Proceedings of ICAPS 2020 workshop on Integrated Planning and Execution

Integrated Planning, Execution and Goal Reasoning for Python

Rich Levinson
BrainAid.com
rich@brainaid.com

Abstract

Integrated planning and execution systems often combine a planner using a declarative action representation with an execution system which uses a procedural "reactive" representation. This action representation mismatch creates a language barrier resulting in computational friction for transitions between planning and execution. It may limit the scope of the planner's model or restrict execution system flexibility, and lead to inflexible transitions and division of labor between planning and execution.

We present a library for embedding unified planning and execution, with goal reasoning, into Python code. The library extends Python to include choice points and a goalstack, and provides a message-based interface between domain level Python code and supervisory meta-reasoning processes. The supervisory processes search a space of program variations defined by choice points and manage flexible transitions between execution and planning. The planner and the execution system share the same Python procedures extended with choice points as their action representation. They exchange choice point rules which inform each other of their choices and goal success or failure outcomes.

Motivation

Neuropsychological models of human executive functions are a key motivation for our model of integrated planning and execution. We are inspired by the fact that in humans, planning and execution activate the same neural circuits. There are not separate circuits for planning and executing the same behavior. We interpret that as planning and execution sharing the same activity model (neural circuits for people, and software for machines). We aim to model complex actions and behaviors with the full procedural expressiveness of modern general-purpose programming languages (the most expressive action representations we are aware of).

Our approach aspires to approximate the way planning and execution are integrated in human executive functions (Levinson 1994; Levinson 1995b; Levinson 1995c). Our company has helped hundreds of patients with a wide range of executive function disorders to maximize their autonomy (Levinson 1997; Levinson et al 2007, Modayil et al. 2008, Levinson et al 2009; Chu et. al., 2012;). This experience and extensive neuropsychological research (Barkley 2012; Lezak et al. 2012) provide strong evidence that human autonomy relies on fluid transitions and dynamic balancing between planning and execution. Imbalances take many forms, ranging from inattentive and inactive, to highly distractible, impulsive and reactive, to obsessive-compulsive.

This work is part of an ongoing effort to understand autonomy as a dynamic balancing act between planning and reaction. We are interested in exploring the kind of balance between planning and execution which is a hallmark of the healthy executive functions required for human autonomy. Of particular interest is the case when the execution system has some level of default reactive competence and doesn't strictly require the planner for every situation. Given enough time, one can usually improve hardwired reactions to cover more situations, but what is the best division of labor between reflexive reaction and deliberation, and does it change in different situations?

Autonomous systems which integrate planning with reactive execution often use different action representations for the planning and execution components. For example, NASA has developed systems combining declarative Mixed Integer Program formulation with procedural PLEXIL execution (Aaseng et al. 2018; Levinson 2019). Other systems such as ROSplan provide an interface for dispatching plans to external executors (Cashmore et al. 2015). A number of execution-only systems have been developed such as PRS (Ingrand et al. 1996) and PLEXIL (Verma et al. 1995, Verma et al. 1996) which may accept input plans which are either hand-coded or generated by external planners.

This *language barrier* between planning and execution models increases complexity due to the need to translate between the two languages and maintain two different behavior models. In such hybrid systems, the planner cannot help when execution strays outside of the planner's model. To address this, some systems use the same representation for planning and execution, but force execution to use the planner's (declarative) model rather than force the planner to use the execution system's (procedural) model. Examples of this include IDEA (Muscettola et al. 2000, Muscettola et al. 2008), and KIRK/RMPL (Kim et al. 2001).

Two approaches which do use a shared procedural representation for planning and execution are Propel (Levinson 1995a) and operational models with RAEplan (Patra et al. 2019). The work presented here follows that direction.

Architecture Overview

Our approach to minimizing the language barrier involves adding choice points to a general programming language and using it as the action representation for both planning and execution. The system is called **Propel: The Program Planning and Execution Language** (Levinson 1995a).

