Nyx: Planning for Emerging Problems with PDDL+ and Beyond

Wiktor Piotrowski[1], **Alexandre Perez, Sachin Grover**[1]

[1] PARC, part of SRI International CA, USA
e-mail: wiktor.piotrowski@sri.com, alexandrecperez@gmail.com, sachin.grover@sri.com

Abstract

Real-world applications of AI Planning often require a highly expressive modeling language to accurately capture the intricacies of target systems. Hybrid systems are ubiquitous in the real-world, and PDDL+ is the standardized modeling language for capturing such systems as planning domains. PDDL+ enables accurate encoding of mixed discrete-continuous system dynamics, exogenous activity, and other features encountered in the real world. However, the uptake in usage of PDDL+ has been slow and apprehensive, largely due to a shortage of general-purpose PDDL+ planning software, and rigid limitations of the few existing planners. To overcome this chasm, we present Nyx, a novel PDDL+ planner built to emphasize lightness and adaptability to a wide range of novel real-world problems that often require models beyond the expressiveness of PDDL+. Nyx can be tailored to tackle a wide range novel problems rooted in real-world applications, facilitating wider adoption of versatile planning methods. We evaluate Nyx on openAI Gym's classic control problems: Mountain Car and CartPole, modeled in PDDL+.

Introduction

Realistic planning problems require an expressive modeling language to accurately capture the innate intricacies of the modeled scenario. Indeed, most existing domain-independent planning languages, such a STRIPS (Fikes and Nilsson 1971), ADL (Pednault 1989), PDDL (McDermott et al. 1998), and even PDDL2.1 (Fox and Long 2003), lack features to describe commonplace elements of real-world systems. As a result, the vast majority of planning models are severely abstracted or limited in scope, often being forced to ignore aspects of the domain that, in the real world, have significant impact on the system's operations such as environmental phenomena.

PDDL+ (Fox and Long 2006) is one of the most expressive modeling languages and was designed to model mixed discrete/continuous (hybrid) systems. PDDL+ is, arguably, the closest AI Planning has come to accurately representing realistic scenarios as planning domains. It has proved useful for capturing a wide range challenging AI domains from traffic control(Vallati et al. 2016) and physics-based games(Piotrowski et al. 2021) to spacecraft operations(Piotrowski 2018). However, the resulting PDDL+ planning problems are notoriously difficult to solve. Planning with PDDL+ domains is undecidable and requires reasoning with a wide range of advanced domain features, including non-linear system dynamics, exogenous happenings, high branching factors, temporal activity, and more.

Solving PDDL+ problems requires a powerful and efficient planner, capable of reasoning with the aforementioned aspects of PDDL+. However, development of PDDL+ planners has been slow and underwhelming. There are only a handful of available domain-independent PDDL+ planners, and each one is only suited for specific class of problems. Furthermore, most of them have a rigid and complicated code-base, very specific (often outdated) dependencies, and are generally too cumbersome to adapt for novel classes of planning models. Consequently, in lieu of AI Planning, the scientific community and potential target users turn to more accessible and flexible approaches which can be easily adapted or trained to solve emerging real-world challenges. Often, AI Planning can be particularly well-suited to efficiently tackle certain classes of problems and provide sought-after characteristics such as robustness, explainability, and transparency. However, it is overlooked as a viable approach simply because decision-makers are unaware of its potential and advantages. This issue is particularly evident for applications requiring more a expressive modeling language (recent example, well-suited for PDDL+ planning, but overlooked in favor of Learning methods: (Vouros et al. 2022)). There is a dire need for more flexible AI Planning tools which can efficiently tackle a wide range of real-world problems and which can be easily adapted to emerging classes of applications.

We present Nyx, a novel lightweight PDDL+ planner designed with adaptability and simplicity at the forefront of development. Nyx is built to increase planning performance via easily implemented heuristics and exploiting high-fidelity settings configurations. In this paper, we describe the motivation behind developing Nyx, discuss the basic principles it is built on, and outline the path for adapting the planner to any potential scenario.

However, having an high-performing and capable planner is often not enough to solve many realistic problems. Real-world systems are often prohibitively complex and cannot be modeled as planning domains. Even the expressiveness of PDDL+ is bound to basic arithmetic operations, and any-

thing beyond is either impossible to include in the domain (e.g., square root, logarithms) or requires numeric approximations (e.g., trigonometric functions). Thus, the second contribution of our work is to overcome the expressiveness limits of PDDL+. Nyx supports advanced domain extensions that span well beyond the current expressiveness of PDDL+.

Related Work

To date, the language with, arguably, the most set of features relevant to real-world problems is PDDL+ (Fox and Long 2006). It was designed specifically as a planning standard for modeling hybrid systems (switched dynamical systems) governed by a set of differential equations and discrete mode switches. Formally defined as a mapping of planning constructs to Hybrid Automata (Henzinger 2000), PDDL+ enables the modeling and solving of problems set in systems exhibiting both discrete mode switches and continuous flows. PDDL+ builds on the expressiveness of its predecessors, it encapsulates the entire set of features from PDDL2.1 (including continuous effects) and supplements that with the inclusion of timed-initial literals (TILs) from PDDL2.2. However, the biggest advancement over its predecessors is the inclusion of new constructs for defining exogenous activity, namely discrete *events* and continuous *processes*.

The expressiveness of PDDL+ enables natural definitions of certain phenomena ubiquitous in real-world scenarios. However, the resulting planning problems are notoriously difficult to solve due to immense search spaces and complex system dynamics. Indeed, planning in hybrid domains is challenging because, apart from the state space explosion caused by discrete state variables, the continuous variables cause the reachability problem to become undecidable (Alur et al. 1995). As a result, efficient model implementation is crucial to solving PDDL+ problems.

Recent works highlight how PDDL+ expressiveness, teamed with clever system modeling, can achieve high planning performance despite the problems' complexity in a range of domains, including Urban Traffic Control (Vallati et al. 2016), Angry Birds (Piotrowski et al. 2021), Chemical Batch Plant (Della Penna et al. 2010), drilling (Fox et al. 2018), Minecraft (Roberts et al. 2017), and UAV control (Kiam et al. 2020).

