OOMPA 2025.08: A First Cut of the Toolkit for Object-Oriented Modeling for Planning and Acting

Mark Roberts[1*] **David H. Chan**[2] **Dana S. Nau**[3] **Jamie C. Macbeth**[4]

[1]Iconium Labs, Tempe, AZ, USA | makro@iconiumlabs.com
[2]U.S. Naval Research Laboratory, Washington, DC, USA | david.h.chan4.civ@us.navy.mil
[3]University of Maryland, College Park, MD, USA | nau@umd.edu
[4]Smith College, Northampton, MA, USA | jmacbeth@smith.edu

Abstract

OOMPA is a partially implemented Python 3.13+ toolkit for modeling hierarchical planning domains as annotated Python classes, without writing a separate PDDL or HDDL domain file. State properties, actions, and hierarchical methods attach to domain classes via decorator syntax; OOMPA projects the resulting model into a flat dictionary of dictionaries, like the Pyhop family of planners. We describe OOMPA's motivation and architecture as we demonstrate its use in a restaurant planning domain. There are many unrealized features, so we end with a discussion of limitations and future work.

Code — https://github.com/makro-ilab/oompa-2025_08

1 Introduction

Modeling a planning domain typically requires expertise in domain-specific knowledge and a formal planning language. Planning languages—e.g., PDDL for classical planning (Haslum et al. 2019), HDDL for hierarchical planning (Höller et al. 2020), among many others—represent the world through typed predicates and lifted rules, a logic-centered formalism that is foreign to most software developers. Developers will often understand a domain's structure in terms of its objects and their properties, operations, or interactions. But they must translate that understanding into a planning representation, which may introduce large gaps between the conceptual and planning models or result in challenges with maintenance or explanation.

OOMPA is a Python 3.13+ toolkit to help overcome some of these challenges by modeling a planning domain in Python. An OOMPA planning domain is a set of standard Python classes augmented with type-hinted decorators that use 3.13 features but still work much like the Python `@property`. Type attributes and relations (i.e., predicates) are declared as decorated class fields or as direct variables, if distinct from a class (e.g., a relation of multiple objects). Actions and hierarchy are declared as decorated class methods using subdecorators analogous to `@property.setter` and `@property.getter`. State is a flat Python dictionary of dictionaries, leveraging a clever design from GTPyhop (Nau et al. 2021). Finally, OOMPA leverages Python type hinting, which facilitates introspection to automatically construct a planning domain for analysis or search. Developers can model a planning domain directly in Python, though they must still reason in planning terms—states, actions, preconditions, effects, and methods.

OOMPA's design is inspired by two systems. It is the successor to ActorSim (Roberts et al. 2021; Johnson et al. 2016), a Java-based system for goal-oriented agent modeling. OOMPA ports ActorSim's best design principles to Python. It borrows ideas from GTPyhop (Nau et al. 2021), which also encodes planning domains in Python. But OOMPA distinguishes action preconditions from effects, enforces declarative conditions and effects (more like PDDL than unrestricted Python), and performs introspection using the type hinting system of Python 3.13.

OOMPA complements existing languages and targets Python developers. Its language defines a restricted fragment of Python that should more directly translate to PDDL or HDDL; a planned export path should also allow OOMPA domains to be compiled to existing languages provided the correct OOMPA features align with a planning language's fragments (e.g., temporal, numeric, hierarchical). After providing a running example (§2) and background (§3), we:

- describe OOMPA, including the decorator-based API for state properties, actions, and hierarchical methods (§4);
- review a *draft* encoding of a restaurant planning domain as a hierarchical goal network along with a test harness demonstration of the problem solution (§5);
- discuss the limitations of OOMPA 2025.08 (§6);
- provide a detailed comparison of OOMPA to related work (§7); and provide an Appendix of full examples.

2 Running Example

We base our demonstration on the encoding developed in prior work (Macbeth et al. 2025) that involves a patron eating at a restaurant. This example is drawn from the literature on scripts. A *script* (Schank and Abelson 1975) is a stereotyped knowledge structure describing the sequence of events for common situations. Schank and Abelson developed scripts to explain how people use stored background knowledge to interpret stories and predict unstated steps—the idea being that background knowledge often augments explicitly stated detail. Scripts became an influential vehicle for studying knowledge representation in cognitive and computational systems.

[*]Work performed at the U.S. Naval Research Laboratory.

Table 1: Summary of the restaurant script: principal roles, their attributes, and their actions and interactions.

Role	Attributes	Actions / Interactions
Patron	table, server, menu, order, money, hunger	Arrives and is seated; reviews menu; requests server; places order; eats; pays and exits
Server	assigned table, proximity to patron	Takes patron's order; relays it to the cook; delivers food; presents bill
Cook	current order	Receives order from server; prepares dish; signals when ready
Shared objects: Table (server, patron, menu, occupied); Menu (items, special); OrderedItem (status); Dish		

The restaurant script (Schank and Abelson 1975) describes a common patron experience at a restaurant: arriving, being seated, ordering food, eating, paying, and leaving. But in the context of story understanding, a reader uses a script to backfill the statement "the patron went to a restaurant and hurriedly left after paying the bill" to imply that food was probably ordered and served; Schank and Abelson argue that scripts explain why that inference is possible.

Table 1 summarizes the principal roles, their attributes, and their interactions. The script involves three principal actors—a patron, a server, and a cook—who interact through shared objects: a table, a menu, menu items, orders, and prepared dishes. Each actor or object carries properties that change as the scenario unfolds:

- the patron arrives hungry, is assigned a table and server, reviews the menu, places an order, eats and becomes sated, and eventually pays and leaves;
- the server takes orders and delivers food; and
- the cook manages the available meals, prepares dishes, and notifies the server.

Hierarchical structure is evident: the scenario plays out in four scenes—Entering, Ordering, Eating, and Exiting—each with clear starting conditions and expected outcomes.

The roles, objects, and interactions of this script are much more natural to implement in an object-oriented system. Here, the domain engineer—mainly the primary author—did not begin by writing PDDL typed objects, predicates, and operators. Instead, the natural starting point was a set of classes—one per role—with properties for the state each object carries, class methods (typical Python functions, which are distinguished from hierarchical methods) for the actions and interactions, and hierarchical methods for decomposition. At this point, the domain closely resembled a GTPyhop domain, with Python type hints and annotated OOMPA primitives that describe planning details. It may be the case that many developers follow a path of working out the object-oriented design and then converting that to PDDL.

3 Background

OOMPA's design builds on two concepts from automated planning—the state-variable representation and hierarchical planning—plus the Python descriptor protocol.

3.1 State Representation

Automated planning systems model the world as an implicit state transition system (Ghallab, Nau, and Traverso 2025). The dominant encoding is predicate-based, as in PDDL: having the server `sam` assigned to a table `table1` could be expressed as the ground atom `(server table1 sam)` (i.e., the `server` of `table1` is `sam`). State variables offer an alternative: `table1.server == sam`. The two representations are expressively equivalent and interconvertible in polynomial time (Ghallab, Nau, and Traverso 2025). However, state variables more closely follow the syntax of most programming languages, making them a natural fit for an OO domain modeling toolkit. In systems that use state variables, a state is a set of (state variable, value) pairs.

GTPyhop (Nau et al. 2021), described below, projects a set of state variables into a flat dictionary of dictionaries: $s[attr][obj] = value$. Suppose `sam` has prepared `table1` for breakfast. A pretty-print of that state might be:

```
1 s_table1:
2   server: {'table1': sam}
3   menu: {'table1': menu_breakfast}
4   occupied: {'table1': False}
```

If `pat` is seated then `s_table1['occupied']['table1']` would have a value of `True`. A benefit of using Python's native dictionaries over a custom struct is that they are fast and support frozen (immutable, hashable) copies for search algorithms that require cycle detection.

3.2 Hierarchical Planning

A hierarchical planning system (Nau et al. 1999, 2003) adds domain knowledge as *methods* that decompose abstract goals into subgoals and primitive actions, improving search efficiency. Hierarchical task network (HTN) planning is more expressive than classical planning (Erol, Hendler, and Nau 1994). *Hierarchical Goal Networks* (HGNs, Shivashankar et al. 2012, 2013) are a variant in which methods decompose *goals*—boolean or numeric conditions on the state—rather than abstract tasks. HGNs and HTNs are formally equivalent (Alford et al. 2016), but HGNs integrate with classical planning. Goal-Task Networks (GTNs) combine both formalisms (Ghallab, Nau, and Traverso 2025).