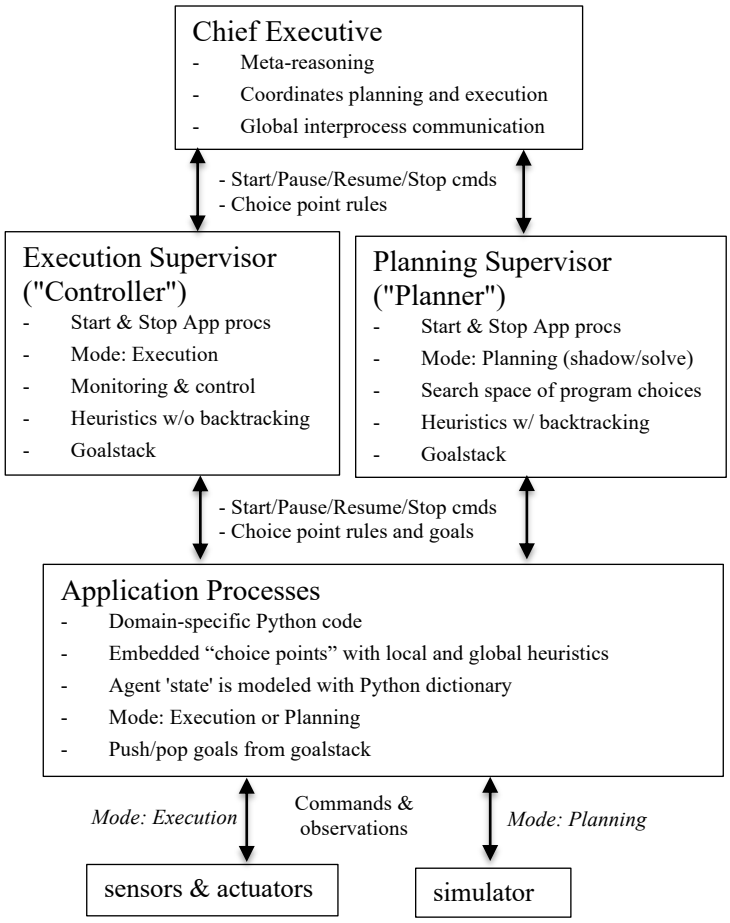

Figure 1: Propel's three levels of processes: *Application, Supervisor, and Chief Executive*.

The original version of Propel added choice points to LISP and laid out the core framework: a general programming language is augmented to include choice points, defining a search space of program variations, along with a message-based interface to supervisory processes including a planner. It demonstrated our original hypothesis that anytime planning can increase execution robustness by extending the execution operating condition range. The second version of Propel (Levinson 2005), added choice points to C++, included a simple temporal network (Dechter et al. 1991), and was targeted at the domain of autonomous software repair. It addressed another motivation: to increase the scope of the planner's model to include the execution system software.

This paper presents Propel 3, dubbed *PropelPy*, a new version which adds choice points to Python and which has been extended to support a simple form of goal reasoning (Aha 2018). PropelPy is a Python library which enables integrated planning, execution and goal reasoning to be added to Python programs. Existing Python code might even be retrofit by replacing current deterministic choices with choice points to create a search space for the planner to explore when the current deterministic choice fails. PropelPy design objectives include: (a) place a minimal learning curve on Python Programmers with minimal AI training, (b) provide a tool for experienced AI practitioners who seek a minimally restrictive (maximally expressive) action representation and more flexible and tightly integrated planning and execution, and (c) provide a tool to explore how balance variations between planning and execution compares with human executive dysfunction.

Figure 1 shows the three process levels within Propel: The *Application, Supervisor*, and *Chief Executive* levels. The Application level contains all of the domain-specific code. The Supervisor level contains meta processes which monitor and manipulate the application processes. The Execution Supervisor is called the *Controller* and the Planning Supervisor is called the *Planner*. The planner searches through choice point space which may involve backtracking. The execution system does not search or backtrack. It uses heuristics to make reactive choices and doesn't strictly require input from the planner. The planner and controller exchange *choice point rules* to inform each other about choices and outcomes. The Chief Executive monitors and manipulates the supervisors. We use the term "Controller" to distinguish the Execution Supervisor's middle-management role from the Chief Executive.

The **Application level** processes execute Python code extended with choice points, heuristics and integration with supervisory processes. The choice points embedded in application code describe nondeterministic assignment statements, defining the planner's search space of program variations. Choices may be a list of any Python data type including class objects and dictionaries.

Search nodes: Each node in the search tree corresponds to an application-level process which spawns a child process for each choice at a choice point. Each tree branch corresponds to the parent process/node splitting into children for each choice. Each node represents a unique program variation based on the sequence of choices (branches) from the root to the node. Each node has information only about its own computational context and reports its computational state, choices and outcomes to its supervisor.

Each node has its own copy of the agent's state dictionary, called the *"node state"*, which corresponds loosely to the agent's "belief" state. The node state has no predefined structure other than being a Python dictionary. It's a black box to the planner which has no domain specific knowledge. The node state contains any data required by the app level to detect goal success and failure states, and to provide inputs to app-level provided heuristic methods.

Node Mode: Each node has a mode which is *execution* or *planning*. The Planner manages the tree of planning nodes. It starts/stops the application level code in planning mode. The Controller runs the same application code in execution mode to manage the execution nodes.

Simulating Primitive Actions: Application code may branch based on the node mode so that "primitive" actions may send out physical actuator commands in execution mode but simulate those commands in planning mode.

The **Supervisor level** includes an Execution Supervisor and a Planning Supervisor. The supervisors perform runtime monitoring, verification, and recovery of application level processes. The planner manages search through the space of application level nodes. The planner and controller are supervisor twins, running nearly the same code to monitor and manipulate the same application-level Python code. They both send commands to start, pause, resume and stop the application processes. However, there are differences between them. The planner can backtrack through the search space, but the controller cannot. Controller decisions are commitments to act which may change the environment, while planner decisions don't change the environment. The controller never backtracks and uses a single process. The planner creates multiple processes.

Execution and planning processes communicate via state-action rules which provide choice point advice. The rule condition is a node's computational state (process control stack) and node state at the choice point and the rule action is a choice and an outcome (goal success or failure).

The **Chief Executive level** is a single process that manages the Planner and Controller. Its primary role is to manage the transitions between execution and planning. It sends start, pause, resume and stop messages to the Planner and Controller, and passes rules between them.

Heuristic functions and reactive competence: The application level may provide domain-specific *local* and *global* heuristics, sorting functions which are passed to the Planner for search control. In execution mode, the local heuristics define the default execution behavior. This provides a default *reactive competence* (Drummond et al 1993) which may be augmented by planner advice.