Currently available PDDL+ planners vary in their performance and abilities to reason with different PDDL+ features. Real-world applications often require high-performing planners to handle large numbers of happenings, wide range of model features, and non-linear system dynamics. SMT-Plan+ (Cashmore et al. 2016) can reason with all features of PDDL+ but it is only capable of dealing with polynomial non-linearity and does not scale well with the number of happenings. UPMurphi (Della Penna et al. 2009) is highly optimized and can reason with the entire set of PDDL+ features and non-linear dynamics, but its scalability is impacted by lack of heuristics. DiNo (Piotrowski et al. 2016) is a PDDL+ planner which builds on the strengths of UPMurphi and extends it by implementing a domain-independent heuristic. However, DiNo's heuristic is computationally expensive and does not perform well some classes of domains. ENHSP (Scala et al. 2016) is a versatile PDDL+ planner equipped with various model-independent heuristics and numerous search configurations. It can handle complex PDDL+ domains and also reason with advanced mathematical expressions. ENHSP has very minor limitations with respect to expressiveness. All of the aforementioned planners have rigid, complicated code-bases that heavily rely on numerous libraries. As a result, they cannot be easily adapted to novel systems and scenarios, limiting their usefulness in tackling emerging classes of problems.

Nyx Planner Implementation

Lightness and accessibility are ubiquitous throughout the entire architecture and implementation of Nyx. The planner is written in Python, a high-level scripting language which facilitates rapid code writing and comprehension. Python is the world's most popular programming language(TIOBE 2022; IEEE Spectrum 2022), and is particularly favored for AI and scientific computing applications. Python was chosen as Nyx's implementation language to make the planner more accessible and encourage various extensions and modifications. In contrast, most established planners are very specialized tools aimed at very narrow target audience. Their code-base is usually written using performance-focused languages (i.e., C/C++) and optimized to extract every bit of performance available. While the performance gains are significant, the software becomes much less flexible and accessible. Modifications to the tool become infeasible due to convoluted and unreadable code-base that is usually also highly-dependent on external libraries.

Nyx, on the other hand, aims at different targets. It serves as a transparent, easy-to-understand platform for rapid prototyping and testing of algorithms and approaches. Nyx itself has a very minimalistic structure, only containing 4 base classes (main, parser, planner, and simulator), 8 object classes (e.g., action, state, trace), and 3 compiler classes (for efficient parsing and evaluating of PDDL+). Finally, our planner has two auxiliary template classes for implementing novel heuristic methods and external functions/semantic attachments. Nyx has no complicated or obscure dependencies, it relies only on the very basic concepts in Python.

Most PDDL+ planners are tailor-built for a specific class of problems and thus only support a subset of PDDL+ features and algorithms. Indeed, there are few planners which are capable of adjusting their planning algorithms and parameters beyond a few simple options. Furthermore, established PDDL+ planners are rigid in their structure and cannot be easily modified to accommodate new classes of planning problems or features. Thus, problems intersecting multiple classes or requiring support for the full set of PDDL+ features have a very limited choice in planning solvers with such coverage. Each class of problems might require a separate planner suited to that particular type of scenarios, defeating the notion of general-purpose PDDL+ planning.

To counter these shortcomings and ensure that Nyx can deal with a variety of problem types, we equipped Nyx with a range of generic strategies and operating options as a foundation for the planner. However, the built-in functionality of Nyx serves as the basis for further extensions. Nyx is specifically designed to adapt to novel classes of problems by al-

lowing semantic and syntactic extensions, and rapid implementations of novel heuristics, external functions, and other features. Nyx facilitates extensibility and straightforward integration of novel features, crucial aspects for attempting to tackle emerging real-world problems and domains whose scope spans beyond PDDL+.

Planning Paradigms

Simplicity is the main focus in Nyx. The planner relies on fundamental approaches for representing time, states, change, and happenings.

Planning via Discretization Nyx adheres to the *planning-via-discretization* paradigm for handling the temporal and continuous behavior of hybrid dynamical systems. Nyx approximates a model's continuous system dynamics using a uniform time step (Δt) and step-functions. Nyx also specifies number precision for representing numeric state variables (which can be specified by the user). Nyx follows standard discretization-based planning approach of introducing a special *time-passing* action (a_{tp}) that, during search, the planner can choose to apply alongside actions defined in the domain. *Time-passing* allows the planner to advance time during search and apply the effects of events and processes, over a duration of the time step Δt. *Planning-via-discretization*, while a straightforward approach, enables Nyx to reason with all types of system dynamics, including non-linear behavior. Plans generated by Nyx can be validated (with VAL (Howey, Long, and Fox 2004)) to complete the *Discretize & Validate* cycle (Della Penna et al. 2009).

More formally, Nyx converts the planning model into a **Discretized Finite State Temporal System (DFSTS)** $\mathcal{S}{=}(S, \mathcal{A}, E, P, \mathcal{D}, F)$, where S is a finite set of states, $\mathcal{A}$ is a finite set of actions (including the special time-passing action $a_{tp}{\in}\mathcal{A}$), and $\mathcal{D}$ is a finite set of durations. $F{:}S{\times}\mathcal{A}{\times}\mathcal{D}{\rightarrow}S$ is the transition function. E and P are the finite sets of events and processes, respectively. The formalisms in this work are adapted from (Fox et al. 2012).

A **state** $s{\in}S$ is a tuple $s{=}(p(s), v(s), t(s))$, where $p(s)$ is a finite set of propositions, $v(s)$ is a finite set of continuous variables, and $t(s){\in}\mathbb{R}$ is the time at which state s occurs (i.e., elapsed planning time from the initial state, $t(s_0){=}0$). The transition function F yields a successor state $s'{\in}S$ by executing action a in state s, i.e., $F(s, a, d(a)){=}s'$ where $d(a){\in}\mathbb{R}_{\geq 0}$ denotes the duration of a and $t(s'){=}t(s) + d(a)$.