GTPyhop (Nau et al. 2021) is a Python GTN planner that uses state variables and state dictionaries. Actions are pure Python functions that take the mutable state as a parameter. Figure 1 shows operators from GTPyhop's SimpleTravel domain, where a is the person, x is the start, and y is the destination. Note that: 1) actions can call arbitrary code to calculate values, as in Line 13; 2) the state is modified inline and immediately returned; 3) the use of a dictionary makes the code easy to understand or modify (`state.loc[obj]` is equivalent to `state['loc'][obj]`). The simplicity of this design is its strength; Python developers can rapidly prototype a domain, and teaching this tool is straightforward. The drawback is that integrating this with a PDDL or HDDL planning system or into an acting system, both of which we tried, requires a parser that reads arbitrary Python and produces a structured planning language. Converting arbitrary programming code to a structured planning representation is non-trivial (see Fine-Morris et al. 2025; Sikes et al. 2025).

```python
def walk(state, a, x, y):
    if state.loc[a] == x:  # if a is at x
        state.loc[a] = y   # assign a to y
        return state       # return revised state
    else: return False     # otherwise not applicable

def ride_taxi(state, a, x, y):
    # if the taxi and person are at start
    if state.loc['taxi'] == x and state.loc[a] == x:
        state.loc['taxi'] = y  # move taxi to y
        state.loc[a] = y       # move a to y
        # assign a.owe a _calculated_ value
        state.owe[a] = taxi_rate(state.dist[x][y])
        return state    # return revised state
    else: return False  # otherwise not applicable
```

Figure 1: GTPyhop `walk` and `ride_taxi` operators from the SimpleTravel domain. Actions are plain Python functions that mutate the state dictionary directly.

```python
class Circle:
    radius: float = property() # class style

    def __init__(self, diameter: float):
        self._diameter = diameter

    # @property    # decorator style
    # def radius(self): pass

    # custom getter: derives from diameter
    @radius.getter
    def radius(self) -> float:
        return self._diameter / 2.0

    # setter: called on c.radius = v
    @radius.setter
    def radius(self, value: float):
        if value < 0:
            raise ValueError
        self._diameter = value * 2.0
```

Figure 2: Python `@property` descriptor pattern. OOMPA uses a similar pattern to annotate state variables, actions, and methods.

3.3 A Primer on Python's Descriptor Protocol

OOMPA heavily uses Python's descriptor protocols, sometimes referred to as descriptors. A *descriptor protocol* is code that customizes how a class member is accessed, mutated, or deleted. Writing custom descriptors is an intermediate topic (see https://docs.python.org/3.13/howto/descriptor.html), but every Python programmer has already used one: `@property` is a built-in descriptor.

Figure 2 shows the pattern, where `radius` behaves like a plain attribute from the caller's perspective while `@radius.getter` and `@radius.setter` intercept reads and writes. OOMPA generalizes this pattern: `StatePropertyFactory`, `@OompaAction`, and `@OompaMethod` each add planning semantics alongside getter/setter behavior. Properties are declared using either the more common decorator style (Lines 7–8) or the class-level call style (Line 2). Thinking of OOMPA primitives as customized variations of `@property` that also carry planning semantics is the right mental model.

4 The OOMPA Language

An OOMPA domain is organized as a set of Python classes, each representing a domain type or role; in the restaurant example these are `Patron`, `Server`, `Cook`, `Table`, `Menu`, `MenuItem`, `OrderedItem`, and `Dish`. Planning-specific details—state variables, actions, and hierarchical methods—attach to these classes via OOMPA decorators that implement specialized descriptor protocols.

4.1 Types and State Variables

Here is the declaration of a `Table` type.

```python
class Table(AbstractNamed, HasStateProperties):
    occupied: bool = StatePropertyFactory(False)
    menu: Menu|None = StatePropertyFactory(None)
    server: Server = StatePropertyFactory()
```

A `Table` is a standard Python class that inherits template code. `AbstractNamed` provides a string name and typing information. This is analogous to declaring types in `domain.pddl` and objects in `problem.pddl`.

The mixin `HasStateProperties` declares that this type has state variables. `Table` has three: a boolean indicating whether it is occupied (Line 2, initially `False`), a menu slot for a `Menu` object (Line 3, initially `None`), and a server slot not yet assigned (Line 4).

An OOMPA state variable is called a `StateProperty` because it works similarly to the Python `@property` decorator and for historical reasons related to ActorSim. A `StatePropertyFactory` is a convenience class that:

- captures the typing information of the state variable,
- implements `.setter` code for the class to validate that value assignments are the correct type,
- implements `.getter` code that correctly retrieves the value of a type instance regardless of whether it is held in a state dictionary or directly in a Python object,
- provides hooks for automatically transforming the state variable into a state dictionary or predicate, and
- supports some features from `@dataclass` like custom initialization via the `default_factory`.

`StateProperty` can declare attributes or relations; it may help to think of them both as predicates with different arity. A `StateProperty` declared for a single object, like `table.occupied`, is a unary relation with a single argument: `self`. An n-ary relation extends the parameter list beyond just `self`. A more detailed example can be seen in the `Map.distance` binary relation of the SimpleTravel domain in Appendix B, Line 95.

A `StateProperty` is not required to be in a class, though that will often be convenient. For example, `distance` or `road` relate locations and can be held in a `Map`. Alternatively, these could be in the Domain, the Location class, or even the global scope. OOMPA supports organizing a domain by classes but does not force facts (e.g., `between(a, b, c)`) into classes; the modeler simply declares them and optionally adds an @-annotation to link an owner.

To make this concrete, Figure 3 extends the restaurant domain with a *party*, a group of patrons dining together. In Figure 3, `patron.party` is a `StateProperty` that links a patron to a `Party` object (Line 10). `Party.size` is a plain Python `@property` computed from the member list—

```
1  class Party(AbstractNamed, HasStateProperties):
2      members: list[Patron] = StatePropertyFactory(
       default_factory=list)
3
4      @property   # plain property, not StateProperty
5      def size(self) -> int:
6          return len(self.members)
7
8  class Patron(Person, ...):
9      ...
10     party: Party | None = StatePropertyFactory(None)
```

Figure 3: `Party` and `Patron` class declarations. `Party.size` is a plain `@property`; `patron.party` (Line 10) is a `StateProperty` visible to the planner.

OOMPA does not require every computed attribute to be a `StateProperty`, only those that participate in the planning state and must be visible to the planner.

OOMPA natively supports typing and a `StateProperty` can hold any type; this subtlety is important and draws on a major benefit of planning systems like GTPyhop. `Party.members` holds a list of patrons, but it could be any valid Python type. There is no limit beyond what would be reasonable for a planning domain. For example, numerical planning is natively supported, as can be seen by the addition of numeric distance to the roads in a map in the Simple-Travel domain (see Appendix B, Line 95).

4.2 Conditions and Effects

OOMPA expresses conditions and effects in a manner closer to PDDL than the free-form Python of GTPyhop. As mentioned, allowing arbitrary Python code in preconditions and effects creates challenges for translating code to a planning model or for analyzing the underlying semantics. OOMPA resolves this expressive ambiguity by requiring declarative expressions for preconditions and effects. Specifically, OOMPA restricts the available operations to those listed in Table 2. Some of these operators are typical for planning languages; OOMPA adds container operators and is designed to be extensible. `StateProperty` exposes these operators as a fluent interface (i.e., method chaining using dotted access to operator names) to overcome Python's limited operator overloading.[1] An example of this will be discussed in the next section along with Figure 4.

4.3 Actions

Figure 4 shows a condensed version of the `Patron` class from the restaurant domain; we refer to it throughout this section. The full description is provided in Appendix A, which, beyond the two state properties shown, also declares state properties for `money`, `is_hungry`, `is_pleased`, `menu`, and `desired_order`. The base class for actions has methods to check applicability and to apply the action to a state.

In Figure 4, the `Patron` class (Line 2) inherits from the simple type `Person`, which is just an `AbstractNamed` type with no attributes; this supports a type hierarchy. It `HasStateProperties` for a `table` and `server` (Lines 3–4). It also `HasOompaActions` and `HasOompaMethods`, which we discuss next. `CreatesNewObjects` declares that this class has actions that introduce new objects when executed; in this case, the patron creates an ordered item from the special on the menu.

Actions are declared on the class that performs them using `@OompaAction`, with separate `.precondition` and `.effect` subdecorator functions. The `sit` action (Lines 6–16) illustrates this pattern. Lines 6–7 declare `sit`'s name and parameters with a decorator style `@OompaAction`. Lines 9–10 declare `sit.precondition` checking that `self.table` is `None`, while Lines 12–16 declare `sit.effect`, which assigns the table and server to the patron and marks the table as occupied.

Table 2: OOMPA operators for conditions and effects.

Category	Operators
Compound Conditions:	`or and`
Preconditions:	
Value comparison	`equals not-equals less-than lt-equals greater-than gt-equals`
Container membership	`contains not-contains`
Container cardinality	`fewer fewer-or-equal more more-or-equal`
Effects:	
Value assignment	`assigned increased-by decreased-by`
Container mutation	`inserts removes appends pushes pops`

4.4 Methods

OOMPA expresses HTN/HGN methods (Section 3) using `@OompaMethod`, with the same multi-stage decorator pattern as actions. Each method specifies a `.goal` for the condition it achieves, a `.precondition` that must hold for it to be applicable, and a `.body` that describes the (ordered) decomposition into subtasks/subgoals and actions.

