Hierarchical Task Network Planning in Common Lisp: the case of SHOP3

Robert P. Goldman and Ugur Kuter
rpgoldman@sift.net
ukuter@sift.net
SIFT, LLC
Minneapolis, MN

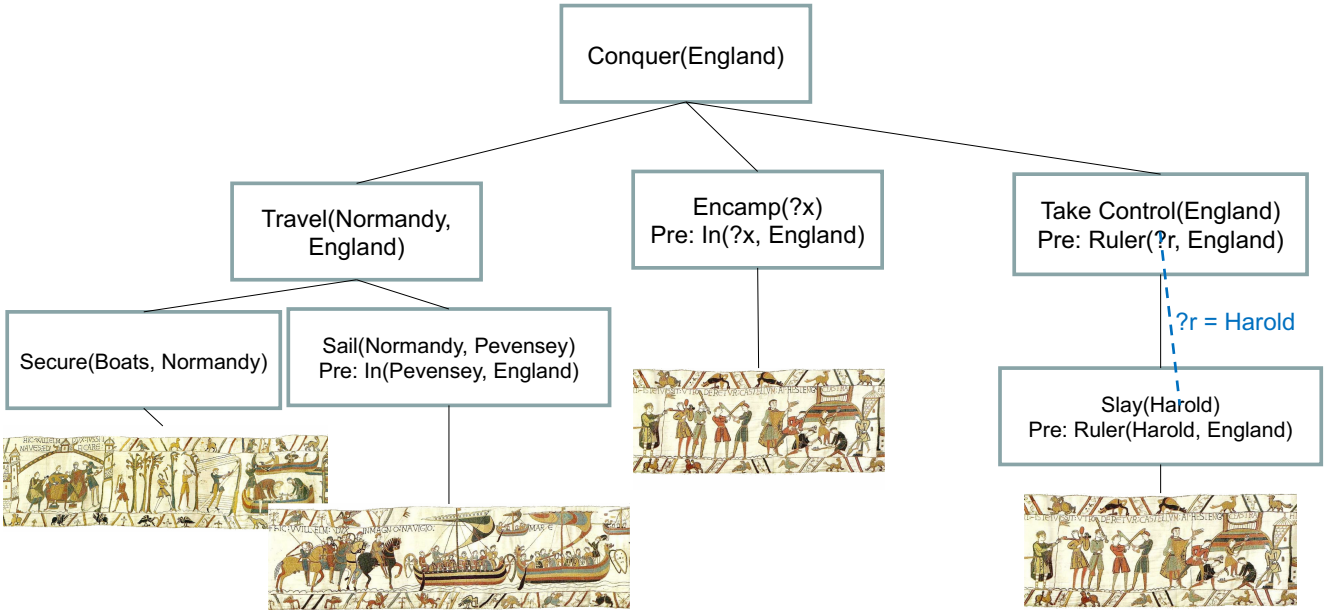

Figure 1: Hierarchical Plan for Conquest of England [11]

ABSTRACT

This paper describes the use of Common Lisp (CL) to develop a new version of the Hierarchical Task Network (HTN) planner, SHOP2, first developed at the University of Maryland (UMD). which we are dubbing SHOP3. We will describe ways in which we have profited from language features offered by CL to build a more solid, efficient, yet flexible planning system, review lessons learned and suggest some best practices. SHOP3 is an open source tool made publicly available by SIFT, hosted on GitHub. It is freely available for use under the terms of the Mozilla Public License. CL provided a good foundation for extensibility and refactoring of the SHOP2 planner to support both more flexibility and extensibility and, at the same time, more usability as a practical tool. By comparison, the Java version of SHOP2 was rigid, and rapidly abandoned after the original developer left UMD. By then it already lagged behind the CL version in terms of features because of Java's rigidity, and poor support for symbolic programming.

ELS '19, April 01–02, 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2633324

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Planning and scheduling**; **Planning for deterministic actions**; **Planning with abstraction and generalization**; *Logic programming and answer set programming*; • **Information systems** → *Decision support systems.*

KEYWORDS

Common Lisp, HTN planning, AI Planning, symbolic reasoning, software abstraction, software modeling, software flexibility

ACM Reference Format:
Robert P. Goldman and Ugur Kuter. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *Proceedings of ELS '19: European Lisp Symposium (ELS '19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2633324

1 INTRODUCTION

AI planning is the subfield of artificial intelligence (AI) that aims at automating processes of *means-ends reasoning*. In general, AI planning is the problem of finding a sequence of actions that, executed in a specified initial state, will reach a goal state. This is a problem with applications to diverse areas including manufacturing, autonomous space and deep sea exploration, medical treatment, and military operations, to name just a few.

We describe our ongoing work, over the past two decades, on the SHOP3 planning system, a Hierarchical Task Network (HTN) planner. SHOP3 is based on SHOP2, which was originally developed at the University of Maryland (UMD). In our work we have made extensive use of Common Lisp (CL) features to extend and harden the SHOP2 code base over the years, culminating in the release this month of its successor, SHOP3.

