple but imprecise metric is computational run-time. One of the major benefits of a R-AMDP is reduced run-time on complex domains, so a better R-AMDP hierarchy should plan and solve a domain faster. Taken together, these metrics provide direction for assessing the structural suitability, accuracy, efficiency, and simplicity of R-AMDP hierarchies.

4 Methodology

To evaluate the performance of R-AMDPs, we will use the classic Taxi domain (Dietterich 2000). Taxi is a discrete environment with a taxi agent, passengers, depot locations, and impassable walls positioned on a constrained map. The objective is to deliver a passenger from their source location to their goal location, both of which are any of the depots on the map. The taxi may move in any cardinal direction that is not blocked by a wall, pickup a passenger when they share the same position on the map, or drop a held passenger when the taxi is at a depot.

Taxi benefits greatly from hierarchical planning, since the root objective may be decomposed straightforwardly into several "layers" of subgoals. The children task nodes for the root, GET and PUT, are parameterized over the taxi and the depot location, abstracting away the Cartesian coordinate attributes of the taxi and depots. This abstraction greatly reduces the state space over which the agent must reason. Planning is simplified at this level to two subtasks: GET a passenger from a source depot and PUT them at a goal depot. In addition to the primitive actions PICKUP and PUTDOWN for GET and PUT, respectively, both tasks may also NAVIGATE to any of the depots on the map. Because the sole knowledge required for NAVIGATE is whether the agent is or is not at a depot, the exact Cartesian coordinates are extraneous and, thus, are abstracted away. The children of each NAVIGATE task node are the atomic movement actions that operate in the base MDP. The straightforward abstractions present in this domain make it an ideal problem to solve with AMDPs and, thus, R-AMDPs. The AMDP hierarchy for Taxi has been formalized in previous literature (Gopalan et al. 2017).

The primary baseline for R-AMDPs is the R-MAXQ algorithm (Jong and Stone 2008), which combines the model-based exploration R-MAX with the abstractions of the MAXQ framework. R-MAXQ provides a similar structure to R-AMDPs, since both use a MAXQ structure. R-MAXQ maintains a model solely for primitive actions; changes in the model propagate up to the estimates of higher-level composite actions. In contrast, each R-AMDP (as a complete MDP in itself) keeps an independent, self-contained model (learned via R-MAX). Each R-AMDP is responsible only for approximating the model defined over its immediate state space and child subtasks, saving it from the need to recursively compute transition and reward functions from all nodes to the ground. Additionally, R-AMDPs also receive all the benefits of state abstraction, but R-MAXQ does not leverage the state abstraction benefits granted by standard MAXQ. Therefore, we expect to see better performance for complex tasks than R-MAXQ.

5 Conclusion

In this work, we propose and motivate an extension to AMDPs, R-AMDPs, which automatically learn the transition functions and reward functions from the base MDP and the given abstraction function, F. R-AMDPs provide the main benefits of abstraction and planning with AMDPs, while relaxing some of the constraints for creating an AMDP. The end goal of this work is for an agent in a complex domain to be able to construct an AMDP task hierarchy and then learn the rewards and transitions at each level. After the agent develops this stratified representation of the domain, it can generate plans more efficiently and transfer knowledge between similar tasks.

In the future, R-AMDPs will be extended to learn the abstraction function F in addition to the transition and reward functions, uniquely enabling R-AMDPs to create their own structure automatically with minimal expert knowledge required. Finding a method to score candidate AMDP graphs for ranking generated R-AMDPs is a necessary step for completely autonomous AMDP generation. R-AMDPs provide the initial backbone for autonomous planning in complex domains.

References

Bollini, M.; Tellex, S.; Thompson, T.; Roy, N.; and Rus, D. 2012. Interpreting and executing recipes with a cooking robot. In *International Symposium on Experimental Robotics*.

Brafman, R. I., and Tennenholtz, M. 2002. R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research* 3:213–231.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* 13:227–303.

Diuk, C.; Cohen, A.; and Littman, M. L. 2008. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*.

Gopalan, N.; desJardins, M.; Littman, M. L.; MacGlashan, J.; Squire, S.; Tellex, S.; Winder, J.; and Wong, L. L. 2017. Planning with abstract Markov decision processes. In *27th International Conference on Automated Planning and Scheduling*.

Jong, N. K., and Stone, P. 2008. Hierarchical model-based reinforcement learning: R-max+ MAXQ. In *Proceedings of the 25th International Conference on Machine Learning*.

Knepper, R.; Tellex, S.; Li, A.; Roy, N.; and Rus, D. 2013. Single assembly robot in search of human partner: Versatile grounded language generation. In *ACM/IEEE International Conference on Human-Robot Interaction Workshop on Collaborative Manipulation*.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112(1-2):181–211.