Returning to Figure 4, the `m_ordering` method (Lines 21–35) illustrates this. Its goal is that the patron's desired order is non-`None` and has status `ORDERED`. Its precondition requires no existing order and a menu on the table. Its body is a `TotalOrderGoalTaskNetwork` containing four actions in sequence.

In the most expansive version of the OOMPA language, it is intended to support both total-order and partial-order goal task networks (GTNs). In practice, the implementation of the language may not support all fragments.

4.5 Discussion of the OOMPA Language

OOMPA's key strengths include separating action preconditions from effects, separating hierarchical methods in a similar way, forcing declarative conditions and effects, and lever-

[1] While Python supports operator overloading, a developer cannot easily override some operators critical to planning domains, namely logical operators (`and`, `or`, `not`), identity (`is`), and assignment (=).

```
1  # SPF = StatePropertyFactory
2  class Patron(Person, HasStateProperties, HasOompaActions, HasOompaMethods, CreatesNewObjects):
3    table: Table | None = SPF(None)
4    server: Server | None = SPF(None)
5
6    @OompaAction  # declares action ---------------------------------------
7    def sit(self, table: Table): pass
8
9    @sit.precondition  # condition for this action to be applicable
10   def sit(self, table: Table): return self.table.equals(None)
11
12   @sit.effect  # change in state from applying this action to state
13   def sit(self, table: Table):
14     return AndEffect( self.table.assigned(table),  # Patron seated at table
15                       self.server.assigned(table.server),  # server assigned
16                       table.occupied.assigned(True))  # table occupied
17
18   @OompaMethod  # declares method ---------------------------------------
19   def m_ordering(self, table: Table) -> GoalMethod: pass
20
21   @m_ordering.goal  # the condition the method matches; determines relevance
22   def m_ordering(self, table: Table) -> Condition:
23     return AndCondition( self.desired_order.not_equals(None),
24                          self.desired_order.status.equals(ORDERED))
25
26   @m_ordering.precondition  # condition for this method to be applicable
27   def m_ordering(self, table: Table) -> Condition:
28     return AndCondition( self.table.not_equals(None),  # Patron seated
29                          table.menu.not_equals(None),  # Patron has menu
30                          self.desired_order.equals(None),) # Patron lacks desired order
31
32   @m_ordering.body  # change in hgn from applying this method to hgn
33   def m_ordering(self, table: Table) -> TOHGN:
34     return TOHGN( self.pickup_menu(), self.review_menu_for_special(),
35                   self.request_server(), self.place_order() )
```

Figure 4: A condensed version of the `Patron` class, showing state properties (Lines 3–4), the `sit` action (Lines 6–16), and the `m_ordering` method (Lines 21–35).

aging Python's type hinting system. These promote several benefits: type checking during assignment or instantiation, flexible binding, deferred instantiation, and instance caching. It also separates checking applicability from action application; this is similar to most planning systems but not something GTPyhop actions can support.

OOMPA defers action instance construction until it is actually used and caches each instance from the (potentially partially) bound instance tuple. As far as we are aware, this is distinct from most other (non-lifted) planning systems, which either ground the entire instance in a preprocessing step or, in the case of the Pyhop family, do not need bindings at all because actions directly modify state. Lazy binding with caching can boost performance for certain algorithms; in a test run of an MCTS variation, these features resulted in faster rollouts. OOMPA may scale to very large domains, but verifying this claim is left for future work.

Along with these limited operators, OOMPA actions or methods do not take the state as a parameter—actions only declare their direct interactions with other objects. This is much closer to a PDDL style of writing actions, except that `self` is now going to be the first parameter. A benefit of this approach is that it reduces the arity of most actions by at least one. But architecturally, this is a desirable design principle that forces objects to declare their dependencies in their interface.

This set of design decisions deviates substantially from most known planning languages—OOMPA sits somewhere between a pure programming language and a language designed from a logic background. Like the attribute and action annotations described earlier, this design burdens the developer. We will discuss this limitation in Section 6 and propose some workarounds.

5 Planning and Acting In OOMPA

We demonstrate OOMPA using the restaurant example (§2).

Domain and Problem. A `Domain` class registers types using `declare_types(Table, Patron, ...)` (cf. A.2, Lines 5–10). This call automatically reviews each Type for `StateProperty`, `@OompaAction`, and `@OompaMethod` declarations. A `Problem` subclass instantiates the domain with concrete objects and the goal. A planner uses `get_applicable_actions(state)` to enumerate all ground action instances applicable in a given state, providing the primary interface for planning algorithms.

Initial state. Figure 5 shows the initial state s_0 for the restaurant problem: patron `pat` is hungry and has some money but has not yet been seated; cook `chris` is present and ready; the special `omelette` costs 12 units and is still available.

State objects are constructed automatically from the annotations on objects. Such an unpacking of the state is natural

```
s_0:
  check: {'sam': None}
  cost: {'omelette': 12}
  desired_order: {'pat': None}
  is_hungry: {'pat': True}
  is_pleased: {'pat': False}
  items: {'menu_breakfast': []}
  menu: {'pat': None, 'sam': None,
         'table1': menu_breakfast}
  money: {'pat': 50}
  near_to: {'sam': None}
  occupied: {'table1': False}
  order: {'sam': None}
  ordered: {'chris': None}
  prepared: {'chris': []}
  server: {'pat': None, 'table1': sam}
  special: {'menu_breakfast': omelette}
  status: {'omelette': AVAILABLE}
  table: {'pat': None}
```

Figure 5: Initial state s_0 for the restaurant problem.

```
s_final: desired_order: {'pat': ordered_omelette}
         menu: {'pat': menu_breakfast}
         near_to: {'sam': pat       }
         occupied: {'table1': True}
         order: {'sam': ordered_omelette}
         server: {'pat': sam, 'table1': sam}
         status: {'ordered_omelette': 'ORDERED'}
         table: {'pat': table1}
```

Figure 6: State changes after executing the plan produced by `m_get_seated` and `m_ordering`.

for a planning expert—it simply expresses the logical primitives in preparation for manipulating the state during search; it is also a compact and computationally efficient way to manage state. But this kind of "unpacking" is anathema to most OO developers—it violates encapsulation, breaks strongly typed objects (even if Python only supports type hinting), and appears disorderly even if it is not. OOMPA straddles both perspectives by only requiring that objects be annotated so the translation can be done automatically.

While automated state construction may be handy, requiring objects to be annotated in this fashion adds a burden to the OO developer's workflow. Section 6 will discuss some of those challenges and provide some possible workarounds.

Manipulating State in OOMPA. Rather than implementing a planner, we display the result of manually applying methods and actions. Appendix A.1 contains the test harness.

For the goal `desired_order(pat)` $\neq$ *None* $\wedge$ `status(desired_order(pat))` $=$ ORDERED, we select `m_get_seated` and `m_ordering` and return the plan: `sit(pat, table1)`, `pickup_menu(pat)`, `review_menu_for_special(pat)`, `request_server(pat)`, `place_order(pat)`.

Figure 6 shows the state changes after executing this plan. The patron is seated at `table1`, holds `menu_breakfast`, the server `sam` is adjacent to the patron, and `ordered_omelette` has status `ORDERED`. This corresponds to the completion of the restaurant script's Entering and Ordering scenes (Schank and Abelson 1977).

Support for Planning and Acting. Although we do not highlight it here, OOMPA is designed to support a more seamless integration of a planner into an acting system. Ideally, OOMPA would represent the objects of a simulator directly in Python, use the annotated classes to perform planning, and then command the simulator from the actions themselves. OOMPA's design has an `@OompaAction.execute` subdecorator that provides this linkage, although the feature has not been tested. OOMPA supports open-world semantics, which can be central to acting in some environments.

6 Limitations

OOMPA is in active development, so features that are mentioned in this paper may only just be planned or partially implemented. This alone probably precludes its use in an extended project, and we welcome feedback on which features would be most useful to the community.

A major limitation of the OOMPA language is its dependence on planning-oriented primitives. OOMPA has certainly made it easier to embed such primitives during prototyping of a *new* domain. However, its current design raises new concerns: (1) it is unclear how to integrate OOMPA with legacy systems; (2) it still burdens the developer, who must now consider the planning scope in addition to other concerns; and (3) it may impact design decisions because its primitives (or quirks) may require workarounds. A well-designed plugin ecosystem may ameliorate some of these.

Generative AI systems may also help resolve these challenges, at least partially. For example, a coding assistant might be able to provide first drafts that hook OOMPA into an existing system. However, the key challenge for such assistants will be needing good examples that link OOMPA primitives to planning languages. Getting that bootstrapped may take substantial effort.