Our company, SIFT, (|www.sift.net|) is a for-profit research lab, employing approximately 35 people, with 13 Ph.D.s. We are in our 20th year, and have offices in Minneapolis, and the Boston, Washington, and San Diego metropolitan areas. We do contract research on AI, Computer Security, Formal Methods, and Human-Computer Interaction, primarily for the US Federal Government. Funding sources include DARPA, NASA, Department of Energy, Air Force Research Laboratories, Office of Naval Reserch, etc.

CL provided a good foundation for extensibility and refactoring of the SHOP2 planner and, at the same time, more usability as a practical tool. By comparison, the Java version of SHOP2 was rigid, and rapidly abandoned after the original developer left UMD. By then it already lagged behind the CL version in terms of features because of Java's rigidity, and poor support for symbolic programming.

We describe ways in which we have profited from language features offered by Common Lisp to build a more solid, efficient, yet flexible planning system, review lessons learned and suggest some best practices. SHOP2, and in turn, SHOP3 are based on a powerful Prolog-style logic programming capability for both logical inference and planning. We will discuss how well-suited CL is for this kind of symbolic computation in an AI planning system.

SHOP3 is available and regression-tested in several CL platforms, including Allegro CL (https://franz.com/products/allegro-common-lisp/), SBCL (www.sbcl.org), and CCL (https://ccl.clozure.com/). SHOP3 is an open source library that is publicly available through SIFT. The final version of SHOP2 is currently available for download from Sourceforge. SHOP3 is available from GitHub, through the "shop-planner" group: |https://github.com/shop-planner/|. It is available for use according to the Mozilla Public License.

We begin the paper with some background material, first a description of the problem of AI planning, and more specifically, the approach of HTN planning. As part of our discussion of HTN planning, we will review the original SHOP2 system, and its history. We will describe a number of challenges we faced in applying SHOP2 to problem domains, making it extensible in fundamental ways, and also more efficient. We will describe how we have addressed these problems and specifically how features of Common Lisp have aided us. We will conclude with some best practices for use of Common Lisp for this kind of symbolic computing, and some holes we would like to see filled to improve the utility of the language.

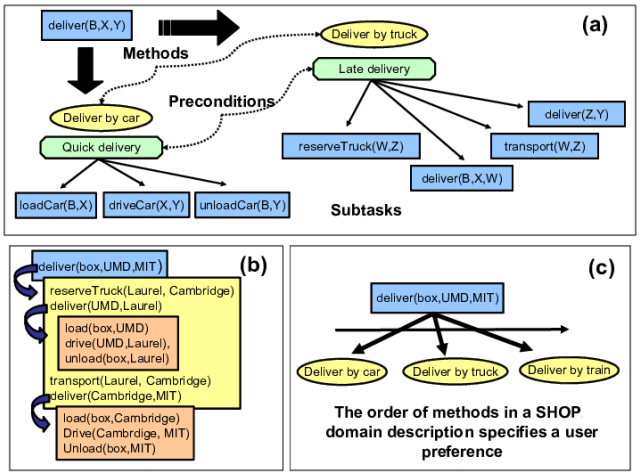

Figure 2: Delivery planning example.

2 PRELIMINARIES: AUTOMATED PLANNING

We provide a brief introduction to AI planning, largely following the discussion of Ghallab, Nau, and Traverso [9]. Formally, a "classical" planning problem involves a *state-transition system*, $\Sigma =< S, A, \gamma >$ where S is a set of states, A is a set of actions, and γ is a state transition function, $\gamma : S \times A \rightarrow S$. A *planning problem* is a triple $P =< s_0, \Sigma, G >$ where s_0 is the initial state, Σ is the state-transition system, and $G \subseteq S$ is the *goal*. A *plan*, or solution, is a sequence of actions $\pi = a_0 ... a_n$ such that $\gamma(\gamma(\gamma(s_0, a_0), a_1)..., a_n) \in G$. In classical planning, the state space is factored into a set of propositions, and every state $s \in S$ is a complete truth assignment to the set of propositions. For example, in a planning problem with a robot, `r`, that moves between three workstations, `w1`, `w2` and `w3`, a state would be a truth assignment to the three propositions `(at r w1)`, `(at r w2)`, and `(at r w3)`. In this case, the three are mutually exclusive, so that there are only three states.

Actions are triples, $a = \langle \text{name}(a), \text{prec}(a), \text{eff}(a) \rangle$ where the name is an arbitrary designator, the *preconditions*, prec(a) specify conditions under which the action can be executed (making γ a partial function), and the *effects*, eff(a) are a factored representation of the transition function. For example, the action `(move w1 w2)` would have the preconditions `(at r w1)` and the effects `(and (not (at r w1)) (at r w2))`. In the interests of convenience, one describes actions using action *schemas* (macros), for example:

```
(:action (move ?r - robot ?w0 ?w1 - workstation)
   :precondition (at ?r ?w0)
   :effect (and (not (at ?r ?w0)) (at ?r ?w1)))
```

The standard language for describing planning *domains* (a set of action descriptions and ancillary information) and problems is the Planner Domain Definition Language (PDDL)[5, 8, 17]. PDDL was developed to facilitate the International Planning Competition (IPC) [12, 17], held in conjunction with the International Conference on Automated Planning and Scheduling (ICAPS). The `move` action definition above is written in PDDL. There are a wide selection of publicly available benchmark problems that have been used in past

IPCs. PDDL has multiple sub-languages of increasing expressive power. We will return to this issue later.