Example: Collecting Rocks

We now introduce the example application which will be used throughout the paper. This example was specifically designed to motivate and drive the Goal Reasoning extensions to Propel. Our example is a rover which must find and pick up rocks and deliver them to one of the designated delivery points depending on the rock type (either triangle or diamond). Figure 2 shows an example initial state. The rover is the green circle on the right. The orange cells are recharging stations and the one on the right, where the rover is, is also "home". The black squares are obstacles, blocked cells where the rover cannot go. The red cell with a triangle in the top-right quadrant is the delivery location for triangles. The red cell with a diamond in the lower-right is the delivery station for diamonds. The problem may be varied by changing the density of obstacles and rocks.

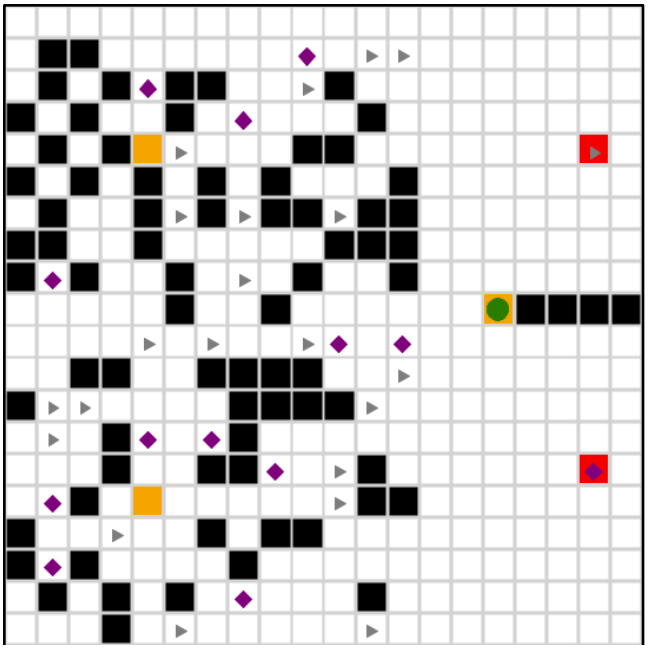

Figure 2: An Initial State

Goal Reasoning: The rover maintains a goalstack for picking up and delivering each rock. Initially the rover sees only the single closest rock, and has a single goal to pickup and deliver that rock. As it moves, it sees any new rocks within 4 cells of its current location. Whenever the rover sees a new rock, it adds it to a list of visibleRocks. After it finishes delivering a rock, it pops the goal for that rock, chooses a new visible rock and pushes a new goal to pickup and deliver that rock.

G3: GoToCell (cLoc) Supergoal: None
G2: GoToCell(pLoc), Precond: R2 @(pLoc), Supergoal: G1
G1: PickupAndDeliver R2 from (pLoc) to (dLoc), Subgoal: G2

Figure 3: Example goalstack

Figure 3 shows an example goalstack. The initial goal G1 is to PickupAndDeliver rock R2 from pickup location *pLoc*, and deliver it to *dLoc*. G1's first subgoal is G2, to GoTo pickup location pLoc. The top goal, G3, is a contingent goal to GoTo the charging station (cLoc) when the energy dips below a minimum threshold.

All three of Propel's process levels have been extended to support goal stack management. *Application* level code is responsible for adding and removing goals on the goal stack. Goals may have preconditions which are domain-specific Boolean Python methods that take the node state as input. The application level informs the supervisor level when goals are added or removed.

At the *Supervisor* level, the Planner associates each search node with a goal and only selects nodes associated with goals currently on the goal stack.

The *Chief Executive* level manages transitions between planning and execution when goals succeed or fail. For example, when the Chief Executive is informed of an execution failure, the planner is called to solve the top goal on the goal stack, and then return the choice point rules for achieving that goal back to the controller for execution.

Contingent actions push new goals onto the stack based on dynamic state conditions. For example, sometimes the rover must preempt its current goal and go to the recharging station. The rover has a limited amount of energy which is consumed by each step and consumption is increased while it is carrying a rock. The rover is aware of its energy level and diverts to the closest charger when it gets below a minimum level by pushing a new goal "GoTo charger" onto the goalstack. The rover heads to the closest charger, which depends on its current location. After recharging, the "GoTo charger" goal is popped from the stack, and the rover resumes the prior goal. Figure 3 shows a contingent goal G3, to go to the charger location, that was pushed onto the stack.

Exogenous events: The agent must contend with exogenous events due to another agent also picking up and delivering rocks. The two agents do not communicate. An exogenous event occurs when another agent moves a rock while our agent is going to pick it up. In that case our agent must discard the goal for delivering the missing rock. In Fig 3, the goal G2 shows a precondition that rock R2 is at the pickup location. If R2 is missing when the rover gets to the pickup location, then the precondition fails, causing G2 and its parent goal G1 to be removed from the stack. An exogenous event also occurs when the agent arrives at the delivery depot and finds out that it is full/closed which causes the agent to find a secondary delivery location on the far (left) side of the grid. A third event is that a rock may fall out of cargo while the agent is carrying it, causing the agent to abandon delivery of that rock.