Recent results suggest LLMs are getting better at producing PDDL (Tantakoun, Zhu, and Muise 2025). While there are ample PDDL examples[2] with tools to support its use[3] and even a book describing core principles (Haslum et al. 2019), there is vastly more Python code. Agentic systems may have an easier time annotating Python with planning-oriented structure than creating a PDDL domain from scratch. Or maybe *smaller* agentic workflows may get further with such an approach. OOMPA may facilitate a more principled use of agentic workflows to encode planning domains simply because it uses Python. As a preliminary demonstration, Appendix D shows algorithm listings for (untested) variations of A* (Appendix D.2) and the Goal Decomposition Planner (Shivashankar et al. 2012) (Appendix D.1).

State-of-the-art performance in automated planning has evolved over nearly fifty years. Recent tools have already taken a strong, perhaps sufficient, step on the path of making planning more accessible. Relatively recent tools like *planning.domains* and the Unified Planning Framework (Andrea Micheli et al. 2025) build on a long line of work in the area

[2]https://www.icaps-conference.org/competitions/

[3]http://planning.domains/

of knowledge engineering. It is not clear whether OOMPA or its design principles will add to this body of work.

OOMPA's relationship to the benchmarks of the International Planning Competition needs more clarity and work. The translation of OOMPA *to* PDDL should be straightforward because OOMPA restricts Python to the same underlying logical primitives as PDDL and variants. But this has not been proven, which is necessary future work.

However, the translation *from* PDDL is less straightforward. First, the natural state of a PDDL model is an "unpacked" version of the objects, attributes, and interactions. Encapsulation and interaction contracts are held within the action specification, not objects. Second, there are many subtle tricks—some have called them hacks—to get a PDDL model to work well within a planner (cf. Haslum et al. 2019). The idiosyncrasies of PDDL mean that the canonical model of a domain may be somewhat distant from an equivalent object-oriented model with similar behavior.

7 Related Work

The Pyhop Family. Pyhop[4] uses the same HTN planning algorithm as in the SHOP planner (Nau et al. 1999), but adapted for use with Python rather than Lisp. States are Python objects that hold arbitrary data structures; actions are Python functions in which the preconditions are ordinary `if`-statements; and HTN methods are Python structures.

GTPyhop (Nau et al. 2021) generalizes Pyhop to handle both tasks and goals (a goal-task network, or GTN). It adds a `Domain` class to support multiple domains in memory, unigoal and multigoal methods for goal-directed planning, and a `run_lazy_lookahead` acting loop for execution in nondeterministic environments. OOMPA adopts GTPyhop's state representation—a flat dict-of-dicts indexed by attribute and object. GTPyhop writes operators and methods as free Python functions, whereas OOMPA attaches them to the objects they describe via decorators, supporting a more encapsulated OO design.

IPyHOP (Bansod et al. 2022) extends GTPyhop to support replanning during plan execution if an unexpected problem occurs. This requires re-entering the planning process at an intermediate point, which GTPyhop's recursive planner cannot do; IPyHOP replaces it with an iterative planner that returns a solution tree, enabling planning to resume from the failure node without re-executing completed tasks. IPyHOPPER (Zaidins, Roberts, and Nau 2023; Zaidins et al. 2025) further extends IPyHOP to repair the solution tree: rather than discarding the tree and replanning, it replans only the smallest affected subtree and forward simulates to detect future failures proactively. Because OOMPA includes symbolic precondition/effect declarations, explicit causal-link analysis is possible—but IPyHOPPER's simulation-based approach is also a natural repair companion for OOMPA domains in an acting context.

PDDL and Unified Planning. PDDL (McDermott, Drew et al. 1998) is the standard language for classical planning. HDDL (Höller et al. 2020) extends it with HTNs. Both require predicate-based representations in a separate domain file. Both are well-supported by competition planners and carry large existing domain libraries. OOMPA complements these with an object-oriented workflow. A planned `as_pddl()` export would allow OOMPA models to compile to PDDL or HDDL, positioning OOMPA as a knowledge engineering layer that lowers the modeling barrier without sacrificing compatibility with standard planners.

Unified Planning (UP) (Andrea Micheli et al. 2025) and OOMPA target different knowledge-engineering challenges. UP is a planner-agnostic Python API for specifying classical, temporal, and hierarchical planning problems; it integrates with more than 35 solvers and supports full PDDL/HDDL/ANML input/output, making it the tool of choice for planning practitioners who need solver portability. OOMPA is an object-oriented modeling toolkit that embeds planning primitives—state variables, actions, and HTN methods— directly into the Python classes a developer would write anyway, using Python's descriptor protocol; state projects to a flat dictionary of dictionaries, keeping planner integration lightweight. Where UP assumes planning knowledge and provides solver flexibility, OOMPA assumes Python knowledge and provides modeling accessibility. The two systems are complementary: a developer could use OOMPA's class-based frontend to lower the domain-authoring barrier, then export to a UP `Problem` to gain access to its broad solver ecosystem—an integration direction that OOMPA's planned `as_pddl()` export would enable.

Functional Representations. Finite-domain representations (FDR) (Helmert 2009) are a cornerstone of recent state-of-the-art planners and heuristics (e.g., Fast Downward), and FDR is often recovered from PDDL models. OOMPA's `StateProperty`, over a finite type, is exactly such a variable. We do not claim equivalence or prove correspondence—multi-valued and propositional encodings are expressively equivalent (Bäckström and Nebel 1995)—but we conjecture that `StateProperty` expresses the same finite-domain state model directly in object-oriented Python.

This pairing of object structure with a functional representation is not unique to OOMPA. ANML (Smith, Frank, and Cushing 2008), the modeling language of the FAPE temporal planner (Bit-Monnot et al. 2020), scopes its time-varying variables inside object types and accesses them by dot-notation, while representing n-ary, non-functional facts as free functions owned by no single object—mirroring OOMPA's domain-level relations. The ASPEN modeling language (AML) (Smith et al. 1998; Fukunaga et al. 1997) likewise declares enumerated single-valued state variables and keeps each action as one self-contained activity block; AML was designed so that mission operators could build models without AI expertise (Sherwood et al. 1998), echoing OOMPA's aim for Python developers. Object-oriented structure and a functional representation are complementary.

These connections suggest a natural export path: because an OOMPA variable is already finite-domain, a compilation to FDR may be a more direct route than the indirect, derived models currently in use. We leave this as future work.

Temporal and Executive Systems. PLEXIL (Estlin et al.

[4] https://bitbucket.org/dananau/pyhop

2006) addresses the plan-execution gap that OOMPA's design targets (but does not yet implement), providing a formal node-based execution semantics designed to transfer plans between planning systems and executives; OOMPA instead keeps execution implicit in the HTN method hierarchy.

EUROPA's (Frank and Jónsson 2003) constraint-based planning and scheduling framework represents plans as attribute-interval mappings governed by declarative configuration rules and compatibilities, with constraint propagation and chronological backtracking search implemented in C++. Its NDDL domain language supports temporal, resource, and concurrency constraints natively—capabilities OOMPA does not yet provide.

MAPGEN (Ai-Chang et al. 2004) is a mixed-initiative planner deployed operationally for rover daily activity planning, combining the EUROPA constraint engine with an interactive editor that enforces constraints in real time during human edits. It supports human planners and encodes domain knowledge as declarative temporal predicates.

T-REX (Teleo-Reactive EXecutive) is a deliberative executive framework for autonomous underwater vehicle (AUV) control developed at MBARI (McGann et al. 2008; Rajan, Py, and Barreiro 2013). It partitions the control problem into concurrent *reactors*, each managing a subset of state variables at a different temporal granularity, and plans continuously using a timeline-based constraint solver. T-REX and OOMPA share a core structural intuition: planning knowledge is encapsulated in modular, object-like components (reactors vs. Python classes) rather than scattered across a flat predicate database. T-REX's reactor hierarchy also parallels OOMPA's `@OompaMethod` hierarchy in that both support hierarchical decomposition of goals into subgoals. The critical difference is orientation: T-REX is engineered for hard real-time AUV deployment with formal temporal semantics and continuous replanning; OOMPA prioritizes developer accessibility and Python-native modeling expressiveness over real-time execution guarantees.

The Reactive Model-based Programming Language (RMPL) and KIRK planning system (Kim, Williams, and Abramson 2001) introduced the idea of compiling hierarchical activity programs into Temporal Plan Networks (TPNs)—extensions of Simple Temporal Networks —that a model-based executive can dispatch. Drake (Conrad and Williams 2011) is an efficient STN-based executive for temporal plans with choice, providing robust dispatching under uncertainty. tBurton (Wang and Williams 2015) extends this line to temporal planning itself, using causal-graph decomposition to plan efficiently over devices with automatic timed transitions. Like OOMPA, RMPL attaches planning knowledge to *device models* rather than flat predicates, and shares OOMPA's ambition to unify the plan representation used for both planning and execution. The key contrast is expressive scope: RMPL compiles programs to temporal constraint networks interpreted by a dedicated model-based executive, while OOMPA's Python class hierarchy is designed to be manipulated directly in Python, making OOMPA's execution model more accessible but currently less temporally expressive.