At its core, planning is a graph search problem, and as such might seem suitable for methods such as Dijkstra's algorithm, no worse than $O(n^2)$. However, for single source shortest path (SSP), n is the size of the graph. For planning the input is extremely compressed, and the state space is exponential in the size of the input, so conventional SSP algorithms are, in practice, exponential. Indeed, the planning problem is very hard: intractable even when the expressive power (domain and problem) are tightly restricted. See, for example, Bäckström, *et al.* [1]. Another way of thinking about AI planning is that it involves synthesizing an open loop controller for goal reachability.

The IPC has spurred a great deal of advancement in classical planning since its inception. Planners can now solve very large classical planning problems, and there are a diversity of different planning methods. The most successful approaches have been based on heuristic search, reduction to propositional satisfiability (SAT), and local search/constraint satisfaction. However, we cannot overemphasize the expressive limitations of these classical planners, which makes them unusable for most practical applications. In the next sections, we will describe hierarchical task network (HTN) planning, sometimes also referred to as "decomposition planning," which has radically more expressive power.

3 HIERARCHICAL TASK NETWORK PLANNING

Hierarchical Task Network (HTN) planning addresses many of the problems of classical, "first principles" planning as described above. Classical planning, for example, is limited to goals of *achievement*. For example, jogging around a track cannot easily be captured as goal achievement, because the goal state is to end up in the starting position. Most classical planners cannot effectively plan for multiple agents, because they only optimize for either plan length or cost minimality for additive, context independent action costs. First principles planners also find plans unconstrained by considerations such as standard operating procedures. Related to this, a first principles planner requires a causal theory (preconditions and effects), whereas an HTN planner can do actions "just because": e.g., part of the protocol for treating stroke victims involves giving aspirin, although we do not have a clear causal theory of its effectiveness.

All of these concerns can be addressed by HTN planning, and the University of Maryland's Shop2 planner was the most mature and complete HTN implementation, so when SIFT was looking for a planner for applications, that is where we started. Unlike a first principles planner, an HTN planner produces a sequence of actions that perform some activity or *task*, instead of finding a path to a goal state. An HTN planning domain includes a set of planning *operators* (actions) and *methods*, each of which is a prescription for how to decompose a task into its *subtasks* (smaller tasks). The description of a planning problem contains an initial state as in classical planning. Instead of a goal formula, however, there is a partially-ordered set of tasks to accomplish.

Planning proceeds by decomposing tasks recursively into subtasks, until *primitive tasks*, which can be performed directly using the planning operators, are reached. For each task, the planner chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks or the interactions among them prevent the plan from being feasible, the planner will backtrack and try other methods.

Shop2 is an HTN planner that generates actions in the order they will be executed in the world. Its backtracking search considers the methods applicable to the same task in the order they are specified in the knowledge base given to the planner. This feature of the planner allows for specifying *user preferences* among such methods, and therefore, among the solutions that can be generated using those methods. For example, Figure 2(c) shows a possible user preference among the three methods for the task of delivering a box from the University of Maryland (UMD) to MIT.

Consider a Delivery Domain, in which the task is to deliver a box from one location to another. Figure 2(a) shows two Shop2 methods for this task: *delivering by car*, and *delivering by truck*. Delivering by car involves the subtasks of loading the box to the car, driving the car to the destination location, and unloading the box at the destination. Note that each method's preconditions are used to determine whether or not the method is applicable: thus in Figure 2(a), the *deliver by car* method is only applicable if the delivery is to be a fast one, and the *deliver by truck* method is only applicable if it is to be a slow one. Now, consider the task of delivering a box from the UMD to MIT and suppose we do not care about a fast delivery. Then, the *deliver by car* method is not applicable, and we choose the *deliver by truck* method. As shown in Figure 2(b), this decomposes the task into the following subtasks: *(1)* reserve a truck from the delivery center at Laurel, Maryland to the center at Cambridge, Massachusetts, *(2)* deliver the box from the University of Maryland to Laurel, *(3)* drive the truck from Laurel to Cambridge, and *(4)* deliver the box from Cambridge to MIT. For the two delivery subtasks produced by this decomposition, we must again consider our delivery methods for further decomposing them until we do not have any other task to decompose.

During planning, the planner evaluates the preconditions of the operators and methods with respect to the world state it maintains locally. It is assumed that planner has all the required information in its local state in order to evaluate these preconditions. For example, in the delivery example, it is assumed that the planner knows all the distances between the any initial and final locations so that it can determine how long a truck will be reserved for a delivery task.

4 SIMPLE HIERARCHICAL ORDERED PLANNER (SHOP2)

The original algorithm for Shop2 is based on recursive depth-first search, and has as its key data structures (1) a stack of open tasks, (2) a state object that supports mutation and rollback, (3) a plan list and, (4) a set of variable bindings. A pseudocode version of the algorithm is given in Algorithms 1, 2 and 3. These are taken originally from the Shop2 Manual, but the control flow has been simplified, and the management of variable bindings, which was suppressed in the original, has been included.

Some notes on the algorithm: the **choose** operator in the pseudocode represents nondeterministic choice, implemented as search.

