

Improving AI Planning by Using Extensible Components

by

Michael Jonas

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2016 by the
Graduate Supervisory Committee:

Ashraf Gaffar, Chair
Adam Doupe
Cormac Herley
Georgios Fainekos

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

Despite incremental improvements over decades, academic planning solutions see relatively little use in many industrial domains despite the relevance of planning paradigms to those problems. This work observes four shortfalls of existing academic solutions which contribute to this lack of adoption.

To address these shortfalls this work defines model-independent semantics for planning and introduces an extensible planning library. This library is shown to produce feasible results on an existing benchmark domain, overcome the usual modeling limitations of traditional planners, and accommodate domain-dependent knowledge about the problem structure within the planning process.

TABLE OF CONTENTS

	Page
LIST OF TABLES	iv
LIST OF FIGURES	v
CHAPTER	
1 INTRODUCTION	1
Introduction	1
Project Motivation	10
Background Literature	16
2 DATA CENTER DOMAIN	19
Data Center Thermal Model	20
Planning Considerations	25
PDDL Implementation	30
Evaluation	32
3 PDDL SEMANTIC LIMITATIONS	33
4 JPDL TRANSLATOR	37
Language Independent Semantics	39
World Model Advancement Algorithm	41
Translation	43
Conclusions	64
5 MODELING LIBRARY SPECIFICATION	68
Language Independent Semantics	70
Modeling Data Structures	75
Conclusions	84
6 PLANNING LIBRARY SPECIFICATION	85
Language Independent Semantics	87
Planning Branch Algorithm	92
Planning Data Structures	93

CHAPTER	Page
Conclusions	104
7 BLOCKS WORLD EXAMPLE DOMAIN	107
Planning Library Model	108
Blind Heuristic.....	119
Satisficing Heuristic	123
Evaluation	125
REFERENCES	131
APPENDIX	
A TABLE OF ICAPS SURVEY	136
B DATA CENTER PDDL DOMAIN FILE	138
C DATA CENTER PDDL FACT FILE	145
D BLOCKS WORLD PDDL DOMAIN FILE	150

LIST OF TABLES

	Page
TABLE	
1. Data Center Thermal Model Symbols.....	23
2. JPDL Semantic Symbols.....	39
3. Java Syntax Support Summary.....	43
4. Modeling Library Symbols.....	70
5. Planning Library Symbols.....	87
6. Overstack Problem Solution.....	126
7. Swapstack Initial Problem Mismatches.....	127
8. Swapstack Initial Problem Clear Height Dictionary.....	127
9. Swapstack Problem Solution.....	128
10. Heuristic Comparison on IPC Optimal Track Problems.....	129
11. Satisficing Heuristic vs. Track 1 IPC Results.....	130
12. Satisficing Heuristic vs. Track 2 IPC Results.....	130

LIST OF FIGURES

	Page
FIGURE	
1. Design Stack of Declarative Systems	3
2. Overview of CPS Interaction.	10
3. Overview of Semantic Mapping between System and Components.	11
4. Design Stack of CPS and Planner Component.	12
5. Survey Result of Application Papers in ICAPS Conference.	13
6. AC Coefficient of Performance.	20
7. Effects of Thermal-Aware Scheduling.	22
8. Discretization Method for Implementing Thermal Model in PDDL.	26
9. PDDL Translation Method of For Loops.	56
10. Modeling Library UML Overview.	75
11. Shared Pointer Structure between State and Effects.	79
12. Memory Waste Caused by Defensive Cloning.	80
13. Alternative Partial Cloning Method.	82
14. Planning Library UML Overview.	93
15. DecisionEpochPattern UML Diagram.	93
16. Checkpoint UML Diagram.	94
17. CheckpointLink UML Diagram.	94
18. LinkTree UML Diagram.	95
19. LinkTree Conceptual Diagram.	96
20. LinkTree Structural Diagram.	97
21. World Lines Visualization.	98
22. PlanningBranch UML Diagram.	98
23. PlanningHeuristic UML Diagram.	99
24. PlanningController UML Diagram.	100
25. BlocksWorld StateVariable UML Diagram.	108

CHAPTER	Page
26. BlocksWorld State UML Diagram.....	109
27. BlocksWorld Effect UML Diagram.....	111
28. Example BlocksWorld Goal.....	112
29. BlocksWorld Goal UML Diagram.	113
30. BlocksWorld Mismatch UML Diagram.	115
31. BlocksWorld Overstack Problem.....	116
32. BlindHeuristic UML Diagram.	119
33. BlocksWorld BFS Problem.	120
34. BFS Problem Search Tree.	121
35. Overstack Problem Revisited.....	125
36. BlocksWorld Swapstack Problem.	127

CHAPTER 1

INTRODUCTION

Procedural code is primarily concerned with specifying an algorithm detailing *how* to do something. Declarative code is instead primarily concerned with specifying facts about a system and *what* is desired while omitting implementation details. For a declarative language to function an algorithm which powers it must understand what changes are possible and be linked to a means of actuation. The nature of this task varies from algorithm to algorithm, from domain to domain. Declarative approaches are intended to make a solution easier to develop by removing the burden of specifying ‘the *how*’ and instead limiting the burden of the developer to specifying ‘the *what*’. A simple example of this comes from the WiX toolset for making installers.

```
<Directory Id='Foo' Name='Foo'>
```

The syntax here uses XML which is generally understood by most developers. This line roughly translates to: “After the installer runs there will be a directory named Foo”. By contrast, a procedural definition roughly translates to “Create a directory named foo”.

```
Directory.CreateDirectory(path+"Foo");
```

There are several important differences here.

- 1) For the former definition to have an effect there must be an algorithm that recognizes the concept of a directory, recognizes or detects that a directory needs creating using logic, and maps this need to a procedural definition of how to create a directory. The algorithm’s domain is the set of concepts the algorithm contains logic and mappings for. Declarative code cannot natively execute on a processor. A processor has no concept of high-level languages; it only recognizes a limited set of low-level, immediate, instructions. All declarative code eventually links to a procedural definition in order for it to have an effect.
- 2) Consider the case where the directory could not be created because it already exists. Refer to the rough translations above and try to predict what the system will look like when the declarative and procedural implementations complete. In the declarative case, with respect to this domain, there is only one sensible interpretation for a response. In the

procedural case, the result of the statement is less predictable. The result may vary in implementation between Java, C#, or Python.

- 3) The burden of responsibility for handling the minutia of possibilities is entirely in the hands of the developer in the procedural case whereas in the declarative case the algorithm may provide a commonly agreed-upon solution.

These observations begin to demonstrate the value of declarative approaches as well as some fundamental requirements of them. If common understanding exists about domain concepts and logic the developer can be alleviated of the burden of minutia of a procedural approach. However, this requires the declarative system and its user to agree upon a set of semantics. This body of work is concerned with the nature of the algorithms which power declarative approaches, the fundamental limits and requirements of such algorithms, and the available alternatives which often go unconsidered.

To elaborate, we need to take a closer look at the fundamentals of a declarative approach. A declarative system is usually a component of or framework used by a larger system as shown in Figure 1.

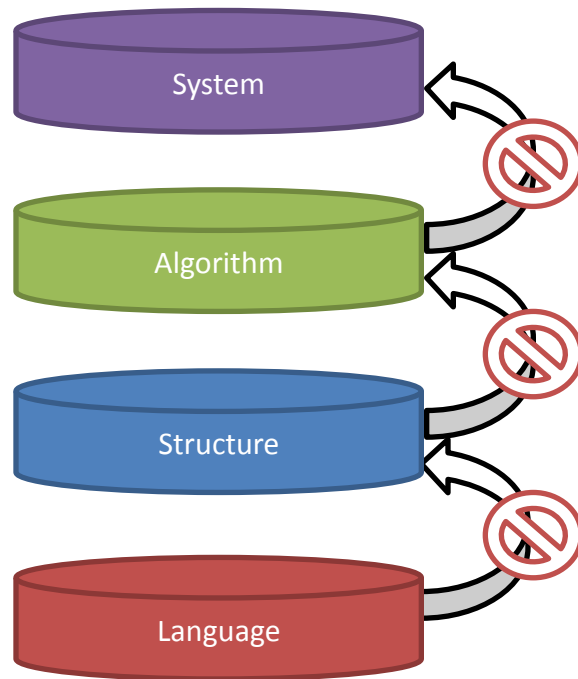


Figure 1: Design Stack of Declarative Systems.

The language representation is the start of the interaction with the system from a user perspective. This could be SQL, Prolog, ADL, WiX, etc. This representation is parsed or interpreted into a structure which is acted upon by an algorithm which follows it. This structure could take many forms, and can often vary from implementation to implementation. For example, MySQL uses a different implementation from MSSQL and hence generate a different structure despite mostly sharing the same language.

The algorithm encodes at least two things. First, it includes logic which recognizes certain concepts and their specific expected behavior and breaks them down into specific, proper responses. Second it contains a mapping of these responses to procedural code, directly or indirectly.

For the directory example above an algorithm could simply create the nested directories from the WiX description in a recursive way. This algorithm encodes two things:

- 1) The knowledge of the middleware structure and the significance of its information with respect to the purpose of the algorithm
- 2) A mapping to a procedural actuation to fulfill that purpose.

A reductive example suitable for discussion is shown below:

```
public class WiXDirectory
{
    public string Name;
    public string Path;

    public List<WiXDirectory> Children;
}
public static void CreateDirectories(WiXDirectory directory)
{
    Directory.CreateDirectory(directory.Path + directory.Name);
    foreach (WiXDirectory child in directory.Children)
    {
        CreateDirectories(child);
    }
}
```

A more realistic solution will include significantly more complex logic such as error handling. WiXDirectory here, or a list of them, is the middleware structure corresponding to the declarative language representation. A parser would build this structure from the WiX representation and at runtime a controller algorithm will invoke CreateDirectories as part of the installation process. If this structure were to change, the algorithm which uses it will fail to function properly. Procedural code needs to address something to invoke it and the structures which are being addressed need to be agreed upon before an algorithm can act on them. Regardless of what the structure is an agreed upon structure is required. This structure is a necessary constraint of the system which uses it. Incorrect behavior can result from an algorithm which uses a structure while having an incorrect or incomplete understanding of it. On the other hand if the purpose of the algorithm requires information which the structure is insufficient to express the algorithm cannot fulfill its purpose.

These structures are not merely convention or a reflection of syntax of the language; the structures used by the algorithm are efficient/convenient for the purpose of that algorithm. They organize information for the algorithm and define with precision the forms that information can

take. The meaning of this information is encoded by an author into an algorithm by logically consistent usage with respect to a purpose.

The algorithm fulfills the purpose of an installer by recognizing the implicit assumption that the WiXDirectory objects correspond to the need to create a directory and by coordinating the creation of the directory. The recognition of a concept, the knowledge of its significance, and the knowledge of what to do about it are generally what we refer to as the semantics of the algorithm.

In addition to semantics an algorithm encodes a mapping to a procedural implementation. In the example above the algorithm provides this by invoking CreateDirectory which actually does the work. Its signature is below and can be linked to a procedural definition which if invoked correctly fulfills the algorithms purpose.

```
public static DirectoryInfo CreateDirectory(string path)
```

The understanding of the procedural definition is encoded into the algorithm by intelligent and consistent usage and in no way is the procedural definition interpreted or inferred by some method of artificial intelligence. A programmer encodes his knowledge of the procedural definition in sufficient detail that his knowledge and intelligence is no longer required at runtime. Artificial intelligence has yet to approach replacing the role of a programmer or designer and it is dubious whether such a thing can be solved using an extension of any known approach. Even if the source code is available it is famously known according the Halting Problem that extracting complete knowledge of a function from source code is impossible. If we cannot extract working knowledge about code from its source the only alternative is by its documentation in the way that a programmer does. However, this method requires a thorough knowledge of and intelligent model of the world that the code is intended to model in order to understand the code and how to apply it in a way consistent with that world.

Even having defined the role of language, structure, and algorithm this is still only part of the system. These systems are intended to provide useful general functionality which is then applied to an actual domain by a domain author who is the user of the declarative system. The domain author which is using the system has a different set of semantics. A domain expert brings

with him a ubiquitous language related to the domain with a distinct set of concepts, structures, relationships, and intuitions about how his domain behaves. In order to use the declarative system the domain author has to map the domain semantics to the algorithms semantics.

Semantics are in some ways like namespaces. Different namespaces may contain the same identifiers to refer to the same real world concept but reason with and structure that concept in completely different ways. The difference between namespaces is similar to the difference between domain and declarative system semantics. These semantics are not always compatible and even when they are the mapping itself is often awkward, clumsy, or suffers from some type of resolution loss or conceptual reduction/generalization. Relationships between entities can be lost, continuous values can be discretized, organization structures which are efficient for that domain can be generalized, etc. Depending on the complexity of the domain and how cleanly a domain maps to these assumptions of the declarative system these constraints can be extremely problematic. However, some form of restriction is necessary and inherent in defining the structure. The algorithm is code, and code has to refer to a mutually understood structure to correctly address it and begin to act on it. There are generally two ways that algorithms achieve these guarantees.

- 1) They restrict the language and structure entirely to a range and combination of values which the algorithm is proven to work for.
- 2) They make assumptions about the behavior of any form of dependency injection and build the algorithm around those assumptions.

We demonstrate each of these cases next before considering alternatives. Below is an example of a closed system.

```

private enum KnownCaseEnum { a, b }

private int Algorithm(KnownCaseEnum caseEnum)
{
    if (caseEnum == KnownCaseEnum.a)
        return ProceduralMethodForA();
    else if (caseEnum == KnownCaseEnum.b)
        return ProceduralMethodForB();
    else throw new Exception();
}

```

The algorithm can be proven to work for the restricted set of cases that the enum allows. In the WiX directory example the BNF of the WiX language restricts the structure that directories can be specified in. For example, a developer cannot specify one directory to reside within two different parent directories. Neither can a directory be specified as a child element of a file. Neither case will compile. When interpreted, the language is restricted to the extent that it never produces a case for directories where the full behavior is not understood. There is no hook for dependency injection anywhere as a child of the directory specification where a procedural mapping can cause unknown behavior and in that sense the system is closed.

The following code demonstrates how to allow dependency injection.

```

private interface MyStruct
{
    int LeafMethod();
}

private int Algorithm(MyStruct myStruct)
{
    int behaviorSwitch = myStruct.LeafMethod();
    return ProceduralMethod(behaviorSwitch);
}

```

By defining itself in this way the algorithm cedes control to an object passed by the invoker. If the *myStruct* reference can be provided by the domain author then the definition of *ProceduralMethod* must either define behavior for every possible integer of its parameter or risk failure. The algorithm treats each point of dependency injection as a black box leaf and assumes it will be well-behaved. This is an inherent risk which can violate any guarantees the algorithm hopes to provide. For example, if the implementation of *LeafMethod* is unstable and crashes then *ProceduralMethod* will never even be invoked.