Related Systems. ActorSim (Roberts et al. 2021; Johnson et al. 2016) is OOMPA's direct predecessor, using a goal-layer network architecture in Java. Prior work with ActorSim demonstrated connections between goal-layer networks and primitive decomposition in hierarchical planning (Macbeth and Roberts 2018). OOMPA preserves ActorSim's design principles while removing the Java and planning-language expertise requirements.

PUG (Langley and Katz 2024) and SOAR (Laird 2012) both organize behavior around goal decomposition, sharing structural similarities with OOMPA's method hierarchy. Kirk and Laird (2019) used SOAR to learn hierarchical game representations with similarities to OOMPA domains.

Timefold uses a similar annotation-based domain modeling approach and targets constraint optimization rather than HTN planning.

8 Conclusion

Modeling a planning domain has usually required two things: domain expertise and planning language expertise. OOMPA attempts to reduce the second requirement. By letting developers annotate standard Python classes with state properties, actions, and hierarchical methods, OOMPA makes it possible to build a planning domain without writing a separate PDDL or HDDL file—at least in principle.

The restaurant planning domain in this paper illustrates what that looks like in practice: a familiar scenario, encoded from scratch in an object-oriented style. The appendices extend the picture with benchmark domains and algorithm listings produced with Claude Code, suggesting OOMPA may be concrete enough to support agentic workflows.

OOMPA 2025.08 is a first cut. Many features are partial or planned, the translation to PDDL remains unproven, and its value relative to writing PDDL or using GTPyhop directly is an open question. If OOMPA's design principles hold up under broader evaluation, they may offer a useful complement to existing planning tools.

Acknowledgments

MR and DC thank the U.S. Naval Research Laboratory for funding this work. Portions of OOMPA and this paper were co-written with the assistance of Claude Code (Sonnet 4.6 and Opus 4.6–4.8). If necessary, the authors can provide chat logs. We also thank the anonymous reviewers, whose feedback, especially in more carefully scoping the contribution and on the discussion of functional representations, greatly improved the paper.

References

Ai-Chang, M.; Bresina, J.; Hsu, J.; Jónsson, A.; Kanefsky, B.; McCurdy, M.; Morris, P.; Rajan, K.; Vera, A.; Yglesias, J.; Charest, L.; and Maldague, P. 2004. MAPGEN: Mixed-Initiative Activity Planning for the Mars Exploration Rover Mission. In *Proc. ICAPS*.

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

Andrea Micheli et al. 2025. Unified Planning: Modeling, manipulating and solving AI planning in Python. *SoftwareX*, 29: 102012.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11(4): 625–655.

Bansod, Y.; Patra, S.; Nau, D.; and Roberts, M. 2022. HTN Replanning from the Middle. In *Proc. FLAIRS*, volume 35.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. *arXiv:2010.13121 [cs]*.

Conrad, P. R.; and Williams, B. C. 2011. Drake: An Efficient Executive for Temporal Plans with Choice. *JAIR*, 42: 607–659.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. UMCP: a Sound and Complete Procedure for Hierarchical Task-network Planning. In *Proc. AIPS*, volume 94, 249–254. Chicago, IL: AAAI Press.

Estlin, T.; Jonsson, A.; Pasareanu, C.; Simmons, R.; Tso, K.; and Verma, V. 2006. Plan Execution Interchange Language (PLEXIL). Technical report.

Fine-Morris, M.; Hsiao, V.; Smith, L. N.; Hiatt, L. M.; and Roberts, M. 2025. Leveraging LLMs for Generating Document-Informed Hierarchical Planning Models: A Proposal. In *AAAI 2025 LLM4Plan Workshop*.

Frank, J.; and Jónsson, A. 2003. Constraint-Based Attribute and Interval Planning. *Constraints*, 8: 339–364.

Fukunaga, A. S.; Rabideau, G.; Chien, S.; and Yan, D. 1997. ASPEN: A Framework for Automated Planning and Scheduling of Spacecraft Control and Operations. In *Proc. i-SAIRAS*.

Ghallab, M.; Nau, D.; and Traverso, P. 2025. *Acting, Planning, and Learning*. Cambridge University Press. Authors preprint at https://projects.laas.fr/planning/.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures. Cham: Springer International Publishing.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173(5-6): 503–535.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Hierarchical Planning Problems. *Proc. AAAI*, 34(06): 9883–9891.

Johnson, B.; Roberts, M.; Apker, T.; and Aha, D. W. 2016. Goal Reasoning with Information Measures. In *Proc. ACS*.

Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *Proc. IJCAI*, 487–493. Morgan Kaufmann Publishers Inc.

Kirk, J. R.; and Laird, J. E. 2019. Learning Hierarchical Symbolic Representations to Support Interactive Task Learning and Knowledge Transfer. In *Proc. IJCAI*, 6095–6102.

Laird, J. 2012. *The Soar Cog. Arch.* Cambridge, MA: MIT Press.

Langley, P.; and Katz, E. P. 2024. A Unified Cognitive Architecture for Embodied Intelligent Agents. In *Proc. ACS*.

Macbeth, J. C.; and Roberts, M. 2018. Exploring Connections Between Primitive Decomposition of Natural Language and Hierarchical Planning. In *Proc. ACS*.

Macbeth, J. C.; Roberts, M.; Zhang, B.; Badhan, S.; Neu, M.; Garg, T.; Zhou, M.; Hua, Y.; and Muco, M. 2025. A View of the Restaurant Script Through the Lens of Hierarchical Planning. In *Proc. ACS*.

McDermott, Drew et al. 1998. PDDL—The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale.

McGann, C.; Py, F.; Rajan, K.; Thomas, H.; Henthorn, R.; and McEwen, R. 2008. A deliberative architecture for AUV control. In *Proc. ICRA*, 1049–1054.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. IJCAI*, 968–973. Stockholm: Morgan Kaufmann Publishers, Inc.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN Planning System. *JAIR*, 20: 379–404.

Nau, D. S.; Roberts, M.; Georgievski, I.; Geib, C. W.; Goldfield, M.; and Zhuo, H. H. 2021. GTPyhop: An HTN/Goal-Task Planner Written in Python. In *ICAPS Hierarchical Planning Workshop*.

Rajan, K.; Py, F.; and Barreiro, J. 2013. Towards Deliberative Control in Marine Robotics. In *Marine Robot Aut.*, 91–175. Springer.

Roberts, M.; Hiatt, L. M.; Shetty, V.; Brumback, B.; Enochs, B.; and Jampathom, P. 2021. Goal Lifecycle Networks For Robotics. In *Proc. FLAIRS*, volume 34.

Schank, R. C.; and Abelson, R. P. 1975. Scripts, Plans, and Knowledge. In *IJCAI*, 151–157.

Schank, R. C.; and Abelson, R. P. 1977. *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures*. Mahwah, NJ: Lawrence Erlbaum Associates.

Sherwood, R.; Govindjee, A.; Yan, D.; Rabideau, G.; Chien, S.; and Fukunaga, A. 1998. Using ASPEN to Automate EO-1 Activity Planning. In *Proc. IEEE Aerospace Conference*.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. S. 2013. The GoDeL Planning System: A More Perfect Union of Domain-Independent and Hierarchical Planning. In *IJCAI*, 2380–2386.

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

Sikes, K.; Fine-Morris, M.; Sreedharan, S.; Smith, L. N.; and Roberts, M. 2025. Creating PDDL Models from JavaScript Using LLMs: Preliminary Results. In *AAAI 2025 LLM4Plan Workshop*.

Smith, B.; Sherwood, R.; Govindjee, A.; Yan, D.; Rabideau, G.; Chien, S.; and Fukunaga, A. 1998. Representing Spacecraft Mission Planning Knowledge in ASPEN. In *K.E. and Acquisition for Planning (AAAI Tech. Report WS-98-03)*, 58–72. AAAI Press.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *Proc. ICAPS KEPS Workshop*.

Tantakoun, M.; Zhu, X.; and Muise, C. 2025. LLMs as Planning Formalizers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. *arXiv preprint arXiv:2503.18971*.

Wang, D.; and Williams, B. C. 2015. tBurton: A Divide and Conquer Temporal Planner. In *Proc. AAAI*.

Yousefi, M.; and Bercher, P. 2024. Laying the Foundations for Solving FOND HTN Problems: Grounding, Search, Heuristics (and Benchmark Problems). In *Proc. IJCAI*, 6796–6804.

Zaidins, P.; Goldman, R. P.; Kuter, U.; Nau, D.; and Roberts, M. 2025. HTN Plan Repair Algorithms Compared: Strengths and Weaknesses of Different Methods. In *Proc. ICAPS*.