An **action** $a{\in}\mathcal{A}$ is a tuple $a{=}(pre(a), eff(a))$, where $pre(a)$ is a set of preconditions (both propositional and numeric), and $eff(a)$ is a set of effects over state variables $p(s)$ and $v(s)$. An action a is *applicable* in state s iff its preconditions $pre(a)$ are satisfied, i.e., $s{\models}pre(a)$.

Events and **processes** are analogous to actions, i.e., $e{=}(pre(e), eff(e)){\in}E$ and $p{=}(pre(p), eff(p)){\in}P$, respectively. Thus, their effects can technically be applied via the transition function F from DFSTS, i.e., $F(s, e, d(e)){\rightarrow}s'$ and $F(s, p, d(p)){\rightarrow}s'$. In practice, however, events and processes are encapsulated in and executed by the special *time-passing* action a_{tp}. The time-passing action has no preconditions itself $pre(a_{tp}) = \emptyset$. On the other hand, the effects of a_{tp} are the union set of effects of all events and processes whose preconditions are satisfied in state s: $eff(a_{tp}){=}\bigcup_{ep \in EP} eff(ep)$ where $s{\models}pre(ep)$ and $EP{=}E{\cup}P$.

A **DFSTS-based planning problem** is a tuple $\mathcal{P}{=}(\mathcal{S}, s_0{\in}S, G, \Delta t, T)$, where $\mathcal{S}$ is an DFSTS, $s_0{\in}S$ is the initial state, $G \subseteq S$ is a finite set of goal states, $\Delta t{\in}\mathbb{R}$ is the discrete time step, and T is a finite temporal horizon which bounds the state space. All states (including goal states) must heed the bound imposed by the finite temporal horizon T, i.e., $\forall s{\in}S : t(s) \leq T$.

A **trajectory** π in DFSTS $\mathcal{S}$ is a sequence of states, actions, and action durations, which ends with a state: $\pi = s_i, a_i, d(a_i), s_{i+1}, a_{i+1}, d(a_{i+1}), ..., s_n$, where $i{\in}\mathbb{Z}_{\geq 0}$, $n{\in}\mathbb{Z}_{\geq 1}$, $s_i{\in}S$, $a_i{\in}\mathcal{A}$, and $d(a_i){\in}\mathcal{D}$. A trajectory π^* is a solution to a planning problem $\mathcal{P}$ if it begins in the initial state s_0 and ends in a goals state $s_n{\in}G$. Furthermore, for each step i in the trajectory, the transition function F yields the following state, i.e., $F(s_i, a_i, d(a_i)) \rightarrow s_{i+1}$.

Nyx dictates a set of additional assumptions about planning via discretization. All actions are instantaneous (i.e., have duration $d(a) = 0$), except *time-passing* a_{tp} whose duration is equal to the discrete time-step Δt, i.e., $d(a_{tp}) = \Delta t$. Thus, in practice, the finite set of durations is $\mathcal{D} = \{0, \Delta t\}$. In the discretized state space, states only occur at time-points which are multiples of time step Δt, i.e., $\forall s{\in}S : t(s) = \Delta t * n$, where $n = \mathbb{Z}_{\geq 0}$ is a non-negative integer. Thus, the preconditions of events and processes are also checked at Δt intervals. Finally, the effects of continuous processes are applied by the transition function F, each time over the uniform interval Δt. Nyx neglects the support for durative actions which can be seen purely as syntactic convenience. Instead they can be equivalently compiled according to the start-process-stop model (Fox and Long 2006) where instantaneous actions and/or discrete events can trigger and terminate continuous change.

Finally, setting the time discretization $\Delta t = 0$ nullifies the continuous aspect of the model and removes the need for the time-passing action a_{tp}. In Nyx, this is done automatically, by simply excluding `:time` from the PDDL domain requirements. However, Nyx allows events to be triggered in non-temporal domains, since they only apply discrete change themselves.

Grounding The initial parsing of the PDDL+ domain and problem files explicitly grounds the initial state. All states in Nyx are currently fully grounded. Similarly, all happenings are fully grounded after the PDDL model is parsed.

Expression Compilation To bridge the performance gap between Nyx and other planners written in compiled languages, Nyx features expression compilation. For every effect and precondition PDDL expressions in a grounded domain, Nyx translates them to equivalent Python bytecode at runtime. This compilation step is transparent to users and is performed in a just-in-time (JIT) fashion as expressions are evaluated for the first time. Subsequent evaluations of a

compiled expression will use JITed Python code instead of interpreted PDDL code for increased efficiency.

Search Algorithms and Heuristics

Similarly to other discretization-based PDDL planners, Nyx converts the discretized PDDL+ problem into a graph search task, where planning states are represented by nodes and actions by edges. Nyx traverses the state space using forward-chaining search algorithms. Currently, the planner is equipped with four basic search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best-First Search (GBFS), A*[1].

By default, breadth-first search is enabled. The user can specify their preferred search algorithm to use by adding a command-line argument when running Nyx. The algorithms change the ordering and queuing procedures of the open list of visited states. Using A* and GBFS requires defining a heuristic function which evaluates each generated successor state. Nyx has a special class which stores the implementations of heuristic functions. Invoking a specified heuristic function is done with command-line parameters. Multiple heuristics can be implemented at the same time, the user only needs to specify the index of the heuristic function they intend to use for a given problem. The heuristic function takes in a planning state object as input and returns a numeric value, i.e., the heuristic estimate for the given state. The heuristic value is then used to enqueue the generated state into the priority open list of explored states. The open list's enqueue mechanism is determined by the active search algorithm, and the states are ordered w.r.t. heuristic estimate.

The planner's search process can be constrained by the user to limit the already vast state space and/or to focus the search effort to a particular subsection. The user can specify the temporal horizon (i.e., states beyond a certain time-point will not be expanded), depth limit (i.e., strict cap on plan length), and planner timeout (limited runtime).