One need not look far in our WiX example to find an example of this. Within the first several pages of every WiX tutorial Custom Actions are introduced. CustomActions are roughly just domain injection to arbitrary methods during the installation process with extra attributes for specifying timing.

```
<CustomAction Id='Foo' BinaryKey='FooBinary' DllEntry='FooEntryPoint'/>
<Binary Id='FooBinary' SourceFile='foo.dll'/>
<InstallExecuteSequence>
  <CustomAction='FooAction' After='InstallFiles'/>
</InstallExecuteSequence>
```

In the example here, after files are installed, the installer will call the FooEntryPoint method on the foo.dll file. The installer has no idea what this custom action will do, but it treats it as a black box leaf and assumes that it will not destructively interfere with the installation process by, for example, tempering with files or directories which are needed.

In neither of the cases above can the domain author augment or extend the implementation of ProceduralMethod within Algorithm with additional structure and mappings. Either the algorithm and its structure are sufficiently complete on delivery for their purpose or they aren't. In practice, declarative systems often provide numerous pieces of functionality. If most of that functionality is still desirable but the domain semantics cannot be mapped for even a single piece of the system the entire process can fail. This raises the question, how can an algorithm use a structure beyond its understanding? Either the information the algorithm needs must be provided by an alternate means or the functionality of the algorithm must be extensible. We can demonstrate both in a single example.

```
private Dictionary<int, Delegate> BehaviorMappings;
```

```
private Object Algorithm(int behaviorSwitch, Delegate domainAuthorDelegate)
{
  if (domainAuthorDelegate != null)
  {
    return domainAuthorDelegate.DynamicInvoke(new object[0]);
  }
  else
  {
    object[] args = new object[1] { behaviorSwitch };
    return BehaviorMappings[behaviorSwitch].DynamicInvoke(args);
  }
}
```

Replacing the CaseEnum logic from the first example with an extensible mapping structure allows extra behaviors that fit within a known structure but are unhandled by an algorithm to be added to the system as needed by the domain author. The domainAuthorDelegate parameter allows the functionality to be replaced entirely if the structure is insufficiently expressive for some reason. This example may seem reductively simple but when demonstrated in the context of more complex systems this change can dramatically improve the usability of the system as a whole. In short techniques such as these enable a domain author to extend the algorithm semantics in those cases where they would otherwise be insufficient for the domain.

This work dives deep into the complex area of planning in artificial intelligence which is historically dominated by declarative/code interpretive approaches. There we observe a limitation of existing systems which raises the question just discussed. How can an algorithm use a structure beyond its understanding? We show several examples of how domain semantics can exceed the traditional assumptions of this field and propose an alternative system which overcomes these limitations.

PROJECT MOTIVATION

Cyber-physical systems (CPSs) are unique in that they contain both a cyber and a physical component. These systems not only exist in the physical world but affect the world as part of their core functionality by means of actuation. The world in turn affects the computation of the cyber domain via sensing. Between this sensing and actuation the cyber component of the CPS makes sense of the world and via some algorithmic process decides on a course of action which fulfills the system's purpose. This general approach has been used by cognitive architectures since their foundation [1-3].

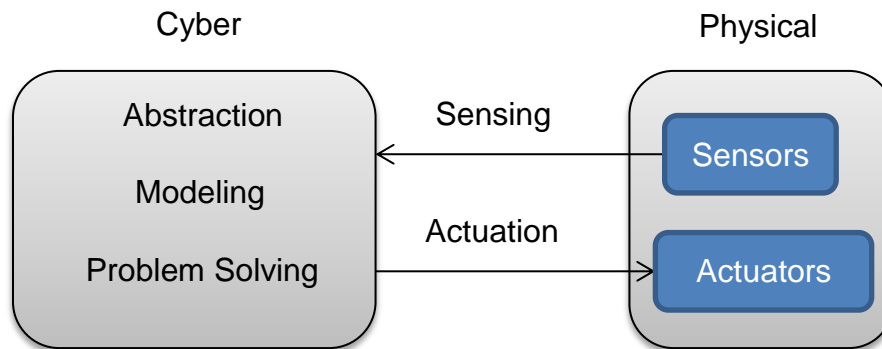


Figure 2: Overview of CPS Interaction.

The purpose and nature of the problem each cyber-physical system attempts to solve will be different, resulting in different solutions. However, many of the solution subtasks are already established in different fields. Many CPSs include foundational components such as statistical libraries, planning libraries, and model checking tools [4-7]. These components are used because they are well-established and CPS owners hope to lower development costs. However, before a component can be used the semantics of the domain must be mapped to the semantics of the component.

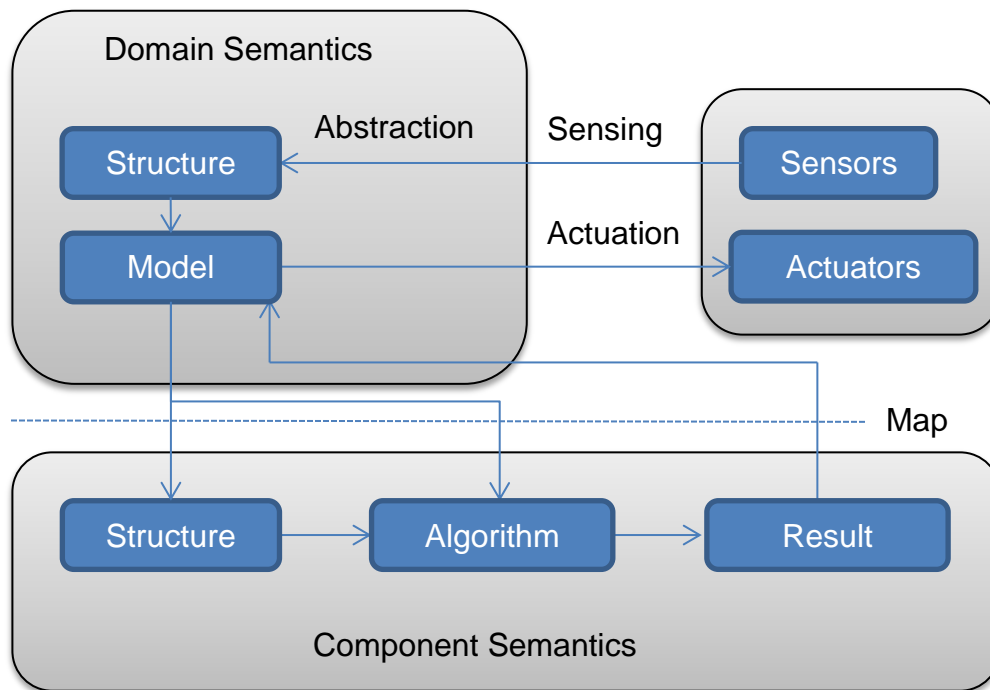


Figure 3: Overview of Semantic Mapping between System and Components.

There are three broad categories of semantic mapping deficiencies which occur.

- Discretization – A resolution loss of data such as representing a double as an integer, or a numeric as an enumerable can cause precision loss in the model which must be reserved as slack or the system can become unstable, dangerous, or expensive.
- Generalization – A loss of relationships such as the loss of a primary key in databases. Without being able to assume certain relationships hold many operations become more difficult. This difficulty amounts to a less efficient system, including the possibility that solution times are no longer feasible for the system.
- Simplification – A loss of functional complexity such as representing an integral as a Riemann sum. Similar to discretization this can cause further precision loss and can compound discretization errors.

In summary, CPSs necessarily deal with complex physical models. An accurate physical model can be extremely complex. CPS domain experts often exploit abstractions and relationships to make modeling costs manageable. When components don't support sufficiently

rich semantics to capture this information important information is lost in translation and the performance and safety of the system is compromised.

Academic planners were the components in my work and the CPS domains ranged from fleet logistics in unmanned aerial vehicles to thermal-aware scheduling in data centers. The planning problems were unable to be solved with or sometimes even expressed to academic planners because of semantic mapping deficiencies. Each case was a different, apparently unrelated, example of semantic mapping deficiency.

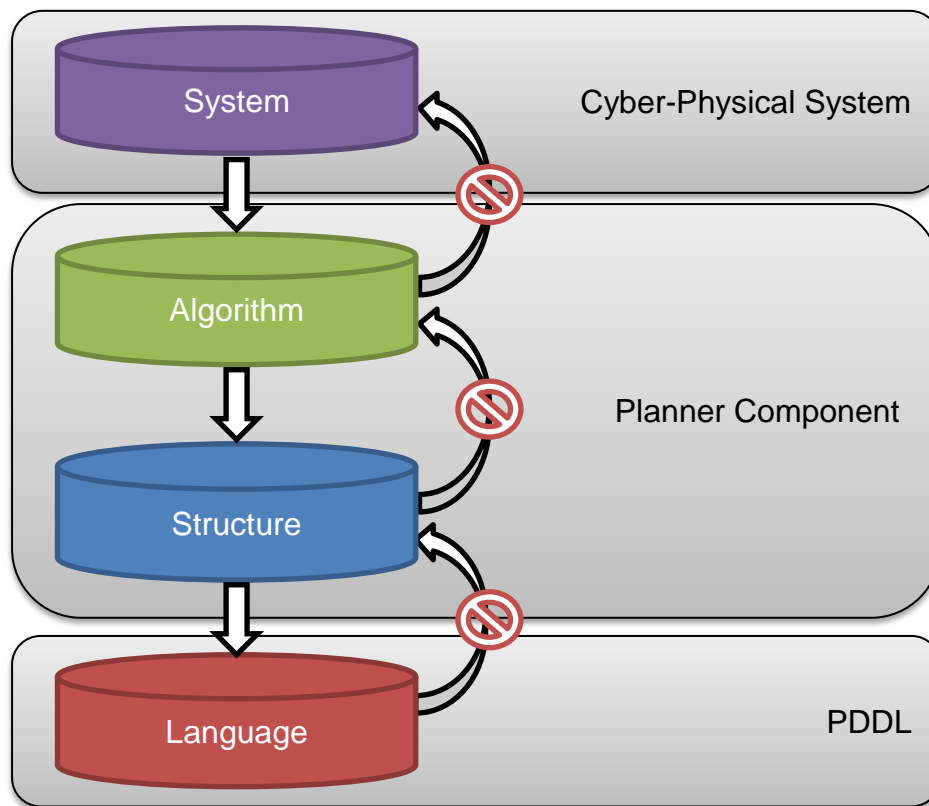


Figure 4: Design Stack of CPS and Planner Component.

Surveying the field of computer science one can conclude that I am not alone in experiencing difficulty using academic planners. There are many examples of what are essentially planning problems which are solved without using academic planners. Some broad examples include node traversal problems like internet routing and the traveling salesman problem which can be modeled using a cost optimal SAS+ representation, pathfinding in video games or fleet coordination which are typically solved by some form of A* or TBA* algorithm, and

robotic systems such as the DARPA grand challenge winner which solves numerous potential planning problems [8-12]. The domain experts in each case chose to use a domain-specific solution. It is possible in some cases this decision was motivated by ignorance of the planning field. However, even within the narrow field of planning literature where that is highly unlikely examples of domain-specific solutions are common. Figure 5 shows the results of a survey of the ICAPS conference application and robotics papers for the last two years (2014-2015) which hosts the international planning competition. This is the most fertile ground possible for domain-independent planners and yet a considerable number of authors chose a domain-specific solution for planning even in this environment.

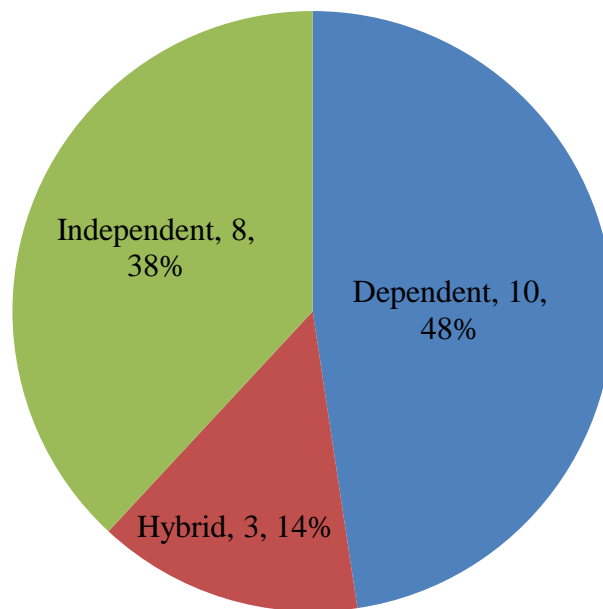


Figure 5: Survey Result of Application Papers in ICAPS conference.

These authors either cannot or have chosen not to use academic planning solutions. This remains true despite the intentions of the planning community to solve more complex and realistic domains. These authors must have their reasons for these decisions. I have observed several drawbacks to the approach the planning literature has taken as a whole which I feel contributed to this lack of adoption:

- The tools available for developing domains for an academic planner are far behind the support of major programming languages both in labor investment and effectiveness. Workshops such as KEPS encourage improvements in this area but the gulf remains.
- The solution space is often inscrutable. While the methodology which led to a solution is explained in great detail in the literature, it is often tedious to answer specific questions about the decision process of the planner for a given problem. For example, as a matter of quality assurance (QA) domain authors are understandably asked to answer questions about why the planner did not yield a plan of an expected form. This information is often not available after the planning process has completed and extracting it mid-execution can require working with planner source code. This is exacerbated by the fact that the middle-ware structures that planners parse problems into and which their heuristics understand are non-standard even if they are similar. These non-standard structures complicate the task of evaluating and selecting between multiple planners for a domain.
- The expressive capabilities of common languages such as PDDL 3.1, RDDDL, ANML, and SAS+ are each limited to a complexity which the supporting planner can parse and understand [12-16]. This is a problem with all code interpretive planning approaches. If the domain intrinsically requires more complexity than the language supports the problem is unsolvable by any planner using that language.
- Planning languages such as PDDL lack heuristic guidance mechanisms as a matter of philosophy [17].

The PDDL language was designed to be a neutral specification of planning problems. *Neutral* means that it doesn't favor any particular planning system. The slogan we used to summarize this goal was "physics, not advice." That is, every piece of a representation would be a necessary part of the specification of what actions were possible and what their effects are. All traces of "hints" to a planning system would be eliminated.