Zaidins, P.; Roberts, M.; and Nau, D. 2023. Implicit Dependency Detection for HTN Plan Repair. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

A Condensed OOMPA Restaurant Example

This listing shows the OOMPA implementation of the Restaurant script for the entering and ordering scenes. It includes only the objects and roles necessary for those two scenes, though we have implemented other scenes as well.

A.1 Example Test Harness

```
class TestRestaurant(unittest.TestCase):
    def test_patron_ordering(self):
        domain = RestaurantDomain()
        problem = domain.test_create_simple_problem()

        sam: Server = problem.sam
        menu: Menu = problem.menu_breakfast
        pat: Patron = problem.pat
        chris: Cook = problem.chris
        table1: Table = problem.table1
        result = ApplyResult()

        s_0 = problem.current_state()
        logger.debug(f"s_0:\n{s_0}")

        pat_sit_at_table1 = pat.sit(table1)
        s_1 = s_0.apply(pat_sit_at_table1, result)
        test_and_reset(self, logger, result, s_0, s_1
    , pat.table.equals(table1))

        jon_grabs_menu = pat.pickup_menu()
        s_2 = s_1.apply(jon_grabs_menu, result)
        test_and_reset(self, logger, result, s_1, s_2
    , pat.menu.equals(menu))

        pat_reviews_menu = pat.
    review_menu_for_special()
        s_3 = s_2.apply(pat_reviews_menu, result)
        special_order = result.added[0]
        test_and_reset(self, logger, result, s_2, s_3
    , pat.desired_order.equals(special_order))

        sam_moves_to_pat = sam.move_near(pat)
        s_4 = s_3.apply(sam_moves_to_pat, result)
        test_and_reset(self, logger, result, s_3, s_4
    , sam.near_to.equals(pat))

        pat_orders_special = pat.place_order()
        s_5 = s_4.apply(pat_orders_special, result)
        test_and_reset(self, logger, result, s_4, s_5
    , sam.order.equals(special_order))

        sam_moves_to_chris = sam.move_near(chris)
        s_6 = s_5.apply(sam_moves_to_chris, result)
        test_and_reset(self, logger, result, s_5, s_6
    , sam.near_to.equals(chris))

        sam_tells_chris = sam.convey_order(chris)
        s_7 = s_6.apply(sam_tells_chris, result)
        test_and_reset(self, logger, result, s_6, s_7
    , chris.ordered.equals(special_order))

        logger.debug("Done!")
```

A.2 OOMPA Restaurant Domain

```
# SPF=StatePropertyFactory,
# TOGTN=TotalOrderGoalTaskNetwork,
# OIS=OrderedItem.Status, CNO=CreatesNewObjects,
# GM=GoalMethod, SELP=OIS.SELECTED_BY_PATRON
class RestaurantDomain(AbstractDomain):
    def __init__(self) -> None:
        AbstractDomain.__init__(self, "restaurant")
        self.declare_types(
            Table, MenuItem, Dish,
            Menu, Person, Server, Patron)

    def test_create_simple_problem(self):
        problem = self.instantiate_problem()
        sam = problem.sam = Server("sam")
        menu = problem.menu_breakfast = Menu("
    menu_breakfast")
        special = problem.special = MenuItem("
    veggie_omelette", 12)
        menu.add_special(special)
        problem.pat = Patron("pat")
        problem.chris = Cook("chris")
        table1 = problem.table1 = Table("table1",
    server=sam, menu=menu)
        return problem

class MenuItem(AbstractNamed, HasStateProperties):
    class Status(StrEnum):
        AVAILABLE = auto()
        UNAVAILABLE = auto()
        def __str__(self):
            return self.name
        def __repr__(self):
            return self.__str__()

    cost: int = SPF()
    status: Status = SPF(Status.AVAILABLE)

    def __init__(self, name, cost: int,
                 status: Status = Status.AVAILABLE):
        super().__init__(name)
        self.cost = cost
        self.status = status

class Menu(AbstractNamed, HasStateProperties):
    items: list[MenuItem] = SPF(default_factory=list)
    special: MenuItem | None = SPF(None)

    def __init__(self, name):
        super().__init__(name)

    def add_item(self, item: MenuItem):
        self.items.append(item)

    def add_special(self, special: MenuItem):
        self.special = special

class Table(AbstractNamed, HasStateProperties):
    server: Server = SPF()
    occupied: bool = SPF(False)
    menu: Menu | None = SPF(None)

    def __init__(self, name, server: Server,
                 menu: Menu, occupied=False):
        super().__init__(name)
        self.server = server
        self.menu = menu
        self.occupied = occupied

    def init_add_menu(self, menu: Menu):
        self.menu = menu

    def init_add_patron(self):
        self.occupied = True

class OrderedItem(AbstractNamed, HasStateProperties):
```

```
    class Status(StrEnum):
        UNORDERED = auto()
        SELECTED_BY_PATRON = auto()
        ORDERED = auto()
        PREPARING = auto()
        PREPARED = auto()
        TOTALED = auto()

    PREFIX = "ordered_"

    patron: Patron = SPF()
    item: MenuItem = SPF()
    status: Status = SPF(Status.UNORDERED)
    dish: Dish | None = SPF(None)

    def __init__(self, patron: Patron, item: MenuItem
    ):
        super().__init__(OrderedItem.PREFIX + item.
    name)
        self.patron = patron
        self.item = item


class Person(AbstractNamed): ...

class Server(Person, HasStateProperties):
    near_to: Person | None = SPF(None)
    order: OrderedItem | None = SPF(None)
    menu: Menu | None = SPF(None)
    check: Check | None = SPF(None)

    def __init__(self, name):
        AbstractNamed.__init__(self, name)

    @OompaAction # ==============================
    def move_near(self, person: Person):
        pass

    @move_near.precondition
    def move_near(self, person: Person):
        return AndCondition(
            self.near_to.not_equals(person),
        )

    @move_near.effect
    def move_near(self, person: Person):
        return AndEffect(
            self.near_to.assigned(person),
        )



class Patron(Person, CNO, HasOompaMethods):
    problem: AbstractProblem
    money: int = SPF(50)
    is_hungry: bool = SPF(True)
    is_pleased: bool = SPF(False)
    table: Table | None = SPF(None)
    server: Server | None = SPF(None)
    menu: Menu | None = SPF(None)
    desired_order: OrderedItem | None = SPF(None)

    def __init__(self, name):
        AbstractNamed.__init__(self, name)

    @OompaAction # ==============================
    def sit(self, table: Table):
        pass

    @sit.precondition
    def sit(self, table: Table):
        return self.table.equals(None)

    @sit.effect
    def sit(self, table: Table):
        return AndEffect(
            self.table.assigned(table),
            self.server.assigned(table.server),
            table.occupied.assigned(True),
        )


    @OompaAction # ==============================
    def pickup_menu(self):
        pass

    @pickup_menu.precondition
    def pickup_menu(self):
        return AndCondition(
            self.table.menu.not_equals(None),
            self.menu.equals(None),
        )

    @pickup_menu.effect
    def pickup_menu(self):
        return AndEffect(
            self.menu.assigned(self.table.menu),
        )


class Patron(Person, CNO, HasOompaMethods):
    @OompaAction # ==============================
    def review_menu_for_special(self): pass

    @review_menu_for_special.precondition
    def review_menu_for_special(self):
        return AndCondition(
            self.menu.not_equals(None),
            self.desired_order.equals(None),
        )

    @review_menu_for_special.effect
    def review_menu_for_special(self):
        return AndEffect( InsertNewObjectEffect(
                self,
                self.desired_order,
                OrderedItem,
                [self, self.menu.special],
                self.problem,
                {}, ),
            self.desired_order.status.assigned(SELP),
        )


    @OompaAction # ==============================
    def request_server(self):
        pass

    @request_server.precondition
    def request_server(self):
        return AndCondition(
            self.server.near_to.not_equals(self),
        )

    @request_server.effect
    def request_server(self):
        return AndEffect(
            self.server.near_to.assigned(self),
        )


class Patron(Person, CNO, HasOompaMethods):

    @OompaAction # ==============================
    def place_order(self):
        pass

    @place_order.precondition
    def place_order(self):
        return AndCondition(
            self.desired_order.not_equals(None),
            self.server.near_to.equals(self),
```

```
        )

    @place_order.effect
    def place_order(self):
        return AndEffect(
            self.server.order.assigned(self.
 desired_order),
            self.desired_order.status.assigned(OIS.
 ORDERED),
        )


    @OompaMethod # ==============================
    def m_get_seated(self, tbl: Table) -> GM:
        pass

    @m_get_seated.goal
    def m_get_seated(self, tbl: Table) -> Condition:
        goal = self.table.equals(tbl)
        return goal

    @m_get_seated.precondition
    def m_get_seated(self, tbl: Table) -> Condition:
        return AndCondition(
            tbl.occupied.equals(False),
            self.table.equals(None), )

    @m_get_seated.body
    def m_get_seated(self, tbl: Table) -> TOGTN:
        return TOGTN(
            self.sit(tbl),
        )




class Patron(Person, CNO, HasOompaMethods):

    @OompaMethod # ==============================
    # NOTE: named m_order_special in the OOMPA source
 ; shortened here for the paper
    def m_ordering(self, table: Table) -> GM:
        pass

    @m_ordering.goal
    def m_ordering(self, table: Table) -> Condition:
        return AndCondition(
            self.desired_order.not_equals(None),
            self.desired_order.status.equals(
                OIS.ORDERED))

    @m_ordering.precondition
    def m_ordering(self, table: Table) -> Condition:
        return AndCondition(
            self.desired_order.equals(None),
            table.menu.not_equals(None),
        )

    @m_ordering.body
    def m_ordering(self, table: Table) -> TOGTN:
        body = TOGTN(
            self.sit(table),
            self.pickup_menu(),
            self.review_menu_for_special(),
            self.request_server(),
            self.place_order(),
        )
        return body
```

B OOMPA Travel Example

This example encodes the SimpleTravel domain from GTPyhop. `Person` objects travel between `Location`s either on foot (if close enough) or by taxi. Numeric fare calculation via `Taxi.fare` and the map's `distance` relation illustrate OOMPA's native support for numeric planning.