Algorithm 1 Planning algorithm

```
1: procedure PLAN(S, t)                                   ▷ state, task
2:     return FIND PLANS(S, {t}, ∅)
3: end procedure
```

In practice, SHOP2 uses depth-first search[1] so, for example, in Algorithm 3, line 8, **choose** $b \in B$, what is done is to attempt to proceed with b bound to the first element of B, and if this fails, to try again with the next element of B until a solution is found or B is exhausted, at which time this line of code fails. In practice there will be something like an `OR` in the code, and there will be a non-tail recursive call, leaving a new frame on the stack. As one would expect, difficult search problems – or even degenerate problems involving no search, but requiring long plans, cause stack exhaustion.

Note also that this involves being able to roll back state changes that come from operator application when the system backtracks to an earlier point in the search and consider different alternative plans. This backtracking is not done on the stack: instead the state objects are built out of an original state and set of incremental updates, which are labeled. When the system backtracks, it undoes changes (by removing updates from the state object), using the labels in the update sequence.

The **query()** function in the following is all-solutions Prolog-style database retrieval (backward chaining). For those familiar with Prolog, this is similar to a **bagof** query. The return is a list of binding sets. Each binding set associates some set of variables with values (which may be other variables), by unification. The **apply()** function returns a new expression that results from applying a set of variable bindings to an original expression.

We can see from this that the key operations are (1) **tree search**, (2) **operator application**, (3) **task reduction**, (4) **retrieval** from the state database, and (5) **unification**. All of these operations involve symbolic computation.

4.1 Issues with SHOP2

SIFT has been working with SHOP for approximately 15 years now; we chose it because it was the only open source HTN planner available, it was relatively efficient and well-tested, and it performed well in the 2002 IPC – the last IPC in which HTN planners competed [16]. SHOP2 also has been used in a number of planning applications, including recently at SIFT for Air Operations and UAV planning [14, 15, 18, 19], cyber security [3], cyber-physical systems [10], planning for synthetic biology experiments [26], and software vulnerability analysis [13], to name a few. For an earlier survey of SHOP2 applications, see Nau, *et al.* [21]. Another advantage of SHOP is that it provides easy call-out to special purpose solvers through an ability to invoke arbitrary Lisp code. For example, we used this to invoke code in a navigation library that could generate route plans, compute distances on the globe, and retrieve ground elevation information from a GIS to plan safe routes.

Our initial steps working with SHOP2 involved modernizing the code base for use in larger systems. After the modernization, we built a number of large systems incorporating our version of SHOP,

[1] Although variants, including depth-first iterative deepening and branch and bound search are also available.

Algorithm 2 Simplified planning search algorithm

```
1: procedure FIND PLANS(S, T, B)                          ▷ state, tasklist, bindings
2:     if T = ∅ then
3:         return ()                          ▷ No tasks: return empty action sequence.
4:     end if
5:     choose t ∈ T with no predecessors
6:     if t is primitive then
7:         o ← operator for t
8:         if o is applicable in S then
9:             S′ ← result(o, S)
10:            T′ ← T − t
11:            P ← FIND PLANS(S′, T′, B)
12:            return cons(o, P)
13:        else
14:            return FAIL
15:        end if
16:    else                                                ▷ t is a complex task
17:        < b, R′ >= reduction(t, S)
18:        if b is FAIL then
19:            return FAIL
20:        else
21:            B′ = apply(b, B)
22:            if B′ is FAIL then
23:                ▷ Merge new bindings with incoming.
24:                return FAIL
25:            end if                ▷ Replace t with its expansion R′ in T
26:            T′ ← replace(t, R′, T)
27:            return FIND PLANS(S, T′, B′)
28:        end if
29:    end if
30: end procedure
```

Algorithm 3 Task reduction procedure

```
1: procedure REDUCTION(t, S)                              ▷ task, State
2:     choose m a method for name(t)
3:     ▷ List of bindings from precondition query.
4:     b* = query(pre(m), S)
5:     if b* = ∅ then
6:         return FAIL
7:     else
8:         choose b ∈ b*                          ▷ bindings from preconditions
9:         R ← task-net(m)
10:        R′ ← apply(b, R)
11:        return b, R'
12:    end if
13: end procedure
```

and became increasingly aware of issues in programming and using it. This led us to add features to make it easier to program (develop new domains and programs) *correctly*, and to debug. We rearchitected the SHOP2 to SHOP3 for two reasons: to make it easier to incorporate/reuse individual components of SHOP in external systems and to support extending and adapting SHOP's planning model, by incorporating new input languages, inference methods, etc. This rearchitecting has also enabled us to incorporate an entirely new search engine into SHOP3, in a way that enables us to better fit search methods to application domains.

5 SHOP3 ARCHITECTURE

Like much University research software, SHOP2 was originally distributed as a single file, not at all suitable to incorporation in a larger system and also available in many hard-to-track variants – approximately one per graduate student! So for our first application using SHOP2, we did a thorough modernization. The first step was namespacing: creating a SHOP package and identifying SHOP2's API. Another software engineering chore was to add an ASDF system definition and decompose the source code into multiple files organized topically, and by dependency. We originally moved it to Sourceforge's subversion server for revision control. Finally, as we used SHOP more and more, we accumulated a large set of regression tests for the system.