Application-level procedures. Figure 4 shows the key application level procedures for our example, simplified slightly for presentation. This is our unified action representation, used for both planning and execution. The library calls to Propel's meta-reasoning Supervisor processes are shown in bold. The top-level procedure is *Rover*(), which repeatedly chooses a visible rock (line 5), then calls PickupAndDeliver() (line 7), which pushes the top-level goal PickupAndDeliver onto the goalstack (line 9). PickupAndDeliver() then calls GoToCell() twice, to go to the pickup location and then to go to the delivery location.

GoToCell() repeatedly chooses a move direction (line 18) then moves in that direction, until it reaches the target cell. This is where most of the choices occur, choosing between North, South, East and West at every step.

GoToCell() first pushes a goal onto the goalstack to go to the cell *loc* (line 16). This is a subgoal for the initial goal PickupAndDeliver. The local heuristic s*ortBy-GoalDistance()* is passed to **`chooseValue()`** (line 18) to sort choices in order of increasing plan cost plus goal distance based on the node state.

```
1.  Rover()
2.     visibleRocks = list(closestRock())
3.    while not timeExpired:
4.       if visibleRocks:
5.          rock = chooseValue(visibleRocks,
6.                    sortByDistance, nodeState())
7.          PickupAndDeliver(rock)

8.  PickupAndDeliver(rock):
9.     goal = pushGoal(pickupAndDeliver(rock,
                  precond: isRockAt(rock.pos))
10.    GoToCell(rock.pickupLoc)
11.    pickup(rock)
12.    GoToCell(rock.deliveryLoc)
13.    putdown(rock)
14.    removeGoal(goal)

15. GoToCell(loc):
16.   g = pushGoal(goto(loc), preconds: None)
17.   while not at loc and currentGoal == g:
18.      dir = chooseValue({N,S,E,W},
                  sortByGoalDist, nodeState(), g)
19.      Move(dir)
20.      updateHeuristicScore()
21.      updateVisibleRocks()
22.      testPrecondsAndUpdateGoalstack()
23.      if subTourDetected():
24.         fail(reason: "subtour")
25.      elif lowEnergy():
26.         GoToCell(chargerLoc)   // recursive call
      // end while
27.    removeGoal(g, "success")

28. Move(dir):
29.    if isPlanningMode():
30.      simMove(dir)  // update node state
31.    else:
32.      doMove(dir)  // send cmd & update node state
```

Figure 4: **Application-level source code** for the Rover process with embedded library calls to search engine and goalstack.

After choosing a direction, the move is executed at line 19, which updates the node state with the new agent position. The node state's set of visible rocks is updated on line 21, and preconditions are tested against the updated node state on line 22. The precondition is specified as part of the goal statement on line 9. It's a Boolean Python method called isRockAt() which takes as input the expected location of the rock the agent is going to pick up. If updateVisibleRocks() removes that rock from the node state (because the other agent moved it), then isRockAt() returns False and **`testPrecondsAndUpdateGoalstack`**() will remove the related goal(s) from the goalstack. This removes the top goal on the stack, causing GoToCell() to exit the while loop (line 17).

If a subtour is detected on line 23 (the rover revisits the same cell location twice while achieving the same goal) then **`fail`** is called on line 24, which informs the Supervisor level. This is the trigger for calling the reactive planner.

GoToCell() is recursive and calls itself on line 26 to go to the charger location in the contingent case when energy is low. After the agent arrives at the goal location, the associated goal is removed from the goalstack (line 27).

Line 29 shows where the primitive action Move() branches to either execute or simulate the physical move depending on if the node is in planning or execution mode. In either case, the nodeState is updated to reflect the outcome of the move by simMove() or doMove().

The primary interface between the Application level and the Supervisor level is the statement: **`chooseValue(choices,choiceSorter, nodeState, goal)`** (lines 5 and 18). This is a nondeterministic assignment statement because it may produce different results in different contexts. The choices are sorted by the domain-specific heuristic function *choiceSorter*, which is defined at the application level. This choiceSorter takes the nodeState as input. The *goal* parameter identifies which goal is associated with this choice. Application level code may contain **`success`** and **`fail`** statements to inform the Supervisor level when a domain-specific success or fail state has been detected. The Supervisor may suspend a failed application process and inform the Chief Executive. In execution mode, a call to fail triggers a transition to planning. In planning mode, the node is pruned from the open node list so will not be selected for expansion.

Strategies for integrated planning and execution. PropelPy enables development of a wide range of different strategies for integrating planning and execution. We present two methods below which are used in our experiments. *Reactive planning* where the planner is called to solve the current goal when the controller gets stuck in a loop trying to move to a location. *Proactive planning* where the planner is called before execution starts.

Reactive Planning: Shadow/Solve planner modes. The reactive planner needs to stay in sync with execution choices, computational state, and external state. If execution moves several rocks before planning starts, then when app-level procs start in planning mode, they will see different rock positions compared to when those same app-level procs ran in execution mode. This motivated a new planning/execution transition strategy: *shadow/solve planning modes*. The agent starts by executing the application code, using default heuristics at each choice point. The planner starts off in "*shadow*" mode which means it will track and mimic the execution choices so the planner stays in sync with the controller. When the controller fails, the planner switches to "solve" mode. The planner is called to "solve" the top goal on the stack, starting from the same current state as execution. While the planner is in solve mode, the controller pauses the app-level processes which are in execution mode. When the planner achieves the goal, it switches back to shadow, and sends choice point rules back to the controller which resumes execution using the planner's advice. At this point, the planner is ahead of the controller, so the planner pauses while the controller executes the plan and catches up to the planner. After the controller follows all of the planner's advice it continues execution without planner advice. The planner switches back to shadow mode, mimicking execution choices until it is called again.