B.1 Example Test Harness

```
class TestTravelHGN(unittest.TestCase):
    def test_method_travel_by_taxi(self):
        domain = SimpleTravelDomain()
        problem = domain.instantiate_problem()
        domain.test_add_default_locations(problem)
        domain.test_add_alice_and_taxi1(problem)

        alice: Person = problem.alice
        taxi1: Taxi = problem.taxi1
        park: Location = problem.park

        goal = alice.location.equals(park)
        gn1 = TotalOrderGoalTaskNetwork(goal)
        alice_travel_by_taxi = taxi1.travel_by_taxi(
    alice, park)
        result = ApplyResult()

        s_0 = problem.current_state()
        alice_travel_by_taxi.decompose(gn1, s_0,
    result.reset())
        # gn1 now begins with alice.hail(taxi1)
        self.assertEqual(str(gn1.get_unconstrained())
    ,
                         str(alice.hail(taxi1)))

        s_1 = s_0.copy(freeze=False)
        while isinstance(gn1.get_unconstrained(),
    Action):
            gn1.get_unconstrained().apply(s_1, result
    .reset())
            gn1.release()
        self.assertTrue(goal.is_entailed_by(s_1))
```

B.2 OOMPA Simple Travel Domain

```
# SPF=StatePropertyFactory, TOGTN=
    TotalOrderGoalTaskNetwork
class Location(AbstractNamed):
    def __init__(self, name: str):
        AbstractNamed.__init__(self, name)

class Locatable(HasStateProperties):
    map: Map
    location: Location = SPF(LOC_NONE)

class Person(Locatable, HasStateProperties, NamedHash
    ):
    cash: int = SPF(0)
    owe: int = SPF(0)
    taxi: Taxi | None = SPF(None)

    @OompaAction # ==============================
    def walk_to(self, destination: Location) ->
     Action: pass
    @walk_to.precondition
    def walk_to(self, destination: Location) ->
     Condition:
        return self.location.not_equals(destination)
    @walk_to.effect
    def walk_to(self, destination: Location) ->
     Effect:
        return self.location.assigned(destination)

    @OompaAction # ==============================
    def pay_taxi(self) -> Action: pass
    @pay_taxi.precondition
```

```
    def pay_taxi(self) -> Condition:
        return self.owe.greater_than(0)
    @pay_taxi.effect
    def pay_taxi(self) -> Effect:
        return AndEffect(
            self.cash.decreased_by(self.owe),
            self.owe.decreased_by(self.owe),
        )

    @OompaMethod # ==============================
    def travel_by_foot(self, destination: Location) -> GoalMethod: pass
    @travel_by_foot.goal
    def travel_by_foot(self, destination: Location) -> Condition:
        return self.location.equals(destination)
    @travel_by_foot.precondition
    def travel_by_foot(self, destination: Location) -> Condition:
        return Comparison(
            self.map.distance[self.location, destination],
            LESS_THAN_EQUALS, Person.DEFAULT_MAX_WALKING_DISTANCE,
        )
    @travel_by_foot.body
    def travel_by_foot(self, destination: Location) -> TOGTN:
        return TOGTN(self.walk_to(destination))

class Taxi(Locatable, HasStateProperties, NamedHash):
    passenger: Person | None = SPF(None)

    @StaticSPF
    def fare(self, start: Location, end: Location) -> float:
        distance = self.map.distance(start, end)
        return distance * 1.5

    @OompaAction # ==============================
    def transport(self, person: Person, destination: Location) -> Action: pass
    @transport.precondition
    def transport(self, person: Person, destination: Location) -> Condition:
        return AndCondition(
            person.location.not_equals(destination),
            self.location.equals(person.location),
            person.cash.greater_than_equals(
                self.fare[person.location, destination]),
        )
    @transport.effect
    def transport(self, person: Person, destination: Location) -> Effect:
        return AndEffect(
            person.owe.assigned(self.fare[person.location, destination]),
            person.location.assigned(destination),
            self.location.assigned(destination),
        )

    @OompaMethod # ==============================
    def travel_by_taxi(self, passenger: Person, destination: Location) -> GoalMethod: pass
    @travel_by_taxi.goal
    def travel_by_taxi(self, passenger: Person, destination: Location) -> Condition:
        return passenger.location.equals(destination)
    @travel_by_taxi.precondition
    def travel_by_taxi(self, passenger: Person, destination: Location) -> Condition:
        return passenger.cash.gte(self.fare[passenger.location, destination])
    @travel_by_taxi.body
    def travel_by_taxi(self, passenger: Person, destination: Location) -> TOGTN:
        return TOGTN(
            passenger.hail(self),
            self.transport(passenger, destination),
            passenger.pay_taxi(),
        )

class Map(AbstractNamed):
    @StaticSPF
    def distance(self, start: Location, end: Location) -> int | None:
        arg = (start, end)
        return self.edges[arg].value if arg in self.edges else None
```

C OOMPA Depots Example from the Koala Benchmarks

This example encodes a portion of the Depots domain from Koala (Yousefi and Bercher 2024).

C.1 Example Test Harness

```
class TestDepot(unittest.TestCase):
    def test_lift_drop(self):
        domain = DepotDomain()
        problem = domain.test_create_problem_0()
        crate_0 = problem.crate_0
        hoist_1 = problem.hoist_1
        pallet_1 = problem.pallet_1

        result = ApplyResult()

        print(crate_0.__dict__)

        s_0 = problem.current_state()

        a_1 = hoist_1.lift(crate_0)
        s_1 = s_0.apply(a_1, result.reset())
        self.assertEqual(result.status, ApplyResult.Status.SUCCESS)

        a_2 = hoist_1.drop(pallet_1)
        s_2 = s_1.apply(a_2, result.reset())
        self.assertEqual(result.status, ApplyResult.Status.SUCCESS)

        a_3 = hoist_1.lift(crate_0)
        s_3 = s_2.apply(a_3, result.reset())
        self.assertEqual(result.status, ApplyResult.Status.SUCCESS)
```