The original monolithic SHOP2 system contained at least three different subsystems that were of general usefulness, and that should be usable separately. See Figure 3 for a diagram of the SHOP3 system architecture. Two low-level subsystems that could be loaded independently were the `shop3/unifier` and `shop3/common` (state) systems. The unifier, as the name suggests, provides an implementation of the unification algorithm [24] over s-expressions, with supporting data structures for binding, etc. This can be used as a library independent of the rest of SHOP. Similarly, `shop3/common` provides an implementation of a logical database supporting change over time. It contains state data structures (of different forms, offering different tradeoffs in cost between update and retrieval), with update and undo operations.

The `shop3/theorem-prover` provides a Prolog-like logic programming framework on top of the state data structures of `shop3/common` and the unification algorithm of `shop3/unifier`. Like conventional Prolog, the SHOP theorem-prover provides Horn clause deduction, but unlike it, the theorem-prover allows the programmer to reason about a state that changes over time, forming a state *trajectory*. Note that the theorem-prover does *not* support temporal logic, but it does allow the programmer to provide Prolog-style reasoning as a state evolves through changes. Temporal logic would be an interesting extension, but it remains to be seen how well it would interact with the changeset-style representation SHOP uses. The `shop3/theorem-prover-api` provides a programmer-friendly general `query` interface to the theorem-prover, an addition to the more primitive API used by the planner itself.

On top of the theorem-prover and state representation, SHOP3 contains a sub-library of "planning operations" that cover the core operations of primitive task insertion and task reduction we see in Algorithms 2 and 3. This layer was separated out in order to support the development of the more general Explicit Stack Search engine described below in Section 5.2. These operations include adding an operator/action to the plan sequence, while updating the state; and replacing a complex task with the task network from a compatible method.

At the top of the pyramid is the SHOP3 system itself, marrying search with planning operations, theorem-proving, and state update. The system also provides assorted utilities for visualization, etc., including a plan grapher library that can draw a plan derivation tree using CL-DOT [25] and `graphviz` [4, 6]. This was a door opened to us by the modernization of the system, and the interoperability and supply of libraries enabled by the common adoption of ASDF.

The original SHOP2 system had its own idiosyncratic syntax, which evolved more than being designed. This led to parsing code that developed into a hard-to-maintain ball of special cases. Also, the requirement to use that syntax made comparisons with other planners more difficult, and generally closed off access to the large number of publicly-available planning models and problems that have come out of the IPC. Accordingly, we added support for the PDDL language, as we describe below. A substantial part of this involved making the input of domains and problems generic and dispatching on domain type (see below). The theorem-prover's behavior was also made generic, which allows SHOP3 to support the different notation for logic used by PDDL and SHOP2, and their different scoping disciplines.

5.1 Object model: Domains

As described earlier, a planning *domain* is a repository of action models, and in the case of SHOP, also method definitions, and axioms. Domains capture everything that is common among a set of related planning problems. For this reason, when we were designing SHOP3 for greater extensibility, subclassing the `DOMAIN` object class was the obvious step to take. Figure 4 gives a simplified diagram of the type lattice around the `DOMAIN` class.

Domains enable customization of domain and problem notation through generic functions for parsing, theorem-proving, and search. Figure 4 gives a sense of how this is done. At the root of the hierarchy is the theorem-prover domain. This class provides the theorem-proving behaviors through methods on `REAL-SEEK-SATISFIERS-FOR` and `LOGICAL-KEYWORDP` that dispatch on the domain and a logical keyword. Definition of such methods is aided by a `DEF-LOGICAL-KEYWORD` macro.

The standard SHOP3 `DOMAIN` combines the theorem-prover domain with a mixin that stores action definitions to enable state progression. Examples of customization can be seen in the immediate subclasses of `DOMAIN`, including `TEMPORAL-DOMAIN`, which adds the ability to have operators with durations, and the `LOOPING-DOMAIN`, which adds looping constructs to the SHOP3 domain language.

More interesting are the cluster of classes that support PDDL. The PDDL language has a `:requirements` construct that allows features of the language to be enabled separately. The basic language supports only simple action schemas with conjunctive preconditions and effects, and untyped variables (implicitly universally quantified). Various requirement keywords enable the addition of more complex constructs to the language. For each of these we have provided a corresponding `-MIXIN` class (see the left side of Figure 4). Note the addition of the ADL (Action Description Language) [22] class, which assembles a bundle of logical operators, quantifiers, and conditional effects.

5.2 Explicit Stack Search (ESS)

The core of AI planning is *search*; the search for a path of states from the initial state to a state that satisfies the goal. For a number of reasons, we would like to experiment with different search methods, but the original SHOP2 implemented its search process using the Lisp process stack. This makes it difficult, if not impossible, to control the search. In general, while using the processor stack as the search stack provides a convenient and rapid way to implement