Plan Metrics Temporal planning is undecidable, thus most planners reasoning with PDDL2.1 and beyond are usually only concerned with finding a feasible solution to a given problem. Due to this obstacle, plan quality has been pushed to the background when working with more advanced domains written in more expressive paradigms such as PDDL2.1 or PDDL+. Plan metrics measure the quality of a solution based on a pre-specified function in the problem file. However, plan metrics have been largely neglected in temporal planning and beyond.

Nyx allows for the parsing of a plan metric defined as a mathematical expression in the PDDL problem file. The metric definition follows the syntax of PDDL+ and requires a function that returns a numeric value. The planning objective is to find solutions with optimized metric (either minimize or maximize). There are two special cases for plan metrics which concern important aspects of the solution but cannot be defined with respect to state variable values. Instead, these metrics focus on finding plans with optimized

[1]Currently, the look-back value $g(s)$ of the A* algorithm is implemented to track the number of actions back to the initial state s_0. This will be updated in subsequent version of Nyx.

```
(:metric minimize
    (total-time))
(:metric maximize
    (* (fuel_remaining) (total_reward)))
(:metric minimize
    (total-actions))
(:metric minimize
    (* (- (revenue) (cost)) inflation))
```

Figure 1: Examples of PDDL plan metrics.

with respect to makespan ("total-time") and number of plan actions ("total-actions"). If no metric is specified, the planner will minimize plan makespan.

Anytime Search Nyx is equipped with anytime search algorithm (Hansen, Zilberstein, and Danilchenko 1997) for improving the quality of solutions as a trade-off for search time. After the first valid goal state has been encountered, the anytime algorithm exploits prior search progress and continues to explore the state space in search of improved solutions. The anytime approach uses plan metrics which quantify solution quality. Found plans are ranked with respect to plan metrics. After the allotted search time has been exhausted, the planner returns the ranked list of solutions.

For practicality reasons, the number of solutions concurrently held in memory is limited by the user who can specify the maximum size of the list containing the top plans. Similarly, the user specifies the planning timeout limit. When the limit is reached, Nyx returns the encountered plans.

Nyx does not have a separate anytime search algorithm as such. Instead, the user can select from the already implemented algorithms (BFS, DFS, GBFS, A*) or implement their own search algorithm. The selected algorithm continues to explore the state space instead of terminating at the first encountered goal state (default timeout is 30 minutes).

Nyx Planning Algorithm

The main planning approach of Nyx is summarized in Algorithm 1. The presented algorithm is a skeleton which can be modified by the user to include various operations, such as an additional event check (at the very beginning of the time passing block). The algorithm returns a set of goal states (*reachedGoals*) from which a trajectory can be extracted via backtracking using *predecessor(s)* and *achiever(s)* functions which return the predecessor state and achieving action of any state s.

Precondition Tree

Internally, Nyx reasons with grounded representations of states, actions, events, and processes. Unfortunately, this approach can cause an explosion in the number of happenings which need to be repeatedly checked for applicability at every step during search. Checking a list of preconditions of all actions, events, and processes in a linear fashion is often unfeasible. To improve the efficiency of the planner, this paper introduces **precondition tree**, a novel approach to checking happening applicability inspired by the concept of the trie (Fredkin 1960). Structuring the precondition checking

Algorithm 1: Nyx Planning Algorithm

Input : $\mathcal{A}_g$, E_g, P_g - list of grounded actions, events, and processes, respectively.
Input : requirements - list of domain requirements.
Input : s_0 - initial state.
Input : Δt - time discretization quantum.
Output: reachedGoals - list of reached goal states.

OPEN$\leftarrow \{s_0\}$
while *OPEN* **do**
  $s \leftarrow$ pop(OPEN)
  if $s \in G$ **then**
    enqueue(reachedGoals, s, metric(s))
    if *not anytime* **then**
      return reachedGoals
  $s' \leftarrow$ copy(s)
  predecessor(s') $\leftarrow s$
  foreach $a \in$ *applicable(actions, s)* **do**
    if a *is time-passing* **then**
      if *":semantic-attachment"* $\in$ *requirements* **then**
        $s' \leftarrow$ semanticAttachment(s')
      foreach $p \in$ *applicable(processes, s')* **do**
        $s' \leftarrow F(s,p,\Delta t)$
      foreach $e \in$ *applicable(events, s')* **do**
        $s' \leftarrow F(s,e,0)$
    else
      $s' \leftarrow F(s,a,0)$
      if *":time"* $\notin$ *requirements* **then**
        foreach $e \in$ *applicable(events, s')* **do**
          $s' \leftarrow F(s,e,0)$
    achiever(s') $\leftarrow a$
    if *isValid(s')* **then**
      insert(visited, s)
      enqueue(OPEN, s', $h(s')$)
  if *runtime limit reached* **then**
    break
return reachedGoals

as a tree traversal task allows for efficient pruning of actions which contain falsified preconditions. A single falsified precondition can prune multiple actions from being unnecessarily checked for applicability.

A **precondition node** of a precondition tree is a tuple $pn{=}(expr(pn), \mathcal{A}_{expr}(pn), C(pn))$, where $\forall a{\in}\mathcal{A}_{expr}(pn){:}expr(pn){\in}pre(a)$ is a propositional or numeric precondition expression, a finite set of actions $\mathcal{A}_{expr}(pn){\subseteq}\mathcal{A}$ which all contain the $expr(pn)$ precondition, i.e., $\forall a{\in}\mathcal{A}_{expr}(pn) : expr(pn){\in}pre(a)$. $C(pn)$ is a finite set of child precondition nodes.

Note, that for each precondition node pn, each action in $\mathcal{A}_{expr}(pn)$ also contains all preconditions from its parent nodes. Thus, for each action $a{\in}\mathcal{A}$, its set of preconditions $pre(a)$ is represented as a trajectory (a sequence of linked precondition nodes) in the precondition tree, where the final node of the trajectory contains the action a.