Contrast this with successful declarative industrial languages like SQL which include query hints that enable algorithmic variations in the underlying implementations. A planning equivalent would be flagging a specific predicate list as friendly for relaxation. Excluding them from the language only makes sense from an academic evaluation perspective. It does not make sense from the perspective of a language for serious problem solving.

These four drawbacks can be summarized as a lack of support, transparency, expressiveness, and control. These drawbacks are not intentional. They likely exist due to some combination of higher priority, competing needs of academic planners. These include low barrier to entry for planner authors, freedom of implementation enabling various middle-ware structures to be attempted, and strong delegation of responsibility between the domain author and planner. Support and transparency are largely conveniences. While still important a solution is still feasible using technologies with poor support and transparency. Expressiveness and control shortfalls on the other hand can render a solution impossible.

This work proposes an alternative to address these shortfalls; a planning library that facilitates domain expert knowledge injection by a variety dependency injection design patterns to overcome the usual structural and algorithmic limitations of declarative systems described in the introduction. This work fundamentally reconsiders the relationship between a planner and domain author. This paradigm shift arises from a philosophical difference and divergent design goals between academic solutions and industry solutions.

CHAPTER 1 - INTRODUCTION

BACKGROUND LITERATURE

Several of these shortfalls are already widely recognized and have some existing work attempting to address them.

A plethora of work exists from the KEPS workshop devoted to improving support/transparency. Shah et al. provide an excellent survey of knowledge engineering (KE) tools in planning [18]. They group existing tools by methodology and rate each methodology according to these metrics (quoted from citation):

Operationality. How efficient are the models produced? Is the method able to improve the performances of planners on generated models and problems?

Collaboration. Does the method/tool help in team efforts? Is the method/tool suitable for being exploited in teams or is it focused on supporting the work of a single user?

Maintenance. How easy is it to come back and change a model? Is there any type of documentation that is automatically generated? Does the tool induce users to produce documentation?

Experience. Is the method/tool indicated for inexperienced planning users? Do users need to have a good knowledge of PDDL? Is it able to support users and to hide low level details?

Efficiency. How quickly are acceptable models produced?

Debugging. Does the method/tool support debugging? Does it cut down the time needed to debug? Is there any mechanism for promoting the overall quality of the model?

Support. Are there manuals available for using the method/tools? Is it easy to receive support? Is there an active community using the tool?

The survey concludes that the KE tools available are particularly poor at the experience, collaboration, maintenance, debugging and, support metrics and that this is a significant deterrent to adoption of planning technologies. This directly supports my view.

Given the hardness of generating domain models for planning, many users are not exploiting automated planning but use easier approaches, even if they are less efficient [18].

Another recent survey has provided a review of nine approaches which learn a domain description from observation rather than relying on humans to encode the domain with similar conclusions that further work is necessary [19]. Other publications indirectly help the support shortfall, such as by improving visualization [20-21].

The expressiveness shortfall in particular has been recognized for decades and there have been significant semantic enrichments to PDDL since its inception as a purely propositional language to extend its scope to include time and numbers [22]. There exists a frontier in knowledge representation in planning languages which is gradually expanding. At the moment it lies in probabilistic planning with PPDDL and RDDL2 and planners like PROST and in continuous linear models with PDDL+ and planners like COLIN [23-26]. These works expand the expressiveness limitation of the language and expand the capacity of planners to handle new language features respectively.

There is evidence that this frontier will continue to expand. For example, recent work is attempting to extend PDDL beyond even these limits. Work on Satisfiability Modulo Theories (SMT) allows functions to be specified in external theory modules. Planning Modulo Theories attempt to apply SMT reading to planning domains [27]. More recently attempts are being made to apply this work using hybrid planning techniques to combine specialized solvers with general purpose planners [28].

Regardless of how far this frontier progresses there will always be domains which exist on the frontier or quite beyond it until the problem of hard AI is solved. Beyond this frontier lie problems of representational complexity which, while neither unfathomable nor unsolvable, remain decades out of reach if the frontier of knowledge representation progresses as it has.

Surrounding this frontier are various works on applications papers which the previous section showed the survey results for. The detailed breakdown of this survey is shown in the Table in Appendix A.

Domain-dependent results use custom languages or representations, or extend existing declarative languages or write a custom heuristic/planner specifically for their application. Hybrid results combine an off-the-shelf planner with a pre/post processor algorithm which abstracts out considerable portions of the domain from the planning problem and recombines the planner result with these domain elements to produce a different plan before actuating. Domain-independent results use an off-the-shelf planner or technology such as Partially Observable Markov Decision Processes (POMDPs).

The control shortfall is a topic that has fallen out of fashion despite its early success over 10 years ago [5]. Over a decade ago, work which capitalized on domain-specific knowledge was more common. In the 2000 and 2002 International Planning Competitions domain-configurable planners dominated the performance with planners like TLPlan, TALPlanner, and SHOP2 [54-56].

Authors of this early work believed that this direction was not only beneficial but indispensable.

... can often convert an intractable planning problem to a tractable one; i.e., it can often be the only way in which automatic planning is possible. [54]

The approach appears to be the key to the next order of magnitude scaling of state-space planning algorithms. Problems that could not be solved in days could be solved in seconds with additional axiomatic knowledge. [57]

Since then no domain-configurable heuristic languages have been officially developed. Rather, domain-configurable planners exist with no unifying representation. The lack of progress and apparent regression of domain-configurable work gives the general appearance of a cultural bias in the planning community against domain-configurable work.

Most recent work in this direction has appeared immediately outside the planning community in areas like logical programming and answer set programming [58-59]. All existing domain-configurable work imposes modeling limitations so that a heuristic may act. This is an unfortunate state of affairs for CPS owners and leads many applications to use simpler, less efficient technologies as demonstrated by the survey.

CHAPTER 2

DATA CENTER DOMAIN

The thermal modeling problem described in this chapter was the first thing I did with the Impact lab. The problem began with a machine learning task but became much bigger when it was proved that the underlying physical model was too complex to represent using a machine learning library. However, once we had the refined model to a more accurate one it was not clear how to use that model to solve the problem because it could no longer be input to a planner.

This chapter will outline the importance and basic function of the domain itself, cover the mathematical foundation of the transient thermal model, discuss why this model is problematic to represent in PDDL, show what an implementation in PDDL would look like, and then evaluate planner performance on intuitive examples of this domain compared to a library-based alternative approach.

CHAPTER 2 – DATA CENTER DOMAIN

DATA CENTER THERMAL MODEL

Due to their \$7.4 billion in annual electricity use, data centers are an important area in green computing. Researchers have developed numerous energy-aware approaches to reduce support infrastructure costs. Many of these improvements rely on a thermal model to predict the temperature at places of interest throughout data center facilities. These predictions are useful in evaluating potential changes in equipment utilization (e.g., which server to assign an incoming workload). However, the majority of this field of work only predicts steady state temperatures, ignoring critical temporal aspects of thermal behavior. This not only diminishes effectiveness but also can cause service-level agreement (SLA) violations and equipment damage through unforeseen temperature spikes.

To address these problems we developed and published a transient thermal model at IGCC12 that captures the transient behavior. This transient thermal model captures the behavior necessary to predict thermal hot spots before they arise [60].

The basis of the energy savings in our scheduling problem comes from creating a balanced thermal profile across the data center. Consider the efficiency of an AC unit as a function of its output temperature.

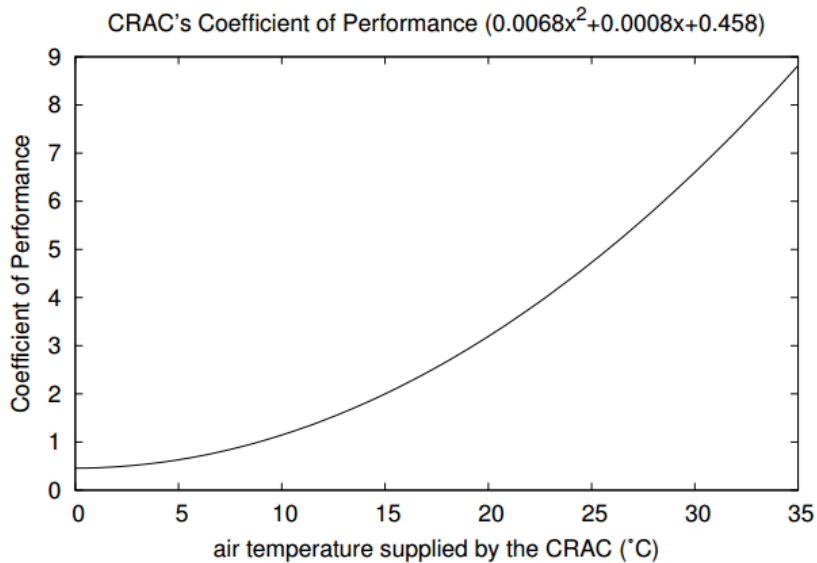


Figure 6: AC Coefficient of Performance

AC efficiency is defined in terms of coefficient of performance (COP). The COP of the AC is the amount of heat removed per energy consumed by the AC.

$$\text{COP} = \frac{\text{Heat Removed}}{\text{Work Consumed By AC}}$$

With a COP of 1, fifty percent of the energy expended by the data center will be the cost of cooling. Simply note that the lower the return temperature and the bigger the delta, the more energy is spent per Joule to remove it. In our data center example if the set of jobs produce an invariant total Joules of energy the cost of removing that energy from the system is higher if the air is being cooled more. When a chassis in the data center is significantly below its manufacturer-specified red-line temperature energy is allocated inefficiently in the room. In a perfect allocation all equipment is running at its redline temperature and the COP and data center efficiency is thus maximized.

If the temperature of a data center can be predicted with respect to a given schedule of jobs, a planner could provide temperature aware task schedules for the equipment and a temperature schedule for the AC to prevent wasteful overcooling of the data center. As long as the inlet of all equipment is kept below its manufacturer-specified red-line temperature and system performance is not overly affected, this efficiency gain causes no SLA violations. An example of the effect that load distribution can have on temperature is shown in the figure below. For this summary, it is sufficient to say that cooling efficiency of a data center is maximized when the inlet temperatures of all machines are equal to their redline temperatures. The more accurate a thermal model, the less slack that needs to be left when performing thermal-aware scheduling. This can lower the total cost of ownership of very large data centers by millions of dollars annually.

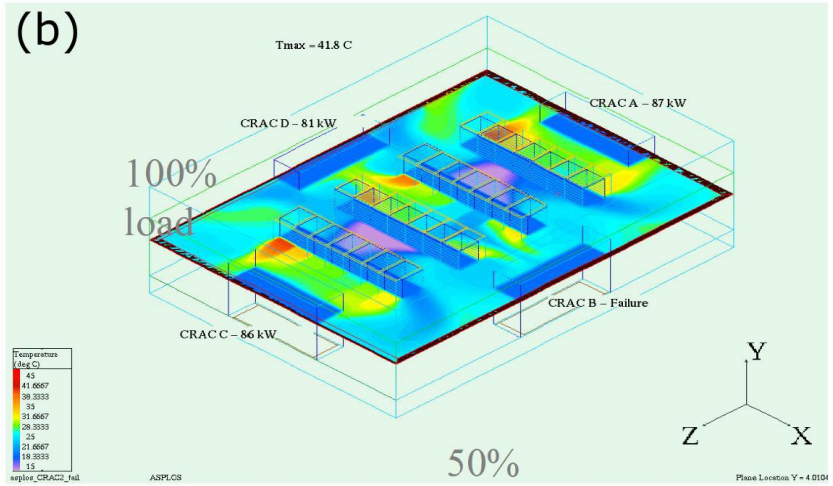
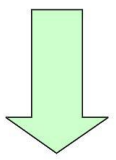
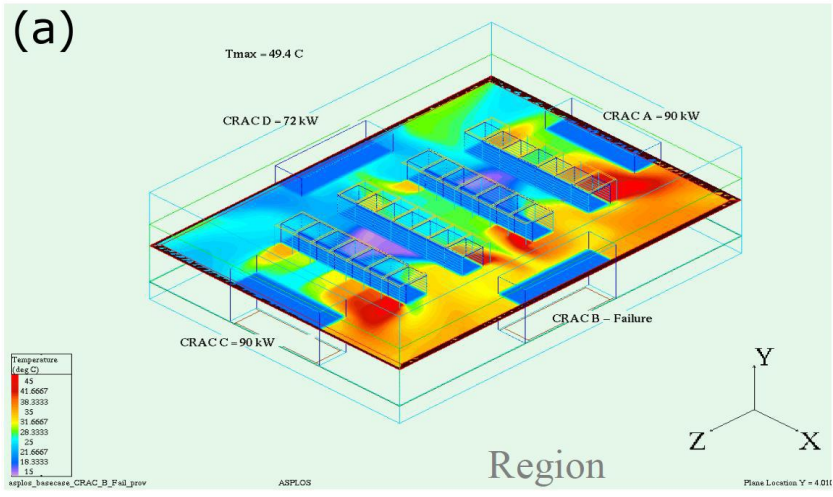


Figure 7: Effects of Thermal-Aware Scheduling [75].

Symbol	Definition
n	Number of points of interests: Chassis, AC inlets, etc.
t	Time
$T_{i+}(t)$	Inlet (sink) temperature of point j at time t .
$T_{j-}(t)$	Outlet (source) temperature of point j at time t .
$T_j(\infty)$	Starting steady state temperature of point j .
$c_{ij}(t)$	Temporal influence curve of point i 's temperature upon point j .
w_{ij}	Weighting matrix of point i upon point j .
f	Transfer function of point j

Table 1: Data Center Thermal Model Symbols.

Our model uses the symbols of Table 1, where source temperatures are initialized as steady state temperatures. The full equations for the model are below.

$$T_{j+}(t) = \sum_{i=1}^n w_{ij} \int_{-\infty}^t c_{ij}(\tau) T_{i-}(t + \tau) \partial\tau$$

$$T_{j-}(t) = f_j(T_{j+}(t))$$

This model accounts for air spreading out over time as it travels from location to location within the model using the integral of the temporal influence curves. Further, it accounts for air from multiple locations using a weighting matrix.

To calculate the temperature predictions of the model for a location at a time t :

- 1) A contribution temperature is calculated using an integral of a $c_{ij}(t)$ curve for each source in the model upon the desired location.
- 2) The contribution temperatures are averaged together using a weighting matrix to derive the predicted sink temperature for each location.
- 3) The sink temperature is modified by a transfer function to produce the source temperature to be used at future times.