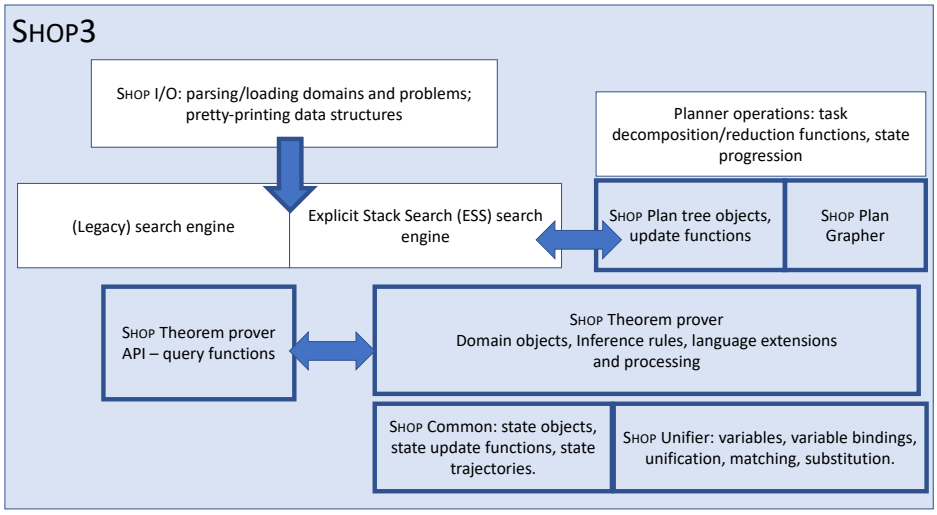

Figure 3: High level SHOP3 system architecture. Shaded blocks indicate components directly accessible by programmers. Dark outlines indicate components that can be loaded stand-alone.

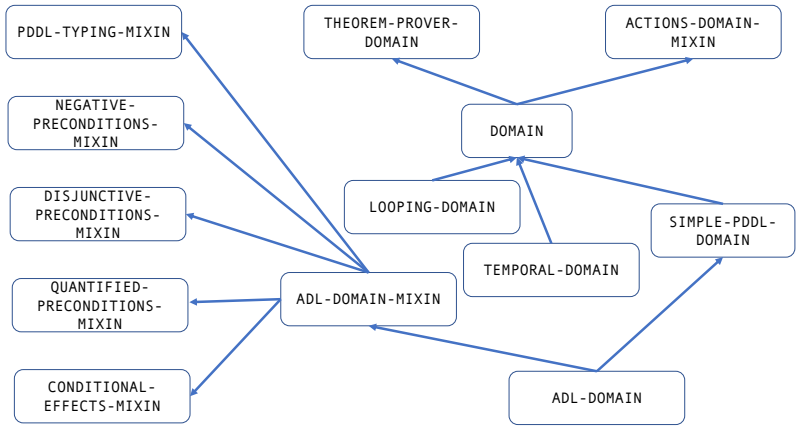

Figure 4: Object model: domains.

search algorithms, tackling complex search problems requires finer control than this implementation strategy provides.

To address this issue, we have introduced Explicit Stack Search (ESS) as an alternative to the standard SHOP2 search algorithm. The first step in doing this was to reify the search engine as a CLOS object, so that we could use generic function method dispatch to tailor individual behaviors. The next step was to introduce the "planning operations" layer (see Figure 3) to tease apart the planning operations from the recursive function invocations of the stock SHOP2 depth-first search strategy.

To implement ESS, we adopted a technique from the CIRCA planner [20], and constructed an abstract finite state machine. Each state in the virtual machine will carry out some computation, update virtual machine data structures, and then jump to a new state. This allows easy incorporation of new behaviors. The current version of SHOP3 supplies standard depth-first search, mirroring the original SHOP2 search implementation, and also *backjumping* [7], which we are using in recent work on plan repair.

The core data structures for ESS are a search state object, and the backtrack stack. The search state groups together all of the information in the state of the search as described in Algorithms 1 through 3 – the current task, the plan so far, the cost, etc. – in addition to the state of the virtual machine – the mode, unexplored alternatives to the current decision, etc. The backtrack stack is made up of two kinds of object. The simplest is a marker, which indicates a choice point, and is pushed onto the stack whenever there are unexplored alternatives. The other is one of a set of objects that are pushed to record the effects of a decision. These objects hold data: e.g., when a new choice point is reached, ESS must push the unexplored alternatives from the last choice point. For each class

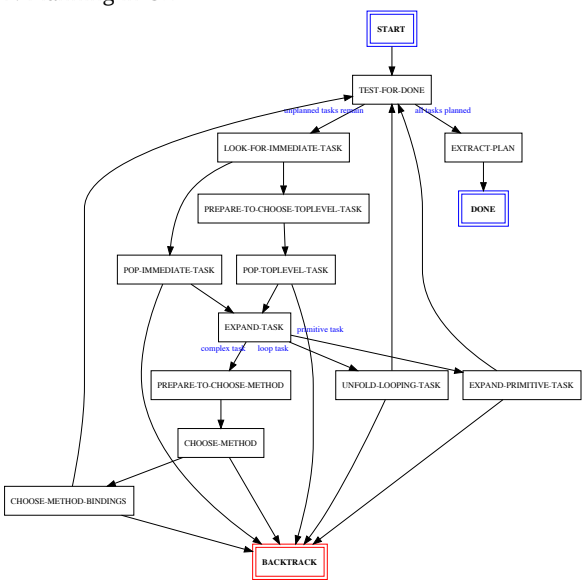

Figure 5: State machine for ESS.

of these objects, there must be an undo method provided, that can restore the state of the search machine, and of the search itself.