Algorithm 2: Precondition tree traversal to find applicable grounded actions.

Input : $s \in S$ - a planning state.
Input : $pn_{rt} = (expr(pn_{rt}), \mathcal{A}_{expr}(pn_{rt}), C(pn_{rt})$ - root node of a grounded precondition tree.
Output: APPLICABLE - list of applicable actions.

OPEN$\leftarrow \{pn_{root}\}$
APPLICABLE $\leftarrow \emptyset$
while *OPEN* **do**
  $pn \leftarrow$ pop(OPEN)
  if $s \models expr(pn)$ **then**
    foreach $a \in \mathcal{A}_{expr}(pn)$ **do**
      APPLICABLE.append(a)
    foreach $pn_c \in C(pn)$ **do**
      OPEN.append(pn_c)
return APPLICABLE

More formally, a **precondition trajectory** is a sequence of precondition nodes $\tau{=}pn_i, pn_{i+1}, ..., pn_n$, where $i{\in}\mathbb{Z}_{\geq 0}$ and $n{\in}\mathbb{Z}_{\geq 1}$. Each precondition node pn contains the following node inside its set of child nodes, $\forall pn_k{\in}\tau{:}pn_k{\in}C(pn_{k-1})$ where $k{\in}\mathbb{Z}_{\geq 1}$. The final node pn_n of any trajectory τ must contain a non-empty set of actions $\mathcal{A}_{expr}(pn_n) \neq \emptyset$. The set of preconditions $pre(a)$ for any action $a{\in}\mathcal{A}_{expr}(pn_k)$ contained inside some precondition node $pn_k{\in}\tau$ must also contain the preconditions contained by all of its predecessor nodes in the precondition trajectory, i.e., $\forall a{\in}\mathcal{A}_{expr}(pn_k){:}pre(a) = \bigcup_{i=0}^{k} expr(pn_i)$.

In a precondition tree, multiple precondition trajectories can partially overlap. In fact, a precondition trajectory of some action $a_k{\in}\mathcal{A}$ may contain a whole precondition trajectory of another action $a_j{\in}\mathcal{A}$ iff $pre(a_j) \subset pre(a_k)$. If $pre(a_j) = pre(a_k)$, they would be part of the same trajectory τ with the final precondition node pn_n containing both actions, i.e., $a_j, a_k{\in}\mathcal{A}_{expr}(pn_n)$

Given a grounded state $s{\in}S$, the mechanism traverses the precondition tree, starting at the root, evaluating the precondition expression $expr(pn_i)$ in each expanded node pn_i. If the grounded precondition is satisfied in the given state, all actions contained by that precondition node are deemed applicable, i.e., $\forall a{\in}\mathcal{A}_{expr}(pn_i) : s \models expr(pn_i) \implies s{\models}pre(a)$. Furthermore, if $s \models expr(pn_i)$ then all child nodes $C(pn_i)$ will be expanded and evaluated. Conversely, if precondition $expr(pn_i)$ is falsified in s, i.e., $s \not\models expr(pn_i)$, the entire branch containing precondition node pn_i and all of its successors is pruned away. The precondition tree traversal is shown in fig. 2.

A precondition tree is generated by iteratively inserting a precondition node pn for each precondition $expr{\in}pre(a)$ for each action $a{\in}\mathcal{A}$ in the grounded planning problem $\mathcal{P}$. Algorithm for generating a precondition tree is shown in 3.

Since all happening types (i.e., actions $\mathcal{A}$, events E, and processes P) have identical structure, the precondition tree approach is applicable to all happenings. Algorithms 3 and 2 can be adapted by replacing the set of grounded actions $\mathcal{A}$, with the set of events E or processes P. In practice, Nyx

Algorithm 3: Precondition Tree Generation

Input : $\mathcal{A}$ - a set of grounded actions $\forall a \in \mathcal{A} : a = (pre(a), eff(a))$.
Input : $pn_{rt} = (expr(pn_{rt}) = \emptyset, \mathcal{A}_{expr}(pn_{rt}) = \emptyset, C(pn_{rt}) = \emptyset)$ - an empty root precondition node.
Output: pn_{rt} - a root precondition node (containing the precondition tree).

```
foreach a ∈ A do
    pn ← pn_rt
    foreach expr ∈ pre(a) do
        pn_match ← ∅
        if C(pn) = ∅ then
            pn_match = (expr, ∅, ∅)
            C(pn).append(pn_match)
        else
            foreach pn_c ∈ C(pn) do
                if expr(pn) = expr(pn_c) then
                    pn_match ← pn_c
                    break
        pn ← pn_match
return pn_rt
```

generates a precondition tree for each type of happenings.

From a design perspective, the precondition tree is a more efficient approach instead of sequentially checking lists of preconditions to determine applicability of each grounded happening. In practice, however, the efficiency of the precondition tree is affected by the planning domains and their characteristics. Many domains have small branching factors, relatively few grounded happenings of each type, and simple precondition expressions. Under those circumstances, building the precondition tree can add unnecessary overhead. As a result, the precondition tree is an optional feature in Nyx that can be activated via a command-line flag when needed. Table 1 presents the impact of the precondition tree on search performance, measured w.r.t. the average number of explored states per second over the first 10000 states. As is evident from the data, the precondition tree approach more than doubled the state space exploration rate for the Angry Birds domain (Piotrowski et al. 2021) where 1413 grounded events need to be checked at each time tick (checking for collisions, explosions, and other phenomena between birds, pigs, blocks, and platforms). It also has the highest average number of preconditions for all happening types and the vast majority of preconditions are complex numeric expressions which are particularly time-consuming to evaluate. However, the event-driven Angry Birds model is an outlier in its complexity compared to other benchmark PDDL+ models (Fox and Long (2006); McDermott (2003); Stern et al. (2022); Howey et al. 2004).