Proactive Planning: This is "anytime-ish", in that you may specify a planner time limit or a number of goals for the planner to solve before handing off the "best" plan to the controller. If a time limit is specified, the planner will return when the next goal is achieved (popped from the goalstack) after the time limit expires. Since the planner in our example uses A*, the most recently expanded node may not be on a solution path, so we defer the handoff until the next goal is solved.

Reactive execution: We are fundamentally interested exploring the division of labor between a controller with default reactive competence and a planner which can reason about all of the controller's behavior. Therefore, we created a "Controller only" mode which is pure execution and serves as a baseline for experiments to assess the value added by planning. Spending time to improve the controller's robustness often reduces the cases when the planner helps, but that is the trade off we are interested in studying. When is it worth spending time (and how much time should be invested) to improve the controller's reactive heuristics vs. tossing the problem the planner?

Search Space

Propel searches a space of program variations defined by choice points embedded in the code. When a choice point is executed that process calls Python's *fork*() to create a child process that continues with the selected choice. The parent process remains suspended until backtracking occurs, in which case the supervisor may wake up the node's parent process to generate a new choice (fork a new child).

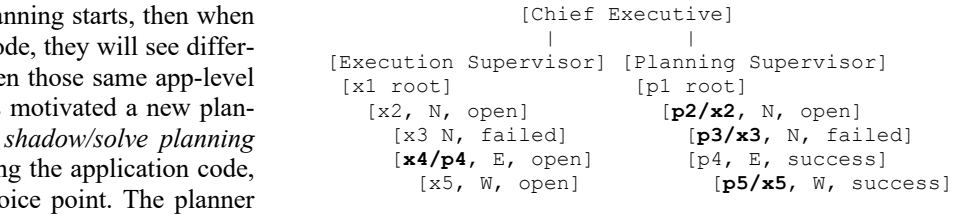

Figure 5: Search trees showing shadow/solve planning modes

Figure 5 illustrates the execution and planning process (node) trees and how they keep in sync. It shows an example of what we mean by "*flexible transitions*". There are several transitions (handoffs) between the Controller and the Planner here. They are taking turns following the leader (the one making choice point decisions). The leader sends its choices to the follower. The follower's node name is shown in bold and has the name of the leader's choice node appended. For example, "**p2/x2"**means that planner node p2 followed the choice from execution node x2.

Execution starts as the leader and the planner starts in *shadow* mode mimicking the execution choices. Execution starts by calling the top level Rover() method at the root node x1. When x1 reaches a choice point, it spawns x2 and informs the planner so it can follow along. Execution makes default heuristic choices for nodes x2 and x3, and the planner follows. When execution node x3's choice N fails, there is a handoff (transition) to the planner. The Chief Executive is notified of x3's execution failure then pauses the controller and tells the planner to *solve* the top goal (planner switches from shadow to solve mode). The planner takes the lead for p4. It backtracks and chooses direction E instead of N which solves the top goal on the stack. The planner informs the Chief Executive about the solved the goal, switches back to shadow mode, and becomes the follower. The Chief Executive restarts the controller which resumes execution using the planner's advice (x4/p4). Execution is now the leader and chooses W at x5, while the planner follows along.

Search Control

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

Heuristics are the second most important search control method. The app-level may provide *local* and *global* heuristics which are called by the Planner to control search. The local heuristics control the order of search node creation by sorting the choices at a choice point, while global heuristics control the order of node selection/expansion. First a node is selected for expansion by the global heuristic, and then the local heuristic is used to sort the order in which that node's children are created. The local heuristic sortByGoalDistance() sorts the choices in order of increasing plan cost plus distance to goal (based on the node State). *Cost* is the # of moves in the plan up to that point, and *distance* is the remaining Manhattan distance to the goal location. This is similar to A* but used only at the local level to sort the choices at a single choice point.

An actual A* search is implemented at the global heuristic level for node selection and pruning of nodes which reach the same state with worse A* score. To implement A* while maintaining our constraint that all domain knowledge is in the app-level, the app level must provide two methods: nodeEqual() and nodeScore(), which each take the nodeState dictionary as inputs. The estimated goal distance is tied to the destination cell for the current goal. When the rover changes destinations, say from a pickup location to a delivery location, then the goal distance calculation also changes. This also works when the contingent recharge action pushes a new goal to go to the charger location and then resumes heading to the prior location.

The Controller is locked into depth-first node selection and cannot backtrack so does not use the global heuristic.