The transfer function for chassis is a simple constant load value based on the utilization of the chassis. The AC has a step-wise linear cooling function which motivated the model to begin with.

The publication discusses further aspects of the problem such as assumptions, a proof that equilibriums are impossible for most configurations, how to learn the various model parameters, and comparisons against previous work but for the purpose of this work all that is needed is a basic understanding of the operators of the model and what they mean.

As will be shown this model is problematic to represent in PDDL and even much simpler variants of this domain suffer significantly from semantic mapping deficiencies to planning semantics.

CHAPTER 2 – DATA CENTER DOMAIN

PLANNING CONSIDERATIONS

On trying to implement this model in any planning language the first problem encountered is that the calculus required for the temperature calculation of is not supported in a continuous manner. Some languages like PDDL+ allow a description of continuous change of the form $\partial V / \partial t = f$. An example in PDDL+ is shown below

```
(:process charging
:parameters (?r - rover)
:precondition (and (<= (charge ?r) (capacity ?r))
                 (in-sun ?r)
                 (charging ?r))
:effect (increase (charge ?r)
                (* #t (charge-rate ?r))))
```

This process is modeling battery charging in the rover domain. This is a standard domain used in the international planning competition (IPC) based on the challenges of the Mars rover. The Mars rover CPS uses a planner component as part of its solution which needs to account for the charge of its batteries. The process above states that while the rover, ?r, has remaining capacity that could be charged, is in the sun, and is charging, the amount of charge is continuously increased by the charge rate.

This looks promising but the formula for f cannot refer to values at any time other than the present and the operators it can include are limited to the arithmetic set (+, −, ×, ÷). Two approximations are necessary because of this.

- The current state must contain all values necessary for the calculation. This requires discretizing all temperature values (T_{i+} , T_i) by sampling their values distinct time points rather than modeling it continuously. This is an example of Discretization in semantic mapping.
- The calculation of T_{i+} must be approximated using a Riemann sum. This is an example of Simplification in semantic mapping.

To demonstrate this, consider the simplified model below which removes recirculation (the weighting matrix) and the complexity of the temporal influence curves entirely. If we have a continuous model:

$$T_i(t) = L_i(t) + \frac{\int_{t_{is}}^{t_{ie}} T_{AC}(\tau) \partial \tau}{(t_{ie} - t_{is})}$$

where T_{AC} is the AC temperature and $L_i(t)$ is a continuous load function, mapping to planning semantics requires discretizing T_{AC} and T_i as shown below.

$$T_i = L_i + \frac{\sum_{i=0}^n T_{AC}(\Delta_i)}{n+1}$$

These approximations require some complex mechanics shown in the figure below.

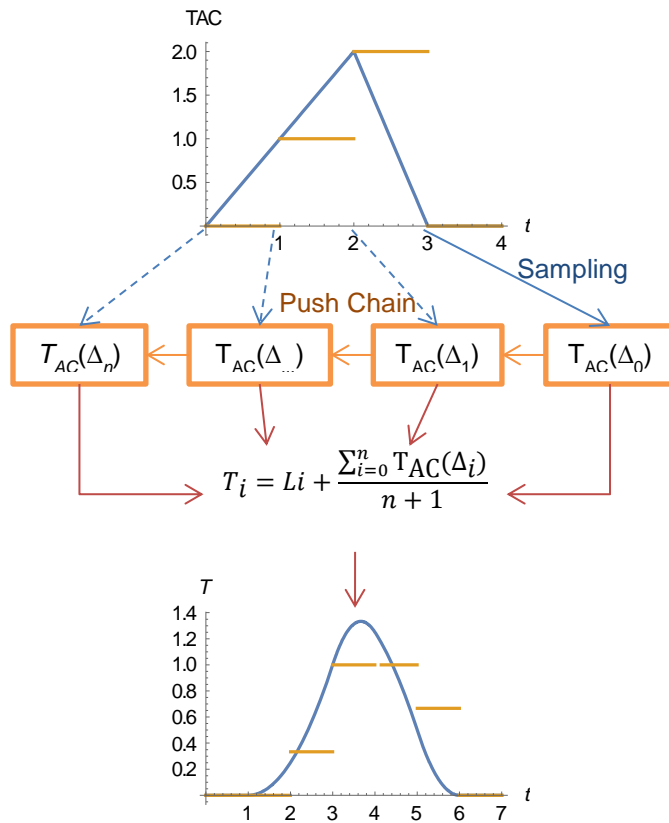


Figure 8: Discretization method for implementing thermal model in PDDL.

An ordered set of sample variables, $T_{AC}(\Delta_i)$, are created to hold historical AC temperature values that are sampled at a regular discretization period, Δ , by an artificial

modeling action. To maintain semantic consistency, past temperature values are pushed from one variable to the next, $T_{AC}(\Delta_{m+1}) = T_{AC}(\Delta_m) \forall 0 \leq m \leq n$, resulting in $n + 1$ variable modifications per Δ as time progresses. These sample variables can then be used to calculate T_i . These approximations come with several unwanted consequences.

First, since no optimal solutions exist for which T_{AC} is discretized, no optimal solutions can be found with these planning semantics. The optimal solution cannot even be expressed as a plan using state of the art planning semantics for basically the same reason a polynomial function can only be approximated by line segments.

Furthermore, discretizing T_{AC} leads to modeling precision loss which carries over into T_i . The T_i calculation simplification compounds this problem. Underestimating T_i can cause overheating which can cause equipment damage and SLA violations from equipment shutdowns if the inaccurate model is acted upon. Even if an acceptable level of precision loss can be found, this is an undesirable outcome of nothing more than semantic mapping. It is impossible to represent this domain accurately or produce an optimal solution using state-of-the-art planning semantics.

Second, the net result of the process shown in Figure 5 is an explosion in the number of variables which model T_{AC} from 1 to $n + 1$. Since the correlation between these variables is lost in translation, they seem independent to proof-based heuristics and such approaches become quickly overrun. Planners which consider combinations of AC temperatures independently are effectively faced with a more complex version of the NP-Complete knapsack problem.

This observation reveals two opposing forces. First, if the discretization period is too large unacceptable precision loss renders the system dangerous or unusable. If the discretization period is too small the variable count explosion renders the system unusable.

For a sense of scale consider several real values from our measured data center. If the heat exceeds the maximum temperature for more than about a second the hardware safety will kick in and the equipment will shut down. When this happens, data integrity can be compromised. At best, restarting the machine and recovering the work takes several minutes during which the performance of the system has suffered. This can cause punitively expensive SLA violations if

there was insufficient slack in the schedule to recover from the sudden loss of a machine. This implies the system requires a Δ of less than 1 second.

The number of discretization windows, n , is on the order of minutes. In the measured data center air could take about 10 minutes to cycle which yields $n \approx 600$. Solving a problem of this scale is already difficult. The full problem is even worse though. In the full domain model the calculation for T_i considers air recirculation between chassis as well as the AC which would increase this value to $j(n + 1)$ where j is the number of thermal points of interest. Large data centers can have tens of thousands of chassis, yielding millions of independent variables. Polynomial time solutions are quickly overrun.

Furthermore, heuristically there are several more problems. First and foremost, the branching factor of possible job schedules is at worst N machines by J jobs per time step. Modern planners rely on sophisticated heuristics to prune this tree based on an action's 'usefulness' in order to reduce the search space to a manageable set. However, 'usefulness' is determined by how an action affects some part of the state. This is problematic because the action that is causing the thermal constraint violations (the heat modeling action) is distantly removed from the action that actually caused the problem (the job assignment action). Therefore, to prune the space at all, a heuristic must consider indirect effects. The further removed those effects are from the action considered, the more costly such heuristics become. In this domain, the important consequences of scheduling a job can be thousands of state transitions deep depending on the discretization interval size and time span of the c curve. Furthermore, each job assignment action indirectly affects the entire temperature model of the data center, so heuristics cannot safely ignore any of the temperature changes anywhere in the model. This becomes extremely problematic as server count and time resolution increase and the sheer number of effects to consider multiplies.

Second, as already mentioned, PDDL does not support planner guidance syntax. This makes the above problem inescapable but also independently wastes many resources. For example, take the simple statement: 'the heat modeling action must be executed at each time step'. To capture this behavior in PDDL we are limited to describing constraints or goals on the

domain that the planner can use to infer that it must apply the heat model action. This is understandably inefficient. After an action is selected, the planner returns to the action selection process. This action selection process can be time consuming and pointless when, by design, there is only one useful action that could potentially be taken, such as the heat modeling action.

Unrelated to the planning heuristic the representation of this domain is awkward in PDDL because the domain file is dependent on the number of chassis which is something that is usually associated with a problem rather than a domain file. Maintaining a solution therefore requires a problem generator to also create domain files as well. This adds programmatic complexity to the domain which feels unwarranted.

CHAPTER 2 – DATA CENTER DOMAIN

PDDL IMPLEMENTATION

Despite the concerns of the previous section a formal basis for a PDDL implementation is provided here.

To calculate $T_{i,t}(t)$ with maximum c_{ij} duration of 5 ($c_{ij}(t) > 0 \forall t \ t_i < t < t_e \ \& \ t_e - t_i \leq 5$), with a discretization window of 1 second, the contribution temperature of location 1 on itself looks something like this:

```
(+
  (* (cCurveDt5 ?server1 ?server1) (temperatureOut ?server1 ?dt5))
  (* (cCurveDt4 ?server1 ?server1) (temperatureOut ?server1 ?dt4))
  (* (cCurveDt3 ?server1 ?server1) (temperatureOut ?server1 ?dt3))
  (* (cCurveDt2 ?server1 ?server1) (temperatureOut ?server1 ?dt2))
  (* (cCurveDt1 ?server1 ?server1) (temperatureOut ?server1 ?dt1))
)
```

Note that if the discretization window changes or c_{ij} changes this function needs to be rewritten accordingly. This complexity of this computation increases as the simulation time step approaches zero, the span of the temporal influence curve expands, or the number of servers increases. Alternatively, we could have one c_{ij} list that uses a new type of object to facilitate the same logic.

```
(+
  (* (cCurve ?server1 ?server1 ?cCurveOffset5) (temperatureOut ?server1 ?dt5))
  ...
)
```

All this really does is to reduce the list count but increase the cost of each predicate lookup and add new parameter objects to the action. It further complicates search, which will hurt planner performance slightly. ?dt5 could not be reused because it refers to an offset of 5 from current time rather than a constant offset of 5.

Next the weighting matrix is used to average the contribution temperatures calculated above. The above contribution temperature would replace T11 here. An equally long calculation replaces each other Txx value.

```
(assign (temperatureIn ?server1 ?currentTime)
  (+
    (* (w ?server1 ?server1) T11)
    (* (w ?server2 ?server1) T21)
    (* (w ?server3 ?server1) T31)
    (* (w ?server4 ?server1) T41)
  )
)
```

Lastly the outlet temperatures are calculated from the inlet temperatures and the transfer function.

```
(assign (temperatureOut ?server1 ?t) (+ (temperatureIn ?server1 ?t) (temperatureIncrease ?server1)))
```

This assumes that temperatureIncrease is maintained by whatever action schedules the job on a server.

A major concern with the above implementation is that it is relatively rigid to a specific data center and time resolution. For instance, if more locations are added, the above computations are not general enough to use without being recoded. The same is true for longer spanning temporal influence curves.

Doing this at scale required a file generator to produce the PDDL files from a concise XML problem description. Appendix B and C contain an example PDDL domain and fact file for this implementation generated using T4 templates.

CHAPTER 2 – DATA CENTER DOMAIN

EVALUATION

There are several factors that would be useful to measure to determine the relative advantage of an alternative approach to PDDL with respect to the issues raised above.

1. The cost of increasing time resolution and location count for a given domain.
2. The overhead cost of just the actions and their coordination.
3. Upper bound complexity limits.

The first two points above correspond to the concerns raised in the planning considerations section. The third point is to justify whether an alternative approach is necessary. For all of the above evaluations, one or more planners would need to be deployed and fed the PDDL files.

To separate the cost of job scheduling from the cost of the heat modeling itself, we can evaluate this domain with various location counts and time resolutions but absent jobs and compare the performance against the JPDL implementation.

To facilitate comparison, the XML problem description used to generate the example PDDL files in the appendix is actually loaded into the planning modeling semantics object web before being used by the template engine. This enables the XML problem description to be used both for outputting to a PDDL planner and directly by the planning library for comparison. However, no planning heuristic exists at present for the data center domain.

CHAPTER 3

PDDL SEMANTIC LIMITATIONS

Several semantic mapping deficiencies for PDDL have already been defined in the data center domain in Chapter 2. Here I reflect on several more specific limitations and the reasons these limitations exist.

PDDL actions cannot represent numeric variables as part of the action header. Instead, the header can only support objects. This has the minor and obvious effect of inconveniencing developers by forcing them to invent container predicate lists if they wish to refer to a numeric in the action description, but it also has the more important but subtle effect of limiting the actions that can be described. For example, we can express an action to move a vehicle to a location:

```
(:action Move :parameters (?Vehicle – Vehicle ?CurrentLocation ?TargetLocation – Location)
  :precondition (at ?CurrentLocation ?Vehicle)
  :effect (and (not (at ?CurrentLocation ?Vehicle))
              (at ?TargetLocation ?Vehicle))
)
```

However, attempting to support an action to move a vehicle east by a numeric unit is clumsy.

```
(action Move-East :parameters (?Vehicle – Vehicle ?VehicleLocation – Location ?Distance – Value)
  :precondition (at ?VehicleLocation ?Vehicle)
  :effect (increase (LocationX ?VehicleLocation) (Value ?Distance))
)
```

The limitation of only using objects in the header has forced the designer to create an entire predicate list to contain values for numeric access (Value). This is obviously inconvenient. The more important but subtle effect is the limit this has placed on expressiveness. The domain is only as complete as the value predicate list is. If this predicate list only holds a single object, any plan produced can only contain actions that move a vehicle east by that value. It would require **infinite** objects in the Value predicate list to completely capture the logical concept of moving a vehicle east. To the degree that this list is incomplete, the expressiveness of the domain is incomplete. This particular implementation also has the additional unintuitive characteristic and

potential bug where multiple vehicles would be moved if they were to share the same location object in the (at) list.

This expressive limitation exists by design to limit the branching factor in search. The number of different objects may be numerous, but for any given problem there are a countable, finite number of plans that are possible up to any arbitrary plan depth N . This is a very useful characteristic for a planner author who wishes to do proofs for a paper. It is a necessary condition for the property of completeness. Consider an alternative representation where the header could contain a continuous value rather than what is effectively an enumerable type. By definition, there are as many possible actions as there are possible numbers. The number of actions becomes uncountable unless you limit the parameter to a set of numbers by some other means.