Finally, like any approach for checking happening applicability, the precondition tree is sensitive to the ordering of preconditions. For uniformity, all sets of preconditions are sorted in lexicographical order. Though, out of scope for this paper, subsequent work will conduct an analysis of node ordering in the precondition tree to optimize its efficiency.

Nyx Expressiveness Extensions

PDDL+ is one of the most expressive planning modeling languages in use today. It is specifically designed for hybrid systems with mixed discrete and continuous dynamics. Hybrid systems are omnipresent in the real world. In fact, most realistic systems exhibit both discrete and continuous behavior. Furthermore, PDDL+ is also able to express exogenous activity in the form of processes and events (i.e., the environment's actions). Deployed intelligent agents are required to interact with real-world phenomena, making PDDL+ well-suited for modeling realistic scenarios.

However, PDDL+ models are still severely limited by the classes of mathematical expressions they can exploit. Currently, PDDL+ (and other numeric versions of PDDL) are limited to basic arithmetic operations, i.e., addition, subtraction, multiplication, and division. Thus, any significantly advanced system dynamics, that cannot be easily defined using the aforementioned operations, must be simplified or approximated. This is at odds with the real-world planning applications that require model accuracy and greater expressiveness. Indeed, even quite basic mathematical operations such as roots, absolute values, or trigonometric functions can be unfeasible to accurately encode in PDDL.

Some approximations are sufficiently accurate to exploit (e.g., Bhaskara's trigonometric approximations) but this often comes at a cost of significant reduction in model clarity, readability, and/or conciseness. Figure 2 shows the PDDL expression required to approximate $\sin\theta^\circ$ and $\cos\theta^\circ$. Modeling complex systems using bloated and overly complicated approximations for basic mathematics is cumbersome and time-consuming. Additionally, such practices are prone to introducing errors into the models. Most importantly, having to resort to using much simplified dynamics or complicated approximations may discourage users from considering AI planning approaches altogether. Nyx facilitates straightforward extensions for defining advanced system dynamics beyond the current arithmetic confines of base PDDL+.

Domain Language Extensions Nyx supports extending the expressive power of PDDL+ by integrating new mathematical expressions and operations for use directly in the planning domain. The integration of new expressions into the planner is done in a straightforward manner directly. The user is only required to add a new entry to an existing list of mathematical symbols/function names, and then define how the parser will evaluate the expression. In practice, adding a new expression requires only about 2 lines of code in one file (one specifying the new symbol and arity of the expression, the other - how to evaluate the expression in Python). Virtually any type of function can be seamlessly integrated into Nyx via Python, including specialized operators/expressions which require external libraries/packages (provided they are imported in the parser).

Additionally, Nyx natively supports the entire Python math library[2] via the `@` symbol for direct use in the planning domain file. Any function from the library can be used by adding `@` in front of the math function name and fol-

[2]https://docs.python.org/3/library/math.html

Domains	# actions (avg # preconditions)	# events (avg # preconditions)	# processes (avg # preconditions)	exploration rate (nodes/sec)		
				with PT	without PT	difference
Car	4 (2.33)	1 (3)	2 (1.5)	44971	54489	-17.5%
Sleeping Beauty	4 (1.67)	6 (2.5)	2 (1.5)	41037	54132	-24.2%
Vending Machine	3 (2)	3 (3)	1 (1)	32932	37260	-11.6%
Convoys	129 (2)	0 (0)	65 (1.98)	6305	7841	-19.6%
Cartpole	3 (4)	4 (2)	1 (2)	10846	11269	-3.7%
Mountain Car	3 (4)	6 (1.7)	0 (0)	27911	33422	-16.5%
Angry Birds	9 (5.5)	1413 (7.1)	5 (3.6)	924	422	+118.8%

Table 1: Comparison of Nyx's exploration rate (w.r.t. expanded nodes) with and without using the precondition tree (PT) approach for various PDDL+ domains. The total number of grounded actions, grounded events, and grounded processes is presented per domain. The average number of preconditions per each happening type is show in brackets.

```
sin_theta= (/ (* (* 4 (theta)) (- 180 (theta))) (- 40500 (* (theta) (- 180 (theta)))))
cos_theta= (/ (- 32400 (* 4 (* (theta) (theta)))) (+ 32400 (* (theta) (theta))))
```

Figure 2: PDDL-style implementation of $\sin(\theta°)$ and $\cos(\theta°)$ using Bhaskara's approximation.