Situated Control Rules (Drummond 1989; Drummond et. al, 1993; Levinson 1995) are the third most important search control method. Planning and execution nodes communicate by exchanging condition-action rules called Situated Control Rules (SCRs) which describe context and outcome for choices made during planning and execution. One SCR is defined for each branch in the search tree. The rule's condition is the computational state (control stack) and the nodeState for the parent process at the choice point. The rule's action is the choice which spawned the branch and the outcome. Computational state is captured using Python's introspection method **`inspect.stack()`.**

Situated Control Rule:
IF <*condition*> THEN <*choice*> <*outcome*>
<*condition*> = (<*StackFrame*>+)
<*StackFrame*> = (*method:file:lineNumber, nodeState*>)

```
Rule P46
IF ((gotoCell:rover.py:176,
       {currentPos: (16, 5), goalPos: (13, 5),
        t: 40, dist: 3, cost: 4, energy: 87}),
    (pickupRock:rover.py:113, {}),
    (pickupAndDeliverRock:rover.py:95, {t: 38}),
    (rover:rover.py:72, {t: 38}))
THEN (choice: 'W', Status: Success)
```

Figure 6: Situated Control Rule (SCR).
There is an SCR for each node in the search space.

Figure 6 shows the grammar and an example SCR. Rule P46 is the rule for planning node p46 in the search tree. Rule P46's condition says (starting from the bottom of the control stack/last stackframe): The top-level method rover() called subroutine pickupAndDeliverRock at line 72 of file rover.py, with local variable t (time) = 38. Then pickupAndDeliverRock called pickupRock at line 95 of rover.py, with t = 38. Then pickupRock called gotoCell at line 113 of rover.py. The top stack frame shows gotoCell executed the choice point leading to this rule at line 176 of rover.py, with node state (local variables) showing the agent's current position (16,5), goal position (13,5), t = 40, with distance to goal = 3, cost (# of prior steps) = 4, and energy level 87. This rule, created by the planner, will be applied by the controller for execution nodes with a matching state. The distance and cost values shown in the top stack frame are passed to the local and global heuristics.

SCRs are collected when transitioning between planning and execution and vice versa. Collecting SCRs is similar to classical goal regression. When the planner solves a goal, then SCRs are collected for the path from the given leaf node to the root node and passed to the controller so it knows the planner's choices. When an application level

process executes a choice point, SCRs are combined with the local heuristics to sort the choices. Success choices are chosen first and failed choices are chosen last.

Managing process combinatorics: `chooseValue`() uses Python's `fork`() method to split a single computational process into multiple processes, each representing a continuation with a different choice. We use several methods to minimize the number of computational processes created and running at any given time. Most importantly, processes are aggressively killed as soon as possible. Since there is no backtracking during execution, only one process is maintained for application level code running in execution. After a choice is made in execution mode, the parent process is killed and only the child continues on. When the planner is in shadow mode, it does the same thing because it also will not backtrack over execution commitments.

Experiment

Our general hypothesis is that planning can extend the range of operating conditions for execution beyond situations covered by default reactions. Our expectation is that with the help of the planner, execution gets more done with fewer steps as problem difficulty increases. Those results were demonstrated with the original version of Propel (Levinson 1995a). We aim to reproduce similar results with PropelPy.

This preliminary evaluation involved running three agent variants. One variant, C, uses only the Controller without the Planner. This means the agent relies solely on the controller's heuristic reactions. The variant S uses the reactive planner with the shadow/solve strategy. Variant A is our proactive, anytime-ish planner.

We built three versions of C, called C1, C2 and C3. C1 is the simplest. It just turns another direction when it runs into an obstacle, but easily gets caught moving in circles in small cul-de-sacs. C2 is a bit smarter, but still gets caught looping in (larger) cul-de-sacs. C3 is the most advanced and gives the planner a good challenge.

C1 and C2 both get stuck in many cases and are easily beat by S and A. Therefore, *our experiments all use C3*, which is a challenge to the planner. Sometimes the planner improves on C3, and sometimes C3 fails, but also sometimes the planner only degrades performance. These different controller behaviors are implemented as variations of the local heuristic which sorts the move direction choices.

Table 1 summarizes our results. The columns are: *P#* is the problem id, *o%* is obstacle density, *r%* is rock density. The obstacle density is the chance that an obstacle is in any given cell (10% chance for first problem), and the rock density is the chance that a rock is in any given cell. Column *x* is the # of execution nodes (corresponds to the # of executed moves); *p* is the # of planning nodes, *r* is the # of rocks delivered, *g* is the number of goals created, *s* is the number of times the planner was called to solve a problem.

P #	C/ S/A	o %	r %	x	p	r	g	s	t	xt
1	C	10	5	181	-	5	25	-	211	211
1	S	10	5	177	198	5	25	2	215	215
2	C	15	5	181	-	7	37	-	214	214
2	S	15	5	181	201	7	37	1	222	222
3	C	18	10	177	-	7	34	-	210	210
3	S	18	10	179	182	7	34	1	222	222
4	C	20	10	186	-	7	33	-	220	220
4	S	20	10	174	183	6	29	-	212	212
5	C*	20	25	-	-	-	-	-	-	-
5	S	20	25	189	243	7	36	2	250	249
6	C	22	25	170	-	7	35	-	205	205
6	A15	22	25	192	245	8	40	1	278	236
6	A30	22	25	192	419	8	40	1	313	240
7	C	22	25	183	-	7	33	-	218	218
7	S	22	25	179	547	7	33	4	322	322
8	C*	27	10	-	-	-	-	-	-	-
8	S	27	10	182	458	6	30	1	290	290
8	A5*	27	10	-	-	-	-	1	-	-
8	A10*	27	10	-	-	-	-	1	-	-
8	A15	27	10	172	215	6	30	1	243	208
8	A20	27	10	172	296	6	30	1	259	209

Table 1: Preliminary Experiment Results

Column *t* is the total time (seconds) for the run, which equals planning time plus execution time. Column *xt* is the execution time component of *t*. *Planning time = t - xt*.