Another problem becomes evident if we try to expand on a domain that uses the location to location approach above. Consider a travel action where one or more effects are dependent upon the Euclidian distance between location objects. In PDDL, there is no mechanism for calculating a Euclidian distance. Instead, all calculations more complicated than the BNF of PDDL must be evaluated in advance and the results must be written to a fact file in a predicate list containing object parameters.

```
(= (EuclidianDistance ?Location1 ?Location2) 1)
```

In our travel action, this can add up to n^2 predicate list values rows of overhead, where n is the number of locations. However, most of these rows will probably never be needed.

Other complex functions can be impossible to represent in this way because important properties of the object may change during the planning process. For example, if a location object ever needs to move, it would invalidate the entire EuclidianDistance predicate list cached in the fact file. There is no mechanism by which the planner can update these rows accurately as they needed to be evaluated in advance. Eventually, a developer is either forced to utilize tabling with these limitations or utilize PDDL as a Turing complete system so far removed from the intent of actions that a planner will fail when it attempts to use the domain. Examples of this are provided in Section 4 under multi-part actions.

All PDDL primitive expressions are predicate based. Objects in PDDL are essentially just identifiers. Object oriented programming can't be supported in a straightforward way. Predicates are essentially just multi-key, single-value tables. Having these structures as the only form of data representation other than constants will inevitably slow down a planner and cost extra memory. To understand this, contrast object oriented field access, which is constant time, to a predicate table lookup, which is, at best, hash table lookup time. In the first travel action above, the only reason for the precondition, (at ?CurrentLocation ?Vehicle), was to bind the ?CurrentLocation object to the location associated with that object in the (at) predicate list. Juxtapose this to the object oriented description of Vehicle.CurrentLocation. The difference in time is unavoidable because predicate tables can support reverse lookup in a way that field access cannot. One cannot use a pointer to CurrentLocation to get all Vehicles that have a CurrentLocation pointing to the same object. In PDDL, accomplishing this is trivial, but computational cost occurs regardless of whether this functionality is required. This does not at all affect the expressiveness of JPD. In fact, it is in fact possible to do a translation of any object oriented structure to a predicate list structure in a straightforward way. The difference is only the speed of using these structures. PDDL offers the domain author no choice.

The reason for this limitation is likely design simplicity. A predicate list is the most general structure and is very well suited to both forward and backward chaining planners. The more complex the underlying BNF the higher the barrier to entry is. Because PDDL has an origin based in an academic planning competition the barrier to entry for planner authors needed to be low enough that multiple teams would compete. Adding representational diversity may increase usability but it is academically uninteresting and would only increase the barrier to entry for planner authors.

In general expressiveness limitations exist because the focus of PDDL is on the planner authors instead of domain authors.

The second shortfall of PDDL is a variety of software development concerns ranging from the single file requirement, to a lack of software reuse, missing IDE's, a lack of support for modern concepts like inheritance, and the unintuitive and unfamiliar structure of PDDL to an

average software developer. In the following chapters, I explore two approaches that address these concerns.

First, I demonstrate an attempt to utilize a subset of a modern language (Java) and translate/compile that representation into PDDL. This allowed developers to use modern tools such as Eclipse to develop a domain. However, this compilation also destroys many potentially useful elements that higher level languages have and PDDL does not such as classes and loops. Alternatively, a planner could try to use JPDL directly. It is unclear how a domain-independent planner would be able to utilize some of the above elements. If a way could be found to utilize some or all of these elements, it would be of interest to the knowledge representation and planning communities. Even if it were proven that these elements could not be utilized efficiently by a domain-independent planner, it would only raise the question of whether domain-independent planners are the end-all be-all, or whether rapidly configurable domain-dependent planners would be a preferable tool to solving many problems.

The planning community is already showing that domain-dependent planners have a future. In the 2011 international planning competition the portfolio planning approach was a clear winner. Portfolio planning is an approach to planning that takes advantage of the relative strengths of different planners at solving different domains. A meta-planner analyzes a domain for computationally important characteristics, and chooses the planner that has been shown to perform best for similar domains. This approach makes each planner domain-dependent in the sense that it is only appropriate to use it on certain domains.

Second, I address the possibility of implementing a planning library in a modern language. In such an environment, actions are executed rather than interpreted. This offers expressiveness equal to the underlying language, but will require extra effort from domain authors to specify not only the action, but also how a planner should use it. This challenges the breakdown of responsibility between planner and domain authors, but it offers a glimpse of industrial grade solutions to planning domains which would previously have been inexpressible/unsolvable by academic planners.

CHAPTER 4

JPDL TRANSLATOR

This section focuses on work published in the ICAPS 11 conference KEPS workshop [61]. Since this publication several changes have been made to the language independent semantics relating to how State Variable sets and primitives are used and what information an Effect has access to but the temporal semantics and basic terms remain largely unchanged.

JPDL (Java Planning Description Language) uses a subset of the existing Java syntax, borrows the semantic meaning from Sun's JVM for this subset, and integrates the semantic meaning of PDDL with this subset. This enables translation software to transform a Java compile-friendly syntax into a PDDL representation. Such a system can theoretically be significantly easier to use than writing PDDL directly and just the exercise helps to identify the relative advantages and disadvantages of PDDL's chosen format by juxtaposing the underlying semantics.

The fact that JPDL syntax is a subset of Java allows JPDL to benefit from a number of trends and improvements in software languages of the last decades. The first such trend is the rise of development environments such as Eclipse and Visual Studio. Many developers prefer IDE's because of the benefits they provide, such as real time feedback in code correctness. Languages such as PDDL, which are so entirely different from what IDE's today support, lose access to these benefits. Additionally, developers lose access to the rich feedback compilers provide for well-supported languages at compilation time. Given the difference in human resources devoted to industrial grade compilers compared to planners like FF, LAMA, and SGPlan, it is only natural that the feedback provided by existing planners on syntactic errors is more sparse and less informative than those provided by Visual Studio or JVC [62-64]. Another trend is the decline of Lisp-like syntax which PDDL is still based on for legacy reasons. This legacy structure fundamentally prevents compilers from being able to diagnose certain syntactic errors that are possible to diagnose in more modern syntaxes such as Java or C. One last trend is the advancement of newer programming paradigms such as object-oriented code which provides certain types of information not available in PDDL. Utilizing this information properly

could increase the efficiency of planners. Taken together, the benefits of these trends increase the usability of JPDL compared to a number of other planning description languages.

CHAPTER 4 – JPDL TRANSLATOR

LANGUAGE INDEPENDENT SEMANTICS

The following table contains the common symbol list of the language independent modeling semantics JPDL defines. These symbols are each formally defined below.

Symbol	Meaning
W	World Model
S_L	State list
E_L	pending Effect list
C_L	Constraint list
G_L	Goal list
S	State
V	State Variable set
P_v	Paired Identity and Primitive Set
E	Effect
t	time

Table 2: JPDL Semantic Symbols.

A **State Variable** v consists of a string **identity** i , a paired **Identity and Primitive Set** P_v of value types, and a **State Variable** set V which allow for recursive hierarchies.

A **State** S consists of a paired Identity and Primitive set P_v , a State Variable set V , and **time bounds** $[t_i, t_e)$ for which these mappings hold.

A **State List** S_L consists of a time ordered set of states with continuous time bounds ranging from a beginning time t_i to an eventual time t_e . This provides a timeline of how values change over time. The current State of the State List is the state with $t_e = t^\infty$ and $t_i =$ current time. States before the current time are considered final and immutable.

An **Effect** E is a function $f(t, P_v, V, W)$ that uses a paired Identity and Primitive set P_v , a set of State Variables V , and a World Model W with a State List with current time t , to produce a new World Model \hat{W} with modified State Variables on the current State, pending Effects, constraints, and goals. A pending Effect is an Effect of a World Model where the time t of the Effect is greater than the current time of the State List of the World Model.

A **Constraint** C is a function $f(t_i, t_e, P_v, V, W)$, that uses a paired Identity and Primitive set P_v , a set of State Variables V , and a World Model W , to evaluate a whether a pattern contained

within f is matched by W within the range $[t_i, t_e]$. If the pattern contained within f is not matched the WorldModel is inconsistent and the constraint has violated.

A **Goal** G is a function $f(t_i, t_e, P_v, V, W)$, that uses a paired Identity and Primitive set P_v , and a World Model W , to evaluate a whether a pattern contained within f is matched by W within the range $[t_i, t_e]$. When the pattern contained within f is first matched by a State of W the goal is considered to be satisfied.

A **World Model** W consists of a paired Identity and Primitive set P_v , a State list, a pending Effect list, a Constraint list, and a Goal list. It contains all information required to apply pending Effects until the current time of the State list matches the eventual time or all goals have been satisfied and no constraints have been violated.

A **Problem** P is an initialization function that returns a World Model that initially violates no Constraints and has unsatisfied Goals.

A Plan is a list of Effects which can be added to pending Effect list of a WorldModel provided by a Problem and validated by Advancing a WorldModel to an eventual time using the algorithm below.

CHAPTER 4 – JPDL TRANSLATOR

WORLD MODEL ADVANCEMENT ALGORITHM

JPDL defines language independent semantics for modeling. This algorithm is used to simulate a model, advance the current time past the time of all Effects, and check the evolving World Model for consistency and validate a plan if one exists.

- 1) A World Model W is initialized by a Problem P .
- 2) Initialize and add all plan Effects to E_L .
- 3) Select all pending Effects with min time $t < t_e$ to yield a next Effect set E_n . If $E_n = \{\}$ Go to step 7.
- 4) Select the current State from S_L . If the current State does not have $t = t$ advance the State List:
 - a. Split the current State by creating 2 states S_1 and S_2 with identical P and V to S . S_1 has t_i equal to S , $t_e = t$, and becomes the new current State. S_2 has $t = t$ and t_e equal to S .
 - b. Finalize S_1 .
- 5) Apply all Effects in E_n to produce a new World Model \dot{W} .
- 6) Set $W = \dot{W}$, Go to step 3.
- 7) Select and Finalize S_e the tail state of S_L .
- 8) If Goals remain unsatisfied fail.

To finalize a State S :

- 1) Check Constraints in C_L that overlap S . If any Constraints are violated fail.
- 2) Satisfy Goals that overlap S . If all goals are satisfied then pass.

There have been several changes to the JPDL semantics and advancement algorithm since publication. During the process of implementing the JPDL semantics in `c#`, creating planning semantics that utilize these modeling semantics, and solving several domains several changes occurred to the JPDL semantics I originally published.

The terminology has evolved in many ways. State Lists are not just consistent with Constraints, but rather when a Constraint is checked it might be violated by a World Model. The World Model semantic did not originally exist, so Effects had to take a State List, pending Effect List, and Constraint List. This led to the awkward exclusion of a Constraint List being accessible to a Constraint even as I talked about how Constraints could modify the Constraint List. Goals were originally just a special type of Constraint that had a start and end time of the eventual time. While a domain author could still choose to do this to only check Goals at the eventual time, the current approach is more flexible.

CHAPTER 4 – JPDL TRANSLATOR

TRANSLATION

JPDL was created in an experimental attempt to improve the usability of PDDL. The basic notion was that Java had superior tooling and support and would be more familiar to average developers than PDDL from a syntax perspective. Because the same basic notions of expressions and change exist in both languages, it was thought that if a description could be provided using a subset of Java based on tightly regulated reserved identifiers which provide a semantic meaning in PDDL, a translation could be created from JPDL to PDDL. From an academic standpoint, any syntactic elements that posed fundamental problems in translation were interesting.

Supported Syntax

Below is a table of what portions of Java syntax were considered, and a summary of the supportability of each element. Even these elements require a restricted version of the JPDL language independent semantics.

Syntax Element	Java Example	JPDL Supported	PDDL Supported
Identifier	A	Yes	Yes
Class Inheritance	Class ClassName	Some	No
Type casting	(ParentClass) subclassIdentifier	No	No
Scope keywords	public, protected, private	Yes	No
Object member access	Object.fieldName	Yes	Some
Logical expression	A && B	Yes	Yes
If Statements	if(a) { statementBlock }	Yes	Some
Assignment statement	A = B	Yes	Yes
Statement blocks	A=1; B=1;	Yes	Yes
Variable definition	Int a;	Yes	Yes
Function calls	functionName()	Some	No
Loops	for(int x=0; x<5; x++)	Some	No
Function calls	functionName()	Some	No
New statements	new Object()	Init Only	Fact Only

Table 3: Java Syntax Support Summary

Identifier

In the most basic sense identifiers are a symbol->reference mapping. The JPDL translator supports local and static identifiers and field access on object identifiers in scope. Due

to assignment statements the meaning of an identifier is not constant over the course of a statement block but rather must be maintained in a symbol table.

The type of the identifier is crucial in translation. Any identifier that refers to a state variable of a state must at some point map to a predicate list in PDDL.

Class Inheritance

Classes in JPDL are just collections of fields and methods. The most straightforward mapping of classes to predicate lists yields a list for each class-field pairing with either a boolean, numeric, or object return type. Certain classes, such as Effect, have reserved semantic meaning for planning and result in translation time errors if used. Namespaces are ignored during translation as implemented but could be added meaningfully later. Inheritance is understood by first gathering all the identifiers for each class and then combining sets of valid symbols using the scoping keywords. Method overriding is not supported but could be in principle. Consider the following classes with basic inheritance:

```
class ParentClass
{
    public int parentClassField;
}
class SubClass : ParentClass
{
    public bool boolField;
    public int subClassField;
    public SubClass objectField;
}
```

This produces the following predicates in PDDL.

```
(:predicates
    (ParentClass-parentClassField ?o1 - ParentClass)
    (SubClass-parentClassField ?o1 - SubClass)
    (SubClass-boolField ?o1 - SubClass)
    (SubClass-subClassField ?o1 - SubClass ?o2 - SubClass)
    (SubClass-objectField ?o1 ?o2 - SubClass))
```

Note that function calls are always either tables to a predicate list or compiled into the methods that call them similar to macros.

Type Casting

PDDL types do not support inheritance. This forces the JPDL library to make an uncomfortable choice between supporting casting between types, and supported PDDL types at

all. This stems from the fact that each PDDL object is exactly one type, rather than a set of types based on inheritance. When allowing PDDL types, if a domain author attempts to upcast a subclass into a parent class, problems occur when attempting to reference that parent class's field as the sub-class object will not exist in the parent-class predicate list.

We could address this issue by removing typing entirely from the PDDL translation, but this would hurt PDDL runtime irrecoverably for some domains.