$$|\vec{v}| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
(assign v_mag
  (^ (+ (^ (-(x2)(x1)) 2) (^ (-(y2)(y1)) 2)) 0.5))
```

Figure 3: Absolute value of some variable z and the magnitude of some vector $\vec{v}$, using Nyx's added power operator ^.

lowing the PDDL prefix notation convention. For example, `(@sin (z))` and `(@pow (x) (y))` in the domain will yield the sine of z and x to the power of y, respectively. In Python, these expressions would be evaluated as `math.sin(z)` and `math.pow(x,y)`. For convenience, the power operator is also implemented in Nyx as the ^ symbol, i.e., `(^ (x) (y))`, where x is the base number and y is the exponent. The power expressions also accept fractional exponents which allows for representation of roots. Figure 3 shows a PDDL snippet using Nyx's language extensions to compute the magnitude of a vector $|\vec{v}|$.

Semantic Attachments Real-world systems can be extraordinarily complex. Unfortunately, some realistic system dynamics cannot be defined as part of a planning domain even exploiting Nyx's domain language extensions described in the previous section. Such cases represent a significant loss of scientifically interesting scenarios to the planning community. In many cases, modeling a real-world system as a planning domain might prove unfeasible due to a single, yet important, feature of the system which cannot be encoded using conventional PDDL+. To overcome scenarios where a piece of the system dynamics is beyond the scope of expressiveness of PDDL+, Nyx is equipped to accommodate plug-in semantic attachments (Dornhege et al. 2009; Hertle et al. 2012). Semantic attachments are external functions to which the planner delegates the computation of some of the system's dynamics. Via semantic attachments, Nyx enables the integration of advanced methods and libraries which cannot be exploited otherwise. Furthermore, it facilitates a straightforward manner of integrating external functions in virtually any form, including pure Python code, trained surrogate models, functional mock-up units, and simulators, to name a few.

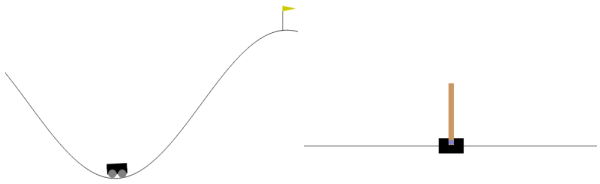

Figure 4: Modeled OpenAI Gym environments: Mountain Car (left) and Cartpole (right).

Average cost	-128.86 ± 31.399
Completion	100%
Planning time	0.0508 ± 0.005791 secs
Best RL agent cost[3]	110

Table 2: Mountain Car results over 100 episodes.

Nyx models a semantic attachment as a Python function that takes a planning state as input, updates a subset of state variables' values, and returns the modified state. In the planner, the feature is activated by simply adding `:semantic-attachment` to the domain requirements.

Results

In this section we evaluate Nyx's capabilities by modeling two of the OpenAI Gym games(Brockman et al. 2016), Mountain Car and Cartpole, as PDDL+ domains for the first time. The models are based on the OpenAI simulator descriptions(OpenAI 2016b,a). The initial state is observed after initializing the OpenAI Gym simulator, and the new problem file with the pre-defined transition model in PDDL+ is solved in Nyx. The planner utilizes domain-specific heuristics to solve the problem. All the results have been run on Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz.

Mountain Car is an OpenAI Gym problem, in which a cart has to be pushed on a sinusoidal curve to a specific height. There are three possible actions in each position – push car left, push car right, do nothing. Observation from the simulator returns position and velocity of the cart (in x direction only). An episode ends either when the cart reaches

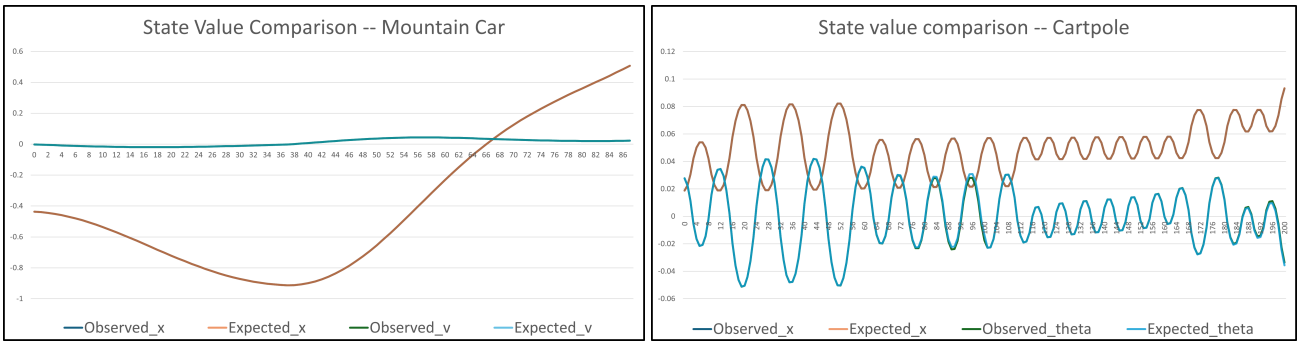

Figure 5: Modeling accuracy comparing state variables with the feature vector from OpenAI Gym. Mountain Car (left): position (x) and velocity (v). Cartpole (right): position of cart (x) and angular displacement of pole (θ).

Average Reward	200 ± 0.0
Completion	100%
Planning time	0.21669 ± 0.021982 secs
Best RL agent reward[4]	195

Table 3: Cartpole results over 100 episodes.

the flag position (0.5) or the episode is truncated in 200 time steps. Table 2 presents results over 100 episodes. Our heuristic was calculated using – h_{mc}=$0.07-1000*f_{app}*v_{cart}$ where h_{mc} represents the heuristic value, f_{app} is the force applied to the cart with direction and v_{cart} is the velocity of the cart. In essence, it prioritizes states in which the force applied was in the direction of the motion to increase velocity. The agent is penalized (-1) for each action taken. Current state-of-the-art RL agents achieve an average score of -110, whereas our agents achieved an average score of -128.86.

Cartpole is commonly known as inverted pendulum, in which the agent has to balance a pole on the cart. A state in the Cartpole is represented as a vector with values of position and velocity of the cart and angle of divergence, with angular velocity of the top point of the pole. There are two possible actions – push cart left, and push cart right. An episode ends if the agent is able to balance the pole for 200 steps (or 4.0 seconds) or if the angular displacement of the pole is greater than $12°$, or position of the cart has diverged greater than $2.4m$ from the center. The score per episode is the number of steps (out of 200) that the agent is able to balance the pole without violating the termination constraints. Table 3 shows the average score and the completion rate in comparison to best RL agent available. A domain-specific heuristic is calculated as $h_{cp} = \sqrt{x^2+\dot{x}^2+\ddot{x}^2+\theta^2+\dot{\theta}^2+\ddot{\theta}^2}*(t_{ep}-t_{elapsed})$ where x represents the position of the cart and θ represents the angular displacement, $\dot{x}$ represents velocity, $\ddot{x}$ represents acceleration, and t_{ep} is the duration of the episode, and $t_{elapsed}$ is the time elapsed. The heuristic prioritizes safe, controllable states (i.e., with low velocities/accelerations, and far from termination boundaries) that are further in time.