We ran C, S and A on several different problem instances with different combinations of obstacle and rock densities. Each experiment ran for 150 execution steps before heading back to home base.

For these experiments, our primary metric is comparing the # of execution nodes created (column *x*) for C vs S and A. We expect to trade planning nodes *p* for execution nodes *x* as the problem gets more difficult. This means the planner explores more choice points (creates more nodes) than the controller, so that the controller executes fewer steps (creates fewer nodes) when advised by the planner. We are also interested in how total time *t*, and execution time *xt,* vary as a function of planning time.

For a given problem, we expect a smaller *x* in the S and A cases compared to C, and would like to see more rocks delivered (*r*) with planning than with C. We also expect those differences to increase with problem difficulty.

The first 3 problems are so simple (unconstrained) that there is marginal benefit from planning while increasing total time *t* and (surprisingly) also *xt*. Problem 4 is the first case where the planner shows some benefit by reducing the number of execution steps and the total time, but surprisingly also delivers one less rock and solves fewer goals.

On problem 5, the C case is marked with a * to indicate that it failed to reach completion. It gets stuck walking in circles until it runs out of energy, so planning clearly helps.

Problem 6 compares C with 2 variants of the anytime-ish planner. A15 means the planner was proactively called

to solve 15 goals before starting execution, and A30 means it solved 30 goals before starting execution. Both A15 and A30 delivered one more rock than C, while increasing the number of execution steps *x* and the taking more time *t*. Problem 7 shows marginal reduction in execution steps while taking significantly more time.

Problem 8 is shown in Figure 2, and is the most interesting. C gets stuck in the large jagged cul-de-sac in the center top of the grid. S is reactively called and successfully plans an escape from that trap. A5 and A10 also get stuck, indicating that pre-planning 5 goals (A5) or 10 goals (A10) is not enough planning to avoid the trap. A15 shows the most benefit for the least cost, where pre-planning 15 goals solves the problem while reducing the # of execution steps, total time and execution time (compared to S). A20 shows that pre-planning an additional 5 goals provides no benefit while taking slightly more time than A15.

This experiment was designed to illustrate our hypothesis and how we might evaluate the system. The results are very preliminary and specific numbers depend on the quality of the application-level heuristics and how much complexity is built into C. The general result patterns do appear to support our hypothesis that planning can increase the range of operating conditions for execution.

Related Work

ERE (Drummond, et. al, 1993). Propel is a direct extension of the Entropy Reduction Engine (ERE) and incorporates several ERE features for integrated planning and execution including SCRs and Reaction-First Search (RFS), where the planner first explores the controller's heuristic choices. A key difference is that Propel uses a procedural action representation compared to ERE's STRIPS-like action representation.

Propel 1 (Levinson, 1995) laid the foundation for this work and **Propel 2** (Levinson 2005) built on that using C++ as the action representation. It supported concurrent application-level procedures (e.g., wheels and camera) and used a Simple Temporal Network (STN) (Dechter et al., 1991) to coordinate the concurrent wheels and camera processes through a shared database.

IDEA (Muscettola et. al, 2000) is a unified planning and execution system like Propel. However, IDEA executes the *planner's* language while Propel plans with the *controller's* language. IDEA's controller executes plans by interpreting the planner's *declarative* language. IDEA has no default reactive competence and calls the planner to refine the plan before each execution step.

Operational Models and **RAEplan** (Patra et al. 2019) has similar motivations to unify the planning and execution with a shared procedural action representation, and to explore how planning effort affects execution performance. RAEplan's action representation is that of RAE (Ghallab et al., 2016), which, although procedural, is less expressive than Python. Choices in RAEplan are restricted to choosing tasks, while choices in PropelPy may be any Python data type or class object.

Semi-Black Box (SBB) methods (Katz et al. 2018) describe methods for integrating planning into Java and is motivated by similar concerns about planning methods being inaccessible to non-AI experts. A key difference is that SBB is planning only, while our approach is designed for integrated planning and execution.

Future Work

Integration/transition Strategies. A primary motivation of this work is to explore different *transition/handoff strategies* between planning and execution. This involves defining the transition triggering events and planner termination criteria. We are continuing development of the strategies presented above, as well as exploring others. For example, we could combine reactive and proactive, so that the planner is called after execution failure, but plans for some time or some # of goals beyond the top goal which failed.

Intra-modal rules: Currently, SCRs generated by the planner are only used by the controller and vice versa (inter-modal). We may explore reusing planner rules within planning, for example transferring rules learned about successful or failed choices from one part of the planning tree to another.

Goal Reasoning integration. This paper presents only an initial effort for integrating goal reasoning into Propel. We plan to explore many issues not yet addressed, such as goal reformulation and opportunistic goals. We plan to tighten the connections between goals, procedures and choice points.

Conclusion

This paper addresses motivations and challenges for tightly integrated planning and execution in autonomous systems. Two key challenges addressed are unifying the action representations used by the planning and execution systems, and using a general programming language for that action representation. We aim to expand the scope of the planner's model to cover the complex behavior of fully-expressive general programming languages. We'd like to do this without sacrificing the benefits of unified planning and execution models to minimize the need to develop and maintain different models, and minimize the risk of information lost in translation between different models.