Scope Keywords

When translating the defined classes into their predicate identifiers, scope information is used to decide which of a parent classes fields to include in the subclass. This limits method declarations and references from a class in the same way. For each type a list is created of valid symbols it can refer to. This list is checked each time an object field or method is referenced. All of this information is used to generate translation time errors to PDDL, but is not present in the resulting PDDL translation.

Object Member Access

Object member access occurs as part of larger expressions and cannot exist as a statement by itself. This includes method calls on classes.

When encountering an object field access, it can be replaced with a reference of the appropriate predicate in the target PDDL expression.

```
subClassObject.boolField => (SubClass-boolField ?subClassObject)
```

In this case subClassObject is an instance of type Subclass. When referred to in a Statement block, in an Effect for instance, the identifier used in code will map to the predicate list in the translation.

Logical Expressions

Both Java and PDDL use left sided expression evaluation. This makes expressions straightforward to evaluate for the most part.

```
A + B + C => (+ (+ A B) C)
A || B => (or A B)
```

Strict type checking is done for the operation in question. Implicit and explicit type casting are not supported, as mentioned above.

If Statements

The basic BNF is

```
ifExpression =>
if(BooleanExpression)
{
    StatementBlock
}
else IfExpression
```

Single if expressions are straightforward enough.

```
if(A) S1 => (when (A) (S1))
```

If-else blocks turn out to be a little trickier because PDDL does not support the notion of else. Still, it can be unfurled as if someone wanted to run each portion in parallel (which many planners likely do).

```
if(A) S1
else if(B) S2
=>
(and
    (when (A) (S1))
    (when ((and not(A) (B)) (S2))
)
)
```

Nested if-else statements similarly cause nesting.

```
if(A)
{
    S1
    if(B) S2
}
=>
(and
    (when ((and (A) (not B))
            (S1))
    (when ((and (A) (B))
            (and (S1) (S2))
)
)
```

This can cause significant repetition of Boolean expression evaluation and a corresponding performance hit in PDDL domains depending on the planner.

Assignment Statements

The basic BNF is

Identifier = Expression;

All assignments statements in PDDL change a predicate list. There are no other mechanisms for persisting data across actions. A direct translation for numerics of an assignment statement creates an assign expression in PDDL.

A=1 => (assign A 1)

A += 1 => (increase A 1)

A -= 1 => (decrease A 1)

For references (A=B) the direct translation is a little less intuitive. A reference is a non-value type in both Java and PDDL. Prior to PDDL 3.1 when this was written, predicate lists could not return non-value types so a pointer in PDDL was basically an association of two objects in a list whose meaning is that object A points to object B. A very brief recap of PDDL predicate lists may be helpful. Consider the following predicates definition in PDDL.

:predicates (PredicateListName ?o1))

Each predicate list is an association of a fact about an object. A translation to English is "PredicateListName is true about o1". If the name of the predicate list was HasRock, we would expect it was used in a way that every object is an owner of a rock. When authoring PDDL without types, it is common to have a predicate list for each type. (ObjectType o1) defines object o1 as being of type ObjectType. In the spirit of this, the following predicate list is saying that o1 points to o2.

(PointsTo ?o1 ?o2)

To change this fact in a way consistent with Java is to negate the first row and assert a new association. Simply adding a new association is insufficient.

A=B => :effect (PointsTo ?A ?B)

This uncomfortably leads to:

PointsTo Predicate List	
A	WhateverAPointedToBefore
A	B

To negate the existing row we actually need a variable first in order to express the intended delete. There are two options, both of which are computationally bad. We can either use a for-all statement to delete all entries of the predicate list associated with A (there should only be one) or we can find and delete the previous row by introducing a new object to the action.

Option 1:

```
:effect (and
  (PointsTo ?A ?B)
  (forall (?APrev) (not (PointsTo ?A ?APrev))))
```

Option 2:

```
(:action ActionName (:parameters ?A ?B ?APrev)
  :precondition (PointsTo ?A ?APrev)
  :effect (and (PointsTo ?A ?B)
              (not (PointsTo ?A ?APrev))
            )
)
```

The latter approach is considered faster, is more common, and is supported by more planners so it is the approach I chose for the JPDL translator. But think for a moment about the consequences of this translation compared to the original. It is trivial to see that the translation of the assignment statement is not localized. It has changed the code at places outside of the statement such as by adding a parameter to the action description. Also, the translation is computationally slower than the original. Both because the list must be changed in two ways where previously a single pointer needed to be changed and because a reference to APrev needed to be found for the translation which is at best a HashTable lookup time while the original didn't even need to dereference the field. Furthermore, the number of possible actions just grew exponentially prior to considering preconditions. A good planner will build possible actions from the preconditions rather than by enumerating the parameters but considering all of the above, what was gained?

In JPDL, the base syntax is Java, so the statement block exists within a method of a class. JPDL has many reserved identifiers to help make sense of Java syntax. Consider the JPDL definition below, all bold words are keywords in translation.

```
public class EffectSubclass : Effect
{
    StateVariableSubclass effectField1;
```

```

StateVariableSubclass effectField2;

public void apply(StateSubclass currentState)
{
    effectField1.fieldName=effectField2;
}
}

```

StateSubclass must be a defined subclass of State. The identifier of the parameter is used to kick off the identifier symbol table. The fields of the Effect are mapped to create a symbol table prior to translating the apply method.

So, before we begin translating the apply method, we already know we can start a translation such as this:

```

(:action EffectSubclass (:parameters      ?effectField1 – StateVariableSubclass
                                         ?effectField2 – StateVariable Subclass
                                         ...))
      :effects(...))

```

We also can start a symbol translation table. Each field of the Effect has an entry. It's worth noting that there's no way to translate numeric fields of Effects, Constraints, or Goals as there is simply nothing to bind them to in PDDL. For example, if you consider creating a predicate list that looks like (EffectSubclass-MyFieldNextValue ?e) what is e? It essentially is used to model a local variable. Even if we define an invocation object to represent e and treat these fields similar to local variables described later what would initialize these values? This is the value genesis problem manifesting again. So while the JPDL language independent semantics allow it the translator will not have a symbol for them when referenced and an error will occur. Here is a simplified symbol translation table.

JavaSymbol	Java Type	PDDL translation
effectField1	StateVariableSubclass	?effectField1
effectField2	StateVariableSubclass	?effectField2

Now we can begin to translate the apply method. The translator sees the basic assignment statement and needs a LHS and RHS identifier from the symbol table to output a translation. The LHS has a field accessor which is used to build a translation of the relevant predicate list from the type of the instance (StateVariableSubclass-fieldName ?effectField1 ?effectField1Target). Based on this we can determine the statement is a reference assignment

rather than a value assignment. The RHS is validated as being the same type of the LHS and existing in the symbol table. At this point the translation would look as follows.

```
(:action EffectSubClass
  :parameters      (?effectField1 – StateVariableSubclass
                   ?effectField2 – StateVariableSubclass
                   ?effectField1Target - StateVariableSubclass)
  :precondition    (StateVariableSubclass-fieldName ?effectField1 ?effectField1Target)
  :effect          (...))
```

Now that the LHS and RHS have been bound we can create our effect list as pertains to this statement and discussed above.

```
:effect (and      (not (StateVariableSubclass-fieldName ?effectField1 ?effectField1Target))
                  (StateVariableSubclass-fieldName ?effectField1 ?effectField2))
```

Voila, we have translated one trivial assignment in PDDL 3.0.

PDDL 3.1 has introduced object-fluents which allow a predicate list to return an object type rather than a Boolean or Numeric. This is an extremely useful addition which addresses the inefficiency of the above approach. Using this, the `?effectField1Target` variable could be removed from the parameters, and precondition line leaving a much simpler result. However, the JPDL translator predates this extension.

```
:effect((assign (StateVariableSubclass-fieldName ?effectField1) ?effectField2))
```

Statement Block

Statement blocks are complicated to translate because in PDDL all effects of an action are atomic. There is no sequencing within an action. Procedural code must either be broken up into multiple actions or translated to a set of formulas based on the parameters. A reasonable translation will require both approaches. The former approach will be discussed later, as it is eventually necessary when dealing with loops. The latter approach is valid within a single statement block. To accomplish this, a symbol table is built and stored that holds the formula of every symbol in the method as it is parsed. Assignment statements update this symbol table as discussed above. Each time symbols are encountered in an expression, substitution is done in translation with the formula. The output of a statement block translation is the set of changes to

the predicate lists and the logic that govern the preconditions of those changes. Consider the following statement block:

```
{
    A=B;
    B=C;
    C=A;
}
```

If identifiers A, B, and C are references that can each be mapped to some predicate list this statement block will produce six clauses in the effect clause of the translation. If A has no mapping to a predicate list, it still has an effect on translation, but the final value of the symbol is irrelevant and not translated. In addition to the symbol translation table mentioned previously which maps JPDL input parameters to PDDL expressions there is a symbol formula table for identifiers in the statement block that contains an expression tree for each of the current identifiers in terms of the original parameters. For the above statement block this table would start out looking very simple.

Original Identifier	Current Identifier
A	A
B	B
C	C

A few lines later, all of these identifiers have been remapped.

Original Identifier	Current Identifier
A	B
B	C
C	A

A different statement block might gradually build a formula rather than remapping everything.

```
{
    A=1;
    A++;
    A=A+A;
}
=>
:effect (assign (ListName ?A) (+ (+ 1 1) (+ 1 1)))
```

At the end of this statement block the symbol formula table would look like this.

Original Identifier	Current Identifier
A	(1+1)+(1+1)

This provides a reasonable translation of JPDL to PDDL. However there are a few eccentricities of PDDL which are hard to capture. The first relates to temporal action complications which haven't been discussed yet. The second relates to duplicate parameters.

Both cases are easier to demonstrate by using an eccentric fact of PDDL effects. A PDDL effect can simultaneously assert and delete a row from a predicate list. This is a difficult thing to capture using JPDL because no parallel distinction exists. All changes of a JPDL effect are atomic with no distinction between add and delete. For instance:

```
{
    A=false;
    A=true;
}
=>
:effect (ListName ?A)
```

The JPDL translation views the method as atomic and therefore ignores intermediate values in translation. This appears to produce the correct result regardless of the input value of A. At a glance that is very promising but complications arise under just slightly different circumstances.

First, consider the case where effect interactions occur. If there is a PDDL effect like this:

```
(:effect (and (ListName ?A) (not (ListName ?A))))
```

The delete may not appear to do anything meaningful. Regardless of the starting state of the predicate list, (ListName ?A) will exist after the effect. However, the official definition of PDDL effects is that all deletes happen before all adds. This becomes important when there are existing constraints prohibiting the delete of this predicate such as this.

```
(:durative-action MyTemporalAction
  :parameters (?A ?B)
  :duration (= ?duration 1)
  :condition (over all (ListName?A))
  :effect (OtherList ?A ?B)
)
```

When this durative action is part of a plan it prohibits another action from containing the previous effect in the duration of the durative action. It's not possible to capture this behavior using JPDL semantics without using multiple actions. At first, that may seem easy to ignore; it's a potential loss of expression in JPDL translation in that there are PDDL actions that have no JPDL

equivalent. However, if multiple parameters map to the same object this phenomenon rears its ugly head once again.

```

    {
        A=false;
        B=true;
    }
=>
:effect (and (not (ObjectType-Field ?A) (ObjectType-Field ?B))

```

During translation these appear to be different objects. During plan generation, ?A and ?B may actually refer to the same object. Because adds occur after deletes, the translation produces the correct behavior for the above case. If we reorder the operations, the problem becomes more evident.

```

    {
        B=true;
        A=false;
    }
=>
:effect (and (ObjectType-Field ?B) (not (ObjectType-Field ?A))

```

The translation is equivalent and correct when A and B are different objects. The translation is irreducible. However, the PDDL meaning of the translation is entirely wrong if ?A and ?B refer to the same object. When that happens, regardless of the input value this effect will result in an add **not** a delete in the final predicate list. The only way to avoid this pitfall is to break the statement block into multiple actions or add logic to the action governing what occurs. Here are two valid translations for the latter approach.

```

(:action MyAction
  :parameters (?A ?B)
  :precondition (not (= ?A ?B))
  :effect (and (ObjectType-Field ?B) (not (ObjectType-Field ?A))
)
(:action MyAction
  :parameters (?A ?B)
  :effect (and
    (when (not (= ?A ?B)) (and (ObjectType-Field ?B) (not (ObjectType-Field
      ?A))))
    (when (= ?A ?B) (not (ObjectType-Field ?A)))
)

```

Both of these translation approaches require a combinatorial number ($\binom{N}{2}$) of object comparisons to accurately translate the PDDL notion of deletes preceding adds. This is

problematic for scalable domains. For example, the data center domain has an object for each server in the update inlet temperatures action, if there are 20 servers, then a translation can require ${}_{20}C_2$ (190) different logical clauses. Furthermore, the simpler approach listed first is actually inaccurate in that it prohibits valid executions. The only alternative is to break the statement block into multiple actions. This is discussed in the loops section where it is unavoidable. As an approach it can certainly produce better results but comes with its own costs as well.

Variable Definition

Local variables are a curious case because they are only meaningful in procedural code while PDDL is orderless. The basic idea is to create an alias that is referred to in later statements. This can be facilitated using the symbol formula table introduced while covering statement blocks. Consider the following Swap Effect.

```
public class Swap : Effect
{
    SVSubclass var1;
    SVSubclass var2;

    public void apply(StateSubclass currentState)
    {
        int localVariable = var1.fieldName;
        var1.fieldName=var2.fieldName;
        var2.fieldName=localVariable;
        localVariable=var2;
    }
}
=>
(:action Swap
 :parameters (?var1 ?var2 – SVSubclass)
 :effect (and (assign (SVSubclass-fieldName ?var1) (SVSubclass-fieldName ?var2))
 (assign (SVSubclass-fieldName ?var2) (SVSubclass-fieldName ?var1)))
)
```

Note that swapping the Effect fields themselves is meaningless because until a field is accessed there is no binding to a predicate list. To accomplish anything we must change the value of the fields of a State Variable rather than the Effect. As with all swap methods, a local variable is required to do the swap in Java. The final value of that local variable at the end of the method is irrelevant to the translation because it has no binding to a predicate list and hence does not appear in the translation. Equally important, this is a great demonstration of orderless

operations in PDDL. The order of the assign statements is irrelevant because the formula for each assign statement is in terms of the input parameters and the change occurs simultaneously.

This paradigm works within a single statement block. When multiple statement blocks are linked together though, the value of local variables suddenly needs to be persisted. The only mechanism to persist information across actions is predicate lists. The necessary mechanisms for this linkage are covered in loops where using multiple actions is unavoidable.