6 SHOP3 IMPLEMENTATION

In this section we discuss a number of miscellaneous topics concerning the implementation of SHOP3, and features provided by and to the community. First, we have worked to clean up the syntax of the languages for the planner. We have also migrated away from using lists as the uniform data structure. Finally, we developed an extensive set of tests and to support it, created a library that integrates the FiveAM testing library with the ASDF build system.

The original logical language of SHOP2 was idiosyncratic in syntax and in the set of capabilities provided. In SHOP3 we provide standard features of Prolog, including all-solutions meta-predicates, etc. We also add more debugging features, including the ability for the programmer to explicitly raise runtime exceptions, singleton variable checks (for misspellings), and anonymous variables (to support singleton variable checks). We are gradually adding more load-time checks to domain definitions, although this is complicated by SHOP3's support for meta-programming.

Debugging and efficiency have both been improved by extensive introduction of special-purpose data structures in place of SHOP2's pervasive use of lists for all purposes. The pervasive use of lists made it quite difficult, for example, to debug the theorem-prover, which did not make a clear distinction between binding lists (lists of variable bindings, representing a single solution) and lists of binding lists (representing multiple solutions, in each element of which variables would be bound to different values). The pervasive use of lists also made SHOP2 less efficient. Use of structures for inner-loop operations (e.g. variable bindings in the theorem-prover) has provided substantial speed-ups, as well as code that is easier to understand and maintain, because of fewer quiet failures.

As part of the SHOP3 development effort, we developed the FiveAM-ASDF library. As its name suggests, this provides integration between the FiveAM Lisp testing library [2], and the ASDF build system [23]. It enables the programmer to designate a set of tests to be run as the `TEST-OP` of an ASDF system, and also provides conditions for test failures, etc. These conditions are necessary because the execution of ASDF operations does not provide a useful return value. This way the system can be tested either interactively, or one can encapsulate the test operation in a trivial bash script that will provide a non-zero error code in case of failures, thus making it suitable for use in a continuous integration framework. Because we found that errors in test definition could lead to silent failures, the system also provides the programmer the ability to specify the number of checks that are expected and FiveAM-ASDF will raise a (different) condition if an unexpected number of checks are run. As an aside, we are starting to decompose the regression tests into multiple sub-systems, as the library of tests has become so extensive, and testing is trivially parallelizable. The test suite takes approximately 45 minutes to run for each CL implementation.

7 CONCLUSIONS AND LESSONS LEARNED

We have described our use of CL and its language features to support extensive symbolic computations in the HTN planning systems SHOP3, and SHOP2. CL provided a good foundation for extensibility and refactoring of these systems, and more usability as practical tools.

One of the advantages of using CL for an AI planning system was the use of s-expressions as the universal data structure in the software. Our experience has been that this can be very convenient for initial prototyping, particularly because they provide more convenient inspection in the debugger, but they should be phased out rapidly for improving the scalability and modularity in the code. The foresight of the CL standardizers in providing `list`-type structures has been an immeasurable help in this process. Another substantial advantage was s-expressions as a convenient data structure for symbolic computing. Unless one has built symbolic computing systems in both CL and a more conventional language like Java or Python, it may be difficult to appreciate how great a help this is.

Going forward, the addition of rigorous gradual typing to CL would be very helpful. SBCL provides excellent type inference, but while it provides very valuable information about correctness, this is a byproduct of a concern for optimization, and the CL type system was not designed as a tool for program correctness.

Many of the things we have seen in our code and our predecessors' in SHOP2 may be common informal knowledge, but it is worth enumerating them so that they can enter the CL community's *explicit* knowledge. Some examples include:

(1) *Ad hoc* development of domain-specific languages
(2) `&allow-other-keys` (and often the `&key` of CLOS) is an anti-pattern: the maintainer or library user trying to determine what arguments are supported will have to traverse class inheritance hierarchies (especially for `initialize-instance`), or method dispatch hierarchies, wasting time, creating confusion, and often leading to errors only caught at run-time. More specifically, if we have base class C, and we need to extend it to another class C', we realized that one should never do just that. Instead, factor out commonalities into a $C0$, and make C and C' both be children of $C0$. If we do not, sooner or later we found that

we would need to add a behavior to C that should not go on C', and then it would be a painful refactoring.

(3) Cascading `initialize-instance` methods also create confusion and cause issues with maintenance.

A foremost best practice that underlies many of the others is to *assume* that the code will break and ask yourself how it can be debugged, and especially how it can be debugged by others. A common mistake by the over-confident programmer is to provide a complex, but elegant structure that resists debugging because of not asking this question.

ACKNOWLEDGMENTS

We would like to thank Prof. Dana Nau and his students and postdocs at the University of Maryland for the original SHOP2. John Maraist for many extensions during his time at SIFT. Rick Freedman, Dan Bryce, and other SIFT employees for the logo. The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under Contract Number FA8750-17-C-0184. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the Department of Defense, or the United States Government.

REFERENCES

[1] Christer Bäckström, Yue Chen, Peter Jonsson, Sebastian Ordyniak, and Stefan Szeider. 2012. The Complexity of Planning Revisited-A Parameterized Analysis.. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI Press, Toronto, Ontario.