State modeling comparison. To evaluate the modeling capability of Nyx, we compare the expected and observed values of the state predicates as calculated by Nyx and the simulator. Figure 5 (left) shows the comparison observed and expected values of of position (x) and velocity (v) in mountain car. The values differ from each other in the order of magnitude of 10^{-9} for x and 10^{-10} for v. Thus, the lines for observed and expected value are overlapping in the figure. Similarly, figure 5 (right) shows the comparison of observed and expected values of position of cart (x) and angular displacement of pole (θ). Nyx is able to model Cartpole accurately to the order of 10^{-3}, that is far greater than the mountain car values especially for rotational functions. This difference accumulates over time for the θ value of the pole. Thus, there is a need to replan after 100 steps as the error in difference accumulates and the transition values differ causing the pole to fall early. Without replanning the average reward for Cartpole is approximately 157 points.

Conclusion & Future Work

This paper presented Nyx, a novel PDDL+ planner for real-world planning problems. Currently available PDDL+ planners have a steep learning curve and require expert knowledge. Nyx aims to increase the accessibility to AI planning, particularly for realistic feature-rich domains. Nyx's design facilitates adaptability to tackle novel classes of domains, and accessibility to promote AI Planning as a viable and usable method for solving interesting real-world problems. We also present the precondition tree, a promising new approach for efficiently evaluating preconditions. Nyx introduces features that support reasoning advanced features beyond the scope of PDDL+. Specifically, Nyx allows straightforward implementation of new mathematical expressions to be used in PDDL+ domains, as well as support for semantic attachments for features that cannot be feasibly included in the PDDL model directly. By discussing the fuel system model, we show that Nyx enables AI Planning to reason with real-world problems. In future work, we will continue exploring different configurations of the precondition tree and analyze impact of different orderings of preconditions in the tree. Most importantly, since developing model-specific heuristics for complex real-world problems is difficult and requires expert knowledge, further work will focus on developing domain-independent approaches that can readily solve emerging classes of problems.

References

Alur, R.; Courcoubetis, C.; Halbwachs, N.; Henzinger, T.; Ho, P.; Nicolin, X.; Olivero, A.; Sifakis, J.; and Yovine, S. 1995. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138: 3–34.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. arXiv:1606.01540.

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

Della Penna, G.; Intrigila, B.; Magazzeni, D.; and Mercorio, F. 2010. A PDDL+ benchmark problem: The batch chemical plant. In *ICAPS*. Citeseer.

Della Penna, G.; Magazzeni, D.; Mercorio, F.; and Intrigila, B. 2009. UPMurphi: a tool for universal planning on PDDL+ problems. In *ICAPS*.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Nineteenth International Conference on Automated Planning and Scheduling*.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

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

Fox, M.; and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27: 235–297.

Fox, M.; Long, D.; and Magazzeni, D. 2012. Plan-based Policies for Efficient Multiple Battery Load Management. *J. Artif. Intell. Res. (JAIR)*, 44: 335–382.

Fox, M.; Long, D.; Tamboise, G.; and Isangulov, R. 2018. Creating and executing a well construction/operation plan. US Patent App. 15/541,381.

Fredkin, E. 1960. Trie memory. *Communications of the ACM*, 3(9): 490–499.

Hansen, E. A.; Zilberstein, S.; and Danilchenko, V. A. 1997. Anytime heuristic search: First results. *Univ. Massachusetts, Amherst, MA, Tech. Rep*, 50.

Henzinger, T. A. 2000. The theory of hybrid automata. In *Verification of digital and hybrid systems*, 265–292. Springer.

Hertle, A.; Dornhege, C.; Keller, T.; and Nebel, B. 2012. Planning with Semantic Attachments: An Object-Oriented View. volume 242.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *IEEE ICTAI*, 294–301. IEEE.

IEEE Spectrum. 2022. Top Programming Languages.

Kiam, J. J.; Scala, E.; Javega, M. R.; and Schulte, A. 2020. An AI-based planning framework for HAPS in a time-varying environment. In *ICAPS*, volume 30, 412–420.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language.

McDermott, D. V. 2003. Reasoning about Autonomous Processes in an Estimated-Regression Planner. In *ICAPS*, 143–152.

OpenAI. 2016a. CartPole. https://gymnasium.farama.org/environments/classic_control/cart_pole/. [Online; accessed 04-April-2024].

OpenAI. 2016b. Mountain Car. https://gymnasium.farama.org/environments/classic_control/mountain_car/. [Online; accessed 04-April-2024].

Pednault, E. P. 1989. ADL: Exploring the Middle Ground Between. In *International Conference on Principles of Knowledge Representation and Reasoning*, 324.

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

Piotrowski, W.; Stern, R.; Klenk, M.; Perez, A.; Mohan, S.; et al. 2021. Playing Angry Birds with a Domain-Independent PDDL+ Planner. In *International Conference on Automated Planning and Scheduling (Demo Track)*.

Piotrowski, W. M. 2018. *Heuristics for AI Planning in Hybrid Systems*. Ph.D. thesis, King's College London.

Roberts, M.; Piotrowski, W.; Bevan, P.; Aha, D.; Fox, M.; Long, D.; and Magazzeni, D. 2017. Automated planning with goal reasoning in minecraft. In *Proceedings of ICAPS workshop on Integrated Execution of Planning and Acting*.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 655–663.

Stern, R.; Piotrowski, W.; Klenk, M.; de Kleer, J.; Perez, A.; Le, J.; and Mohan, S. 2022. Model-Based Adaptation to Novelty in Open-World AI. *Bridging the Gap Between AI Planning and Reinforcement Learning (PRL), ICAPS*.

TIOBE. 2022. TIOBE Index.

Vallati, M.; Magazzeni, D.; Schutter, B. D.; Chrpa, L.; and Mccluskey, T. L. 2016. Efficient Macroscopic Urban Traffic Models for Reducing Congestion: A PDDL+ Planning Approach. In *AAAI Conference*. AAAI Press.

Vouros, G.; Papadopoulos, G.; Bastas, A.; Cordero Garcia, J.; and Rodrigez, R. 2022. Automating the resolution of flight conflicts: Deep reinforcement learning in service of air traffic controllers. In *Prestigious Applications of Intelligent Systems, IJCAI*.