Loops

Loops are difficult to translate because the control flow of the effect are a function of the parameters, and this function is dynamic in a way that cannot be formulaically contained in a static way. If the sum of a predicate list is needed then the formula to express the calculation of this sum is dependent on the number of rows in the predicate list. If the size of the predicate list were static we could hard code the formula for any particular number of rows into the action. However, the size of a predicate list can be dynamic within even a single plan. This makes it provably impossible to capture within a single action. Consider the for loop example below:

```
for(int i=1; i<5; i++)
{
    object.field+=i;
}
=>
(:effect (assign (objectType -field ?object) (+ 0 1 2 3 4)))
```

A translation is possible because the control flow of this loop is static.

```
for(int i=1; i<5; i+=object.field)
{
    object.field+=i;
}
```

The control flow is now dynamic; a translation is not possible within a single action. Here is the template for a standard for loop abstracted to a control flow view and its translation.

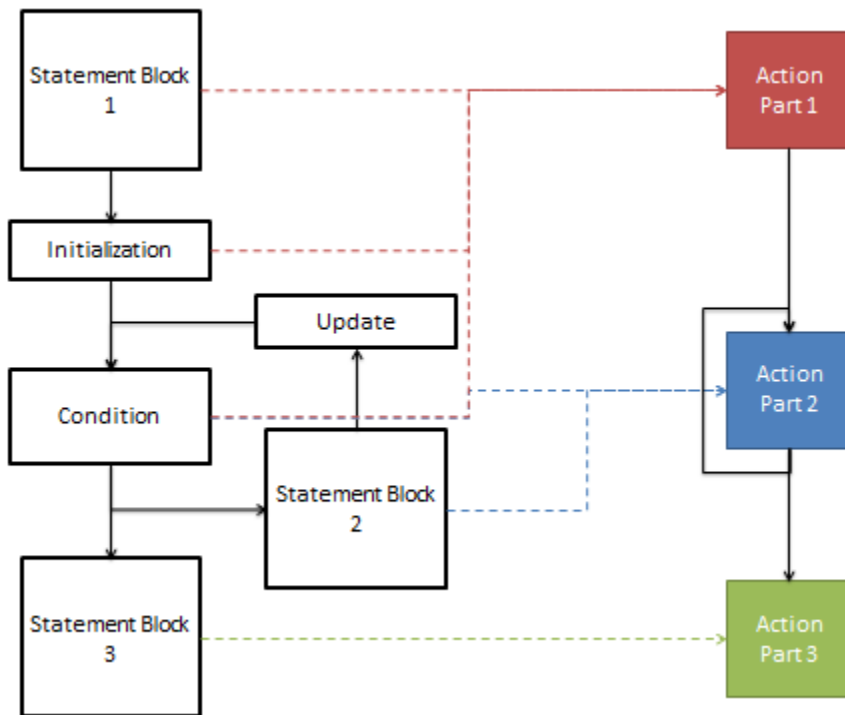


Figure 9: PDDL Translation Method of For Loops.

The BNF for this type of loop is shown informally below.

```
Statement Block1;
for(initialization; condition; update)
{
    Statement Block2;
}
Statement Block3;
```

Supporting dynamic control flow in PDDL requires translating a single Effect into multiple PDDL actions. Multi-part actions require some new machinery including a new predicate list and object concept. Logic needs to be inserted to ensure that all parts of an action are completed once a multi-part action has begun before other actions are considered.

An object needs to be defined to represent the invocation.

```
(:objects Invocation0)
```

A predicate list is created for all but the last part of the multi-part action.

```
(ActionName-Part1 ?Invocation)
```

(ActionName-Part2 ?Invocation)

An action is created for each part with logic that controls these predicates.

```
(action ActionName-Part1
  :parameters (...)
  :precondition (and (not (ActionName-Part1 Invocation0))
                    (not (ActionName-Part2 Invocation0))
                    ...))
  :effect (and (ActionName-Part1 Invocation0)
              (<Initialization Translation>)
              (when (not (<Condition Translation>)) (ActionName-Part2 Invocation0))
              <StatementBlock1 Translation>)
)
```

```
(action ActionName-Part2
  :parameters (...)
  :precondition (and (ActionName-Part1 Invocation0)
                    (not (ActionName-Part2 Invocation0)))
  :effect (and (when (not (<Condition Translation>)) (ActionName-Part2 Invocation0))
              <StatementBlock2 Translation>
              <Update Translation>)
)
```

```
(action ActionName-Part3
  :parameters (...)
  :precondition (and (ActionName-Part2 Invocation0)
                    ...))
  :effect (and (not (ActionName-Part1 Invocation0))
              (not (ActionName-Part2 Invocation0))
              <StatementBlock3 Translation>)
)
```

The above actions force there to only be 1 invocation in process at any given time. This implementation does not support recursion or function calling but it demonstrates the control logic concepts needed for multi-part actions. All other actions in the domain must now contain a precondition of (not (ActionName-Part1 Invocation 0)) in addition to their existing preconditions. This prevents other actions from beginning while parts of a multi-part action remain incomplete. Similarly, the goal statement of the domain needs to be appended so that no valid plans exist that leave an invocation incomplete.

Local variables that need to be available to more than 1 part of a multi-part action need to have their own predicate list to persist that information. Each part that makes a modification to the local variable makes an update to that predicate list. Other parts refer to this list in their symbol table during translation.

(ActionName-VariableName ?Invocation)

This causes an awkward amount of overhead to the action itself, which is bad enough, but it also has a global effect on the performance of every other action in the domain because of the precondition. Each time a part of a multi-part action completes the planner must prove to itself that the next part is the only available action and advance to the next part. From a planner perspective the difficulty of a proof scales with the number of predicates in the preconditions and effects of the actions of a plan as well as the depth of a plan. Multi-part actions contain an unavoidable overhead on all of these factors when translated to PDDL.

Once you establish the basic methodology for multi-part actions you can actually divide up many of the previous concepts using this idea. For instance an alternative approach to the symbol table translation method described in statement blocks would be to divide each sequential set that depends on a previous statement into a different part of a multi-part action. Function calls can similarly be implemented using multi-part actions. There are 2 philosophical approaches and an optimum that lies somewhere in between.

At one logical extreme each statement in JPDL can be used to create a different action in PDDL. The Part identifiers essentially become line numbers and the predicate lists containing Invocation0 essentially become a program counter. If the approach is modified to support multiple invocation variables and recursion then the predicate lists are being used to model stack memory. Used in this way PDDL could be used as a strange form of parallelizable assembly code. The control flow predicates in the effect clause are basically a jump operation. This approach has the benefit of being systemic and simple. A planner could in theory be optimized to work on domains designed in this way to overcome many of these difficulties. Something like a modified landmark planner could theoretically avoid most of the overhead costs by an intelligent treatment of different predicate types.

The downfall of this approach is that planners are not made to use PDDL in this way. This implementation becomes so far removed from the intent behind action design that heuristics are very likely to fail entirely or at least scale poorly. Each multi-part action in the domain adds overhead even if just to prove that it is unnecessary for a solution. Plans can become orders of

magnitudes deeper depending on the statements per Effect and the number of predicates to account for in a formal proof can double or more depending on the number of local variables.

The other logical extreme is to avoid multi-part actions almost exclusively. In general I favor this approach but it also has overhead. Consider the combinatorial cost described in the Statement Block section which arises as a result of attempting to model PDDL deletes preceding adds while avoiding multi-part actions. Or consider again how if-statements are translated using conditional effects.

```
if(A) S1;
else if (B) S2;
else if (C) S3;
else S4;
=>
(and  (when (A) (S1))
      (when (and (not (A)) (B)) (S2))
      (when (and (not (A)) (not (B)) (C)) (S3))
      (when (and (not (A)) (not (B)) (not (C)) (S4))
    )
```

Refusing to break this into multiple actions forces whatever Boolean expression is in A, B, and C to be repeated in the PDDL description. If a planner does not take the initiative to recognize repetitive expressions in an action it will end up reevaluating expression A up to 4 times when applying this effect. Without sequential operations the formulaic parallelizable version of an action will often find itself reevaluating expressions.

This cost also applies when translating for loops. The approach shown contains an expression for the initialization statement in its effect that Part 2 needs to function. That same expression may be needed to evaluate the condition to decide whether to enter the loop when Part 1 completes. Because the initialization effect won't have taken hold when the loop condition expression is evaluated the initialization expression will need to be repeated there. An alternative approach would create more parts which would more closely mirror the JPD L control flow. Which approach is better depends on the overhead of the planner, the complexity of the initialization statement, and whether the planner is reevaluating the expression or has detected and handled the repetition intelligently.

Function Call

There are two basic ways one could attempt to translate function calls to PDDL. The first is similar to macro translation in C. Each invocation of a function begins to process a new statement block using a subset of the symbol table of the parent. This works as long as the control flow is static over an entire problem. The alternative involves using a mechanism built on the multi-part action translation detailed in loops.

In multi-part actions we are using predicate lists for control flow, basically as a jump statement, by forcing the planner to conclude that only one action is applicable. Therefore the various parts of the multi-part action will all be executed in order by the planner. However, while all Effects return to the planner to decide on the next Effect, function calls can return to any Effect that invoked them. This necessitates some extra machinery. Embracing this jump ideology for a moment prior to adding functions any middle statement block of a multi-part action can be translated like this:

```
(action ActionName-PartId
  :parameters (?InvocationId ...)
  :precondition (and (JumpTarget JumpLabelObject ?InvocationId)) //This part is next
  :effect (and (not (JumpTarget JumpLabelObject ?InvocationId)) //This part is done
    <StatementBlock Translation>
    (JumpTarget NextJumpLabelObject ?InvocationId) //Do this next
  )
)
```

This is akin to having labels for each statement block after the entry point to an Effect, requiring each statement block to end with a jump statement, and using invocation objects to keep track of the local variables per invocation as a substitute for stack memory.

Invoking a function needs additional mechanics because a function can return to multiple points. Rather than having a complicated switch case in the function to decide where to jump to the statement block that does the invocation can tell the function where to jump to on return.

```

(action ActionName-PartId
  :parameters (?InvocationId ?FunctionInvocationId ...)
  :precondition (and (JumpTarget JumpLabelObject ?InvocationId) //This part is next
  :effect (and (not (JumpTarget JumpLabelObject ?InvocationId)) //This part is done
    <StatementBlock Translation>
    (JumpTarget FunctionLabelObject ?FunctionInvocationId) //Do this next
    (FunctionNameReturn
      ?FunctionInvocationId
      NextJumpLabelObject
      ?InvocationId
    ) //Return to there
  )
)

```

The function just needs to bind the appropriate objects to assert in the JumpTarget predicate list using the FunctionNameReturn predicate list with the matching invocation id.

```

(action FunctionName
  :parameters (?FunctionInvocationId ?NextJumpLabelObject ?ReturnInvocationId ...)
  :precondition (and (JumpTarget FunctionLabelObject ?FunctionInvocationId)
    (FunctionNameReturn
      ?FunctionInvocationId
      ?NextJumpLabelObject
      ?ReturnInvocationId
    )
  )
  :effect (and (not (JumpTarget FunctionLabelObject ?ReturnInvocationId))
    <StatementBlockTranslation>
    (JumpTarget ?NextJumpLabelObject ?ReturnInvocationId)
  )
)

```

Using one Invocation object per level of recursion allows recursion to a limited depth.

Reverse Translation

It wouldn't be fair to point to all of the things which are difficult to translate from JPDL to PDDL without also looking for things which are elegantly expressed in PDDL and difficult to translate to JPDL.

The most obvious are times where the multiple-key multiple-value dictionary aspect of predicate lists is exactly what you need for a domain. There is no Java library equivalent structure. Dictionaries are multiple-key single-value. There are open source libraries that provide this functionality but translating to JPDL using an open source library and then back would be infeasible if not impossible.

Additionally there are universal quantifiers which can be expressed using a single action in PDDL but become awkward multi-part actions when translated from JPDL loops. It is theoretically possible to detect these specific cases using foreach loops without break statements or perhaps using some special comment tags to serve as translation hints for these cases but in principle a forall statement builds a set of objects that meet a condition amongst all objects defined in the domain. In JPDL accessing all domain objects to accomplish this equivalently would require a special construct on the WorldModel that returns a list of all objects. The keyword identifier for this list could be reserved for the translator like the keywords Effect or State and used with foreach statements to produce a forall. Otherwise the translation to JPDL and then back would be extremely awkward.

Extensions to PDDL for various things would require an equivalent extension of JPDL semantics before a reverse translation would be possible. This applies to PPDDL semantics for probabilistic actions, PDDL+ calculus semantics, etc.

Remaining Concerns

In my JPDL publication I explain a case where ultimate result of Effects depends on their order of application. The solution in the publication is basically a buyer beware approach. The domain author needs to author effects in such a way that they are commutative. I have never been comfortable with this answer, but several years later I still have no better solution to this problem.

An issue which came up more recently is when to actually end the process. In particular, when all Goals are satisfied is it truly necessary to advance the World Model to the eventual time? The advancement algorithm in the original publication did so but I have changed my approach since then. The most compelling reason for this change is it provides greater flexibility with less confusion. If ensuring that the eventual time is reached before returning a plan is essential the domain author can add a Goal that is only satisfied when the next Effect set is empty. The other approach requires the Effect which will meet the Goals criteria to change the eventual time of the current State to force early termination. I feel the former is trivial while the latter is difficult and confusing so I changed the JPDL semantics.

Scheduling jobs to complete before deadlines is the planning Goal in the thermal data center domain. What would a reasonable eventual time be for this domain? Without knowing the plan, the only sensible answer I can think of is the last deadline amongst the jobs to schedule with the possible addition of some overhead time to ensure that the system is not left in a state where overheating is inevitable. But if the deadline is in 40 seconds and the jobs are all completed in 10 what benefit is there in advancing the timeline so far?

CHAPTER 4 – JPDL TRANSLATOR

CONCLUSIONS

Because of PDDL's simplicity, it is relatively easy to understand how a planner would utilize the action description and extract a sense of purpose from it. The only time a variable can be modified is in the effects clause. Every variable in the effect is potentially modified and the change is dependent on every variable in the :precondition clause.

Consider the action below:

```
(MyAction
:precondition (A)
:effect (B)
)
```

This action would be considered for use when B needs to be changed. The usefulness of this action from the planner's perspective is to change B. If A is not satisfied, the planner can add that to a want list of some kind and solve using a variety of heuristics.