Contributions of this work include: (1) A library for integrating planning, execution and goal reasoning into Python, accessible for programmers with minimal AI training, (2) a procedural action representation for unified planning and execution to minimize the language barrier and facilitate flexible transitions, (3) a tool to study autonomy as a dynamic balance between planning and execution.

References

Aaseng, G.; Frank, J.; Iatauro, M.; Knight, C.; Levinson, R.; Ossenfort, J.; Scott, M.; Sweet, A.; Csank, J.; Soeder, J.; Carrejo, D.; Loveless, A.; Ngo, T.; and Greenwood, Z. 2018. Development and Testing of a Vehicle Management System for Autonomous Spacecraft Habitat Operations, *Proc. of AIAA 2018*, Orlando FL.

Aha, D. W. 2018. Goal Reasoning: Foundations, Emerging Applications, and Prospects. *AI Magazine* 39 (2).

Barkley, R. 2012. *Executive Functions: What they are, How they work and whey they evolved*. Guilford Press. London.

Cashmore, Michael & Fox, Maria & Long, Derek & Magazzeni, Daniele & Ridder, Bram & Carrera, Arnau & Palomeras, N. & Hurtós, N. & Carreras, Marc. 2015. Rosplan: Planning in the robot operating system. *Proceedings International Conference on Automated Planning and Scheduling, ICAPS. 2015*.

Chu,Y., Song, YC., Levinson, R., Kautz, H. 2012. "Interactive Activity Recognition and Prompting to Assist People with Cognitive Disabilities". *Journal of Ambient Intelligence and Smart Environments*. 2012.

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

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

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

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

Ingrand, F, Chatilla, R. Alami, R, Rober, F. 1996. PRS: a high level supervision and control language for autonomous mobile robots. *In IEEE Int'l Conf. on Robotics and Autonomation.*

Katz M., Moshkovich D., Karpas E., 2018. Semi-Black Box: Rapid Development of Planning Based Solutions. *AAAI 2018*., AAAI Press, Menlo Park, CA.

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

Levinson, R. 1994. Human Frontal Lobes and AI Planning Systems. *Proc. of AI Planning Systems (AIPS-94*). AAAI Press.

Levinson, R. 1995a. A General Programming Language for Unified Planning and Control. *Artificial Intelligence, Vol. 76. Issue on Planning and Scheduling.* https://brainaid.com/pubs/aij.pdf

Levinson, R. 1995b. A Computer Model of Prefrontal Cortex Function. *Annals of the New York Academy of Sciences: The Structure and Function of Prefrontal Cortex*. Vol. 769. https://brainaid.com/pubs/nyas.pdf

Levinson, R. 1995c. An interdisciplinary theory of autonomous action. *AAAI Stanford Spring Symposium*. Stanford, CA.

Levinson, R., 1997. The Planning and Execution Assistant and Trainer. Journal of Head Trauma Rehabilitation, April 1997, Aspen Press. https://brainaid.com/pubs/jhtr.pdf

Levinson R. 2005. Unified Planning and Execution for Autonomous Software Repair *(unpublished manuscript originally submitted to ICAPS 2005) http://brainaid.com/pubs/ICAPS_05.pdf*

Levinson, R. 2007. An autonomous cognitive aid with integrated Sensing, Planning and Execution. *Workshop on Intelligent Systems for Assisted Cognition,* University of Rochester, NY.

Levinson, R., Halper, D., Harman, C., Kautz, H. 2009. "A Conversational Cognitive Aid with Activity Monitoring, Planning, and Execution." (Best Paper Award). *IJCAI Workshop on Intelligent Systems for Assisted Cognition*, Pasadena, CA. June 2009.

Levinson, R., 2019. Constraint Integer Program Formulations for NASA Planning, Scheduling and Autonomy Problems. *Proc. of ICAPS 12th International Scheduling and Planning Applications Workshop.* Berkeley, CA.

Lezak, M. Howelson, D., Bigler, E., Tranel, D. 2012. *Neuropsychological Assessment. Fith Edition*. Oxford University Press.

Modayil, J., Levinson, R., Harman, C., Halper, D., Kautz, H. "Integrating Sensing, Planning and Cueing for More Effective Activity Reminders". *Proceedings of the American Assoc. for Artificial Intelligence (AAAI) Fall Symposium on AI in Eldercare: New Solutions to Old Problems.* Washington DC. 2008.

Muscettola, N., Dorais G.., Fry,C., Levinson, R., Plaunt, C. 2000. A Unified Approach to Model-Based Planning and Execution. *The 6th Int'l Conf. on Intelligent Autonomous Systems*. Venice.

Muscettola, N., G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt, 2002. "IDEA: Planning at the core of autonomous reactive agents," in *Proc. of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002

Patra, S., Gallab, M., Nau, D., Traverso, P. 2019. Acting and planning using operational models. *In AAAI*. AAAI Press.

Verma, V., Jonsson, A., Simmons, R., Estlin, T., Levinson, R. 2005. Survey of Command Execution Systems for NASA Spacecraft and Robots. *ICAPS-05 Workshop on Plan Execution*, Monterey, CA.

Verma, V, Jonsson, A, Pasareanu, C, Iatauro, M. 2006. Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations. *In AIAA Space.*