C.2 OOMPA Depots Domain

```python
# SPF=StatePropertyFactory, AGM=AbstractGoalMethod
# TOGTN=TotalOrderGoalTaskNetwork
class DepotDomain(AbstractDomain):
    def __init__(self) -> None:
        AbstractDomain.__init__(self, "depot")
        self.declare_type(Place);    self.declare_type(Depot)
        self.declare_type(Distributor); self.declare_type(Moveable)
        self.declare_type(Surface);  self.declare_type(Pallet)
        self.declare_type(Truck);    self.declare_type(Crate)
        self.declare_type(Hoist)

    def test_create_problem_0(self): ...   # sets up depots, hoists,
    def test_add_depot(self, ...): ...     # pallets, trucks, crates
    def test_add_distributor(self, ...): ...
    def test_add_truck(self, ...): ...
    def test_add_hoist(self, ...): ...
    def test_add_pallet(self, ...): ...
    def test_add_crate(self, ...): ...

class Place(AbstractNamed):
    def __init__(self, name: str):
        AbstractNamed.__init__(self, name)

class Depot(Place):
    def __init__(self, name): Place.__init__(self, name)

class Distributor(Place):
    def __init__(self, name): Place.__init__(self, name)

class Locatable(HasStateProperties):
    at: Place = SPF()
    def __init__(self, at: Place): self.at = at

class Surface(Locatable, HasStateProperties):
    top: Crate | None = SPF(None)
    height: int = SPF(0)
    def __init__(self, top: Crate | None = None):
        self.top = top
    def init_add_crate(self, crate: Crate):
        self.top = crate; crate.on = self

class Located(Locatable, AbstractNamed, HasStateProperties):
    def __init__(self, name, at: Place):
        AbstractNamed.__init__(self, name)
        Locatable.__init__(self, at); self.at.freeze()

class Pallet(Located, Surface):
    """Holds containers, stacked up to three high."""
    def __init__(self, name: str, at: Place, top: Crate = None):
        Located.__init__(self, name, at); Surface.__init__(self, top)

class Hoist(Located, HasStateProperties):
    holding: Crate | None = SPF(None)
    MAX_STACK_HEIGHT: int = SPF(3)
    def __init__(self, name: str, at: Place, holding: Crate = None):
        Located.__init__(self, name, at)
        if holding is not None: self.init_add_crate(holding)
    def init_add_crate(self, crate: Crate):
        self.holding = crate; crate.on = self

    @OompaAction # ==============================
    def lift(self, crate: Crate):
        """Lifts a Crate off of its current Surface."""
        pass
    @lift.precondition
    def lift(self, crate: Crate):
        return AndCondition(
            crate.at.equals(self.at),
            self.holding.equals(None),
            crate.top.equals(None),
        )
    @lift.effect
    def lift(self, crate: Crate):
        return AndEffect(
            self.holding.assigned(crate),
            ConditionalEffect(TypeMatches(crate.on, Surface),
                              crate.on.top.assigned(None)),
            crate.on.assigned(self),
            crate.height.assigned(0),
        )

    @OompaAction # ==============================
    def drop(self, surface: Surface):
        """Drops the held Crate on the surface."""
        pass
    @drop.precondition
    def drop(self, surface: Surface):
        return AndCondition(
            self.at.equals(surface.at),
            self.holding.not_equals(None),
            surface.top.equals(None),
            surface.height.less_than(self.MAX_STACK_HEIGHT),
        )
    @drop.effect
    def drop(self, surface: Surface):
        return AndEffect(
            surface.top.assigned(self.holding),
            self.holding.at.assigned(surface.at),
            self.holding.top.assigned(None),
            self.holding.height.assigned(surface.height),
            self.holding.height.increased_by(1),
            self.holding.assigned(None),
        )

class Moveable(Locatable, AbstractNamed, HasStateProperties):
    def __init__(self, name: str, at: Place):
        AbstractNamed.__init__(self, name); self.at = at

class Crate(Moveable, Surface, HasStateProperties):
    """Can be on a surface, crane, or truck; has its own surface."""
    on: Surface | Hoist | Truck | None = SPF(None)
    def __init__(self, name, at, placed_on=None, top=None):
        Moveable.__init__(self, name, at)
        Surface.__init__(self, top=top)
        if placed_on is not None: placed_on.init_add_crate(self)
        if top is not None: self.init_add_crate(top)

class Truck(Moveable, HasStateProperties):
    """Trucks are Moveable and can carry one Crate."""
    crate: Crate | None = SPF(None)
    def __init__(self, name: str, at: Place):
        Moveable.__init__(self, name, at)
    def init_add_crate(self, crate: Crate):
        self.crate = crate; crate.on = self

    @OompaAction # ==============================
```

```
    def drive(self, start: Place, end: Place): pass
    @drive.precondition
    def drive(self, start: Place, end: Place):
        return AndCondition(
            Comparison(start, NOT_EQUALS, end),
            self.place.equals(start),
        )
    @drive.effect
    def drive(self, end: Place):
        return self.place.assigned(end)

class GetCrate(AGM[DepotDomain, "GetCrate"]):
    crate: Crate;  surface: Surface
    @property
    def goal(self):
        return self.domain.is_on(self.crate).equals(self.surface)
    @property
    def precondition(self):
        is_at = self.domain.is_at
        return is_at(self.crate).not_equals(is_at(self.surface))
    @property
    def body(self):
        is_in = self.domain.is_in; is_at = self.domain.is_at
        is_on = self.domain.is_on; is_clear = self.domain.is_clear
        return TOGTN(
            is_clear(self.crate).equals(True),
            is_in(self.crate).not_equals(None),
            is_at(is_in(self.crate)).equals(is_at(self.surface)),
            is_clear(self.surface).equals(True),
            is_on(self.crate).equals(self.surface),
        )
    def __call__(self, crate, surface): ...

class PutOn(AGM[DepotDomain, "PutOn"]):
    crate: Crate;  surface: Surface
    @property
    def goal(self):
        return self.domain.is_on(self.crate).equals(self.surface)
    @property
    def precondition(self):
        if self.crate == self.surface: return NullCondition()
        return self.domain.is_at(self.crate).equals(
            self.domain.is_at(self.surface))
    @property
    def body(self):
        is_on = self.domain.is_on; is_clear = self.domain.is_clear
        holding = self.domain.holding
        return TOGTN(
            is_clear(self.crate).equals(True),
            is_clear(self.surface).equals(True),
            holding(self.hoist).equals(self.crate),
            is_on(self.crate).equals(self.surface),
        )
    def __call__(self, crate, surface): ...

class LoadTruck(AGM[DepotDomain, "LoadTruck"]):
    crate: Crate;  truck: Truck;  hoist: Hoist
    @property
    def goal(self):
        return self.domain.is_in(self.crate).equals(self.truck)
    @property
    def precondition(self):
        is_in = self.domain.is_in; is_at = self.domain.is_at
        return AndCondition(
            is_in(self.crate).equals(None),
            is_at(self.crate).equals(is_at(self.hoist
    )),
        )
    @property
    def body(self):
        is_at = self.domain.is_at; is_clear = self.domain.is_clear
        is_in = self.domain.is_in; holding = self.domain.holding
        return TOGTN(
            is_at(self.truck).equals(is_at(self.hoist
    )),
            is_clear(self.crate).equals(True),
            holding(self.hoist).equals(self.crate),
            is_in(self.crate).equals(self.truck),
        )
    def __call__(self, crate, truck, hoist): ...

class UnloadTruck(AGM[DepotDomain, "UnloadTruck"]):
    crate: Crate;  surface: Surface;  hoist: Hoist
    @property
    def goal(self):
        return self.domain.is_on(self.crate).equals(self.surface)
    @property
    def precondition(self):
        is_in = self.domain.is_in; is_at = self.domain.is_at
        return AndCondition(
            is_in(self.crate).not_equals(None),
            is_at(is_in(self.crate)).equals(is_at(self.hoist)),
            is_at(self.hoist).equals(is_at(self.surface)),
        )
    @property
    def body(self):
        is_clear = self.domain.is_clear; is_on = self.domain.is_on
        holding = self.domain.holding
        return TOGTN(
            is_clear(self.surface).equals(True),
            holding(self.hoist).equals(self.crate),
            is_on(self.crate).equals(self.surface),
        )
    def __call__(self, crate, surface, hoist): ...
```

D Example Algorithms

D.1 Goal Decomposition Planner

The following is a simplified, deterministic implementation of the Goal Decomposition Planner (Shivashankar et al. 2012) built from OOMPA primitives using Claude Code (Claude Sonnet 4.6). It selects the first applicable method at each step (no backtracking), omits the cycle detection present in the full algorithm, and returns the resulting state rather than OOMPA's richer `ApplyResult`.

```
OK = ApplyResult.Status.SUCCESS
def gdp(problem, goal, state):
    gtn = TotalOrderGoalTaskNetwork(goal)
    while not goal.is_entailed_by(state):
        node = gtn.get_unconstrained()
        if isinstance(node, Action):
            result = ApplyResult()
            node.apply(state, result)
            if result.status != OK:
                return None  # action failed
            gtn.release()
        else:  # node is a Goal
            methods = problem.applicable_methods(
                gtn, state)
            if not methods:
                return None  # no method
            method = methods[0]  # first applicable
            result = ApplyResult()
            method.decompose(gtn, state, result)
    return state
```

D.2 A* Search

The following is a standard A* search implemented using OOMPA primitives and Claude Code (Claude Sonnet 4.6). Frozen (hashable) state copies enable the visited set; a caller-supplied heuristic defaults to zero, giving uniform-cost search, with each action costing one unit since OOMPA does not yet expose an `action.cost`.

```
import heapq
from itertools import count

OK = ApplyResult.Status.SUCCESS
def astar(p, goal, state, h=lambda s: 0):
    start = state.copy(freeze=True)
    tie = count()
    # heap: (f, tie, g, state, plan)
    h0 = h(start)
    frontier = [(h0, next(tie), 0, start, [])]
    visited = {start}
    while frontier:
        f, _, g, cur, plan = heapq.heappop(frontier)
        if goal.is_entailed_by(cur):
            return plan
        for action in p.applicable_actions(cur):
            result = ApplyResult()
            ns = cur.copy(freeze=False)
            action.apply(ns, result)
            if result.status != OK:
                continue
            nf = ns.copy(freeze=True)
            if nf in visited:
                continue
            visited.add(nf)
            g_new = g + 1  # uniform cost
            heapq.heappush(frontier,
                (g_new + h(nf),
                 next(tie), g_new,
                 nf, plan + [action]))
    return None  # no plan found
```