[2] Edward Marco Baringer and Stelian Ionescu. [n. d.]. FiveAM. https://common-lisp.net/project/fiveam/

[3] Mark Burstein, Robert Goldman, Paul Robertson, Robert Laddaga, Robert Balzer, Neil Goldman, Christopher Geib, Ugur Kuter, David McDonald, John Maraist, Peter Keller, and David Wile. 2012. STRATUS: Strategic and Tactical Resiliency Against Threats to Ubiquitous Systems. In *Proceedings of SASO-12*.

[4] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2003. Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. In *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 127–148.

[5] Maria Fox and Derek Long. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20, 1 (Dec. 2003), 61–124. http://dl.acm.org/citation.cfm?id=1622452.1622454

[6] Emden R. Gansner and Stephen C. North. 2000. An Open Graph Visualization System and Its Applications to Software Engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30, 11 (2000), 1203–1233.

[7] John Gaschnig. 1979. *Performance Measurement and Analysis of Certain Search Algorithms*. Technical Report CMU-CS-79-124. Carnegie-Mellon University.

[8] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Dan Weld, and David Wilkins. 1998. *PDDL – The Planning Domain Definition Language*. Technical Report CVC TR-98-003. Yale Center for Computational Vision and Control, New Haven, CT.

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann, San Francisco, CA.

[10] Robert P. Goldman, Daniel Bryce, Michael J. S. Pelican, David J. Musliner, and Kyungmin Bae. 2016. A Hybrid Architecture for Correct-by-Construction Hybrid Planning and Control. In *NASA Formal Methods*, Sanjai Rayadurgam and Oksana Tkachuk (Eds.). Vol. 9690. Springer International Publishing, Cham, 388–394. https://doi.org/10.1007/978-3-319-40648-0_29

[11] Ulrich Harm. 2018. Bayeux Tapestry TItuli. http://www.hs-augsburg.de/~harsch/Chronologia/Lspost11/Bayeux/bay_tama.html

[12] Jörg Hoffmann, Stefan Edelkamp, Sylvie Thiébaux, Roman Englert, Frederico dos S. Liporace, and Sebastian Trüg. 2006. Engineering Benchmarks for Planning: The Domains Used in the Deterministic Part of IPC-4. *Journal of Artificial Intelligence Research* 26 (2006), 453–541. http://dx.doi.org/10.1613/jair.1982

[13] U. Kuter, M. Burstein, J. Benton, D. Bryce, J. Thayer, and S. McCoy. 2015. HACKAR: Helpful Advice for Code Knowledge and Attack Resilience. In *AAAI/IAAI Proceedings*.

[14] U. Kuter, R. P. Goldman, and J. Hamell. 2018. Assumption-based Decentralized HTN Planning. In *Proceedings of the ICAPS-18 Workshop on Hierarchical Planning*.

[15] U. Kuter, B. Kettler, J. Guo, M. Hofmann, V. Champagne, K. Lachevet, J. Lautenschlager, L. Asencios, J. Hamell, and R. P. Goldman. 2019. Profiles, Proxies, and Assumptions: Decentralized, Communications-Resilient Planning, Allocation, and Scheduling. In *Proceedings of the AAAI/IAAI-19*.

[16] Derek Long and Maria Fox. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20 (2003), 1–59.

[17] Drew V. McDermott. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21, 2 (2000), 35–55.

[18] Joseph B Mueller, Christopher A Miller, Ugur Kuter, Jeff Rye, and Josh Hamell. 2017. A Human-System Interface with Contingency Planning for Collaborative Operations of Unmanned Aerial Vehicles. In *AIAA Information Systems-AIAA Infotech@ Aerospace (2017-1296)*. AIAA Press. https://doi.org/10.2514/6.2017-1296

[19] David Musliner, Robert P. Goldman, Josh Hamell, and Chris Miller. 2011. Priority-Based Playbook Tasking for Unmanned System Teams. In *Proceedings AIAA*. American Institute of Aeronautics and Astronautics.

[20] D. J. Musliner, E. H. Durfee, and K. G. Shin. 1993. CIRCA: A Cooperative Intelligent Real-Time Control Architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23, 6 (Nov. 1993), 1561–1574. https://doi.org/10.1109/21.257754

[21] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, H. Muñoz-Avila, J. W. Murdock, D. Wu, and F. Yaman. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20, 2 (March–April 2005), 34—41.

[22] E.P.D. Pednault. 1989. ADL: Exploring the Middle Ground between Strips and the Situation Calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Ron Brachman, Hector J. Levesque, and Raymond Reiter (Eds.). Morgan Kaufmann Publishers.

[23] Francois-René Rideau and Robert P. Goldman. 2010. Evolving ASDF: More Cooperation, Less Coordination. In *International Lisp Conference*. ACM Press. https://doi.org/10.1145/1869643.1869648

[24] J. A. Robinson. 1971. Computational Logic: The Unification Computation. *Machine Intelligence* 6 (1971).

[25] Juho Snellman. [n. d.]. CL-DOT – Generate Dot Output from Arbitrary Lisp Data. http://www.foldr.org/~michaelw/projects/cl-dot/

[26] U.Kuter, R. P. Goldman, D. Bryce, J. Beal, M. DeHaven, C. Geib, A. F. Plotnick, N. Roehner, and T. Nguyen. 2018. XPLAN: Experiment Planning for Synthetic Biology. In *Proceedings of the ICAPS-18 Workshop on Hierarchical Planning*.