At the ICAPS 2011 conference where JPDL was published I was told by several people that while they could easily understand how a plan could be verified by the approach I described, they found it difficult to believe that planning itself could be done on that representation. This section is my attempt at a proof that a JPDL solution is sufficient to be directly planned from in principle. In short, if any PDDL description of a problem is sufficient for planning and a JPDL description can be translated to an equivalent of the PDDL description, then that JPDL description is sufficient for planning in principle.

Within every planner that uses PDDL, the PDDL description is parsed into data structures that are integrated with the planning heuristic which that planner offers. The planning heuristic understands that data structure. This section has covered many Java syntactic elements and their equivalent PDDL translation. If a translation can exist from JPDL to PDDL, and PDDL is sufficient to build data structures for planning, then surely the information required to build these same data structures must exist in JPDL as well. If these data structures were described in terms of their PDDL semantic meaning and a PDDL semantic to JPDL semantic translation existed an automated means is theoretically possible to provide these native data structures to each planner

directly as long as their library exposed them. Much of the overhead introduced by translation could be avoided entirely if an appropriate data structure was constructed specifically for planning from JPDL. I would be curious to see how compatible JPDL derived data structures could be with the ideas behind existing heuristics.

When striving for arbitrary expressiveness that exceeds JPDL however, certain code constructs are not only problematic to translate, but also problematic to plan directly with. Take for example for statements where the loop iterator is modified in its statement block. It is extremely difficult to assess the usefulness of an Effect if the planner cannot even predict the control flow of that Effect. According to the famous halting problem the control flow of code becomes impossible to predict for a sufficiently complex function. Furthermore, arbitrarily complex code will inevitably need an import statement. This is when all premise of interpreting/translating planning approaches end.

For a given arbitrary function such as `Math.sqrt(double x)`, it is impossible to code a general understanding of arbitrary calls into a planner. One could certainly hard code understanding of this function into a planner, just as one could extend PDDL to add a new operator for powers. A certain subset of planners may then adopt that operator. However, this is a manual addition. The ability to use one function bears little resemblance on the ability to use the next. An arbitrary function can change numeric predicates to any value; even in the optimistic case where the function has no side effects on the parameters being passed in, the function could return practically anything. The closed world assumption many planners rely on is impossible to guarantee. Planner completeness is irrecoverably lost if an action can also have numeric parameters which are passed to the function.

Consider the following:

```
Public class MyAction extends Effect
{
    Double b;
    public void apply(State currentState)
    {
        ...
        currentState.a = myFunction(b);
        ...
    }
}
```



```
}
```

If the planner wants to change `currentState.a`, the planner knows this action can do so. The difficult question is, to what and under what conditions. For what desired `currentState.a` value should it consider this action for? What is the range of `b` that it should consider if it decides this action might be helpful? It can assign no bounds without first fully understanding the underlying function. When the function is imported, the only way to understand the function is having the context from the documentation. To the planner, the branching factor has just become infinite. If we wish to support arbitrarily complex domain specifications, unknown functions and numeric parameters are absolutely necessary. So what can be done? The only solution that preserves completeness is to add something to the domain specification to fill in the necessary information for planning.

Demonstrating this requires introducing the concept of value genesis. This can be summarized as assigning values to the parameters of actions that are not present on the State to which the Effect is going to be applied. Borrowing an example from our Blocksworld domain:

```
public class PutOn extends Effect
{
    Block x,y,z;
    ...
}
```

It seems intuitive that we don't intend for the planner to instantiate a new block for either `x`, `y`, or `z` but it's not hard to imagine actions where it does seem intuitive. One interpretation of the travel problem demonstrates this:

```
public class Travel extends Effect
{
    Vehicle p;
    Double x, y;
    ...
}
```

This action is simply responsible for moving a vehicle from its current location to a target location. It's intuitively clear that we should not generate vehicles for the Effect, but it's less clear where `x` and `y` should come from. Do we have a list of locations in the domain that contain `x` and `y`? Is there a list associated with each field that should be considered? Infinitely worse is if the

vehicle may need to travel to infinite locations. For example, if this action needs to refer to a location halfway between two existing locations. You can infinitely recurse to arrive at infinite locations. When the solution requires calling code for which usefulness cannot be ascertained or when there are literally infinite actions that could be generated we sacrifice completeness unless something else guarantees that the possibilities not considered can't provide a solution.

Adding this "something else" fundamentally reconsiders the responsibilities of a domain author and planner author. It shifts much of the responsibility to the domain author but in doing so it enables otherwise prohibitively complex domains to be solved. If a domain author is willing and able to provide the knowledge a planner needs but can't extract from the representational language then the representational language becomes almost irrelevant. The research question posed by this work is what additional information should a domain author need to provide to do effective planning? How can this information be minimized? What is the tradeoff between time investment by the domain author and the planning result? What data structures can be used for effective planning that were previously built from action descriptions and can a domain author provide them by an alternative means?

CHAPTER 5

MODELING LIBRARY SPECIFICATION

The original implementation of the data center thermal model was basically some line interpretation logic for the c curves and a series of arrays. This simulator just loaded values from configuration files and iterated time until the eventual time was reached. I suspect that most modeling software is similarly barebones, especially in academics. The problem with this approach is that the software lacks an integration point for anything other than modeling. When the model becomes promising enough to use, it can be unclear how to write a planner to use it.

JPDL addresses this problem by defining modeling semantics that structure domain modeling logic using meaningful planning semantics while still retaining the look and feel of a familiar programming language. However, JPDL is still focused on extracting familiar PDDL planning concepts; it just tries to make authoring domains easier. One limitation of this approach is that the planning-related structures built from a PDDL or JPDL domain are limited to the concepts PDDL can support. For example, predicate lists are the only data structure and all states are discreet. Planners rely on this fact and do proofs by reachability analysis on the predicate lists. They have no need to support variables continuously changing in a complex way. The language constructs don't even exist to express these complexities. Even PDDL+ which extends PDDL to provide continuous effects only allows specifying a rate of change for each predicate which remains constant until set by another action. If the complexity of your domain exceeds the expressive limits of PDDL, any planning-related structures you can find in existing planners will be limiting.

A more solution-oriented approach should be able to express any Turing complete model. A familiar way of achieving this is to express models in existing languages as code libraries. Any other approach that cannot at least invoke foreign code will add considerable overhead to domain authors. Defining a code library rather than a language offers several additional advantages:

- 1) Usability is improved because language documentation is extensive and already exists.
- 2) Just as with JPDL, existing IDE's can be used to create solutions.

- 3) The overhead cost of disk reading and parsing is removed from the planner. For many real-time planning domains, the planning problem can be directly built from what is already in memory rather than parsed.
- 4) If a reasonable API is provided, the solution process can be integrated with surrounding code. This enables using a debugger and persisting and utilizing data structures both into and out of the planning process.

A planning library solution should define reasonably general interfaces and structures that can be implemented by domain authors. These structures implement the JPDL planning and modeling semantics. The domain author implementation defines the planning heuristic and the data structure it acts on. In an object oriented language these translate to abstract classes and interfaces. Many independent solutions have been created amongst all PDDL planners, but because they are not public they vary wildly. There is no official or unofficial standard for either. One of the goals of this line of research is to offer a standard set of semantics which could be implemented in different languages and by different planners which would help to draw meaningful comparisons between different planning approaches. This would enable discourse such as comparing the overhead of modeling a domain for different heuristics rather than looking only at black-box criteria such as the runtime and memory characteristics of planners solving a problem.

CHAPTER 5 – MODELING LIBRARY SPECIFICATION

LANGUAGE INDEPENDENT SEMANTICS

Modeling semantics need to be defined for a planner to use regardless of language, API choices, or even the library versus language decision. The library has the responsibility of defining an API that maps to these modeling semantics. This section reviews the design decisions made in mapping JPDL modeling semantics to an object oriented paradigm and discusses alternative options and the tradeoffs considered. Here are the shorthand symbols from the JPDL language independent semantics that were covered in Section 4.

Symbol	Meaning
W	World Model
S_L	State list
E_L	pending Effect list
C_L	Constraint list
G_L	Goal list
S	State
V	State Variable set
P_v	Paired Identity and Primitive Set
E	Effect
t	time

Table 4: Modeling Library Symbols

A **State Variable** v consists of a string **identity** i , a paired **Identity and Primitive Set** P_v of value types, and a **State Variable** set V which allow for recursive hierarchies.

In an object oriented paradigm it makes sense to make a State Variable an Object. The paired identity and primitive set P_v could be a dictionary with a reserved identifier or if the set of identities is constant the identities could map to fields. To represent State Variables over time, either the State Variable Object must have multiple sets of field/value bindings with a way to find the appropriate set for the time bounds of a State or multiple instances of a State Variable must exist with an identity field for i to find a State Variable on a State with time bounds.

Representationally, I have found the latter to be more intuitive to address when writing domains. However, using class properties correctly it is possible to have a back-end representation that is different from the addressing look and feel. Using the latter approach, when a State Variable is accessed from a State it already has field bindings appropriate for that time. Comparing values of a State Variable over time requires accessing that State Variable from multiple States and

comparing their fields. This approach limits the flexibility of P_v on States and Effects but works well in general and the domain author can override the implementation of what State Variables are synchronized from a State to restore this generality as required or they can fetch the State Variables they need from the World Model.

While the language independent semantics define P_v as being a value type because I was avoiding the concept of reference types entirely, I see no reason to disallow reference types in an object oriented translation. This enables a field of P_v to refer to an array, object, etc. The only thing on the State Variable that is not part of P_v is the identity field, and any State Variable fields because those belong to V , not P_v .

A **State** S consists of a paired Identity and Primitive set P_v , a State Variable set V , and **time bounds** t_i and t_e for which these mappings hold.

If the State Variable to State decision from above is implemented, the most straightforward approach is to simply map P_v to fields of the State that are not State Variables and V to fields that are State Variables. The time bounds could be separate numeric values or a single time bound object. The opened/closed status of the initial and end time bounds are constrained by JPDL semantics which eliminates the need to represent this on the State.

A **State List** S_L is a time ordered set of states with continuous time bounds ranging from a beginning time t_i to an eventual time t^∞ . This provides a timeline of how values change over time. The current State of the State List is the state with $t_e = t^\infty$ and $t_i = \text{current time}$.

Any number of structures could be used for State Lists. Because time bounds are defined on the State an order could be established even if the encoding data structure is not inherently ordered. The only real consideration given to a data structure is to find one with an acceptable balance between expanding the size of the State List as the current time is advanced and accessing States for their State Variables as Effects are applied.

An **Effect** E is a function $f(t, P_v, V, W)$ that uses a paired Identity and Primitive set P_v , a set of State Variables V , and a World Model W with a State List with current time t , to produce a new World Model W' with modified State Variables on the current State, pending Effects,

constraints, and goals. A pending Effect is an Effect of a World Model where the time t of the Effect is greater than the current time of the State List of the World Model.

An Effect could be entirely contained within a function definition with some type of delegate with parameters for t , P_v , V , and W . However, if that were all an Effect is, whatever process builds Pending Effects would need to store sets of parameters so they could be invoked later as current time is advanced. This Pending Effect data structure would need to consist of some dictionary type structure of time, delegate, and parameters. The result is complicated and clunky. What is simpler and more consistent with the rest of the library is to encapsulate this method definition in an Effect Object so that the Object's fields can be used to store P_v and V rather than relying on the method signature. If the Effect object also stores a time value, then there is no need for a Pending Effect definition at all.

A **Constraint** C is a function $f(t_i, t_e, P_v, V, W)$, that uses a paired Identity and Primitive set P_v , a set of State Variables V , and a World Model W , to evaluate a whether a pattern contained within f is matched by W within the range $[t_i, t_e]$. If the pattern contained within f is not matched the WorldModel is inconsistent and the constraint has violated.

The only real difference between a Constraint and an Effect from an object oriented perspective is the time range in place of a single time. While this is a simple change it poses several questions in use. For example, a Constraint will need to be checked multiple times. The alternative would be to only check the Constraint when the timeline is finalized past its end time bound, which could lead to a massive amount of pointless work advancing the time bound if at least one part of the State List has been finalized which would violate the Constraint. While both Effects and Constraints receive a World Model as they are applied and checked respectively, the time bounds of a Constraint make it less obvious which State to synchronize V to. The logical alternatives are the start time of the Constraint, the State which has just been finalized, the current State, or not synchronizing at all. The most consistent option would be the current State, but Constraints shouldn't actually be validating information on the current State as it's subject to further change. The most convenient option for the domain author would be the finalized State but this definitely requires synchronizing which is costly. The least costly option is to no-op

synchronization. I have chosen the most convenient option for the domain author, but this is a suspect decision, and I could understand the value of having some type of information the constraint itself to delineate the type of synchronization to do. For example, subtypes of Constraint could all be synchronized in different ways, a flag of some type could determine the synchronization strategy, or an overloadable solution could conceivably be implemented.

A **Goal** G is a function $f(t_i, t_e, P_v, V, W)$, that uses a paired Identity and Primitive set P_v , and a World Model W , to evaluate a whether a pattern contained within f is matched by W within the range $[t_i, t_e]$. When the pattern contained within f is first matched by a State of W the goal is considered to be satisfied.

Where Constraints must not fail any time a State is finalized in its range, a Goal must succeed at least once when a State is finalized in its range. This doesn't affect anything from an object oriented perspective; it just requires that the planner controller is able to distinguish between them which can be done by having a separate class, by having distinct lists, or both. They should almost certainly either co-inherit a base class or one should inherit the other if there are functional additions for the planner controller. My implementation has Goal inherit from Constraint and stores Goals and Constraints on different lists on the World Model. This seems natural from a domain author perspective and is faster as it avoids needing to use reflection to distinguish between them.

A **World Model** W consists of a paired Identity and Primitive set P_v , a State list, a pending Effect list, a Constraint list, and a Goal list. It contains all information required to apply pending Effects until the current time of the State list matches the eventual time or all goals have been satisfied and no constraints have been violated.

P_v is somewhat special here in that it is intended to hold intransient information. If these values are changed by Effects it becomes impossible to audit Effects. To reflect this, it would make sense to finalize these fields for this object. However, since the domain author identifies the P_v fields, this is difficult to enforce. It's possible to finalize the entire WorldModel or to define a P_v dictionary to hold a collection of identities and their values but this is restrictive and inconsistent

with the rest of the library. Rather I chose a buyer beware approach where domain authors can only hurt themselves if they change P_v or static variables in their methods.

CHAPTER 5 – MODELING LIBRARY SPECIFICATION

MODELING DATA STRUCTURES

Below is a UML diagram of an object oriented set of classes which implement the domain independent semantics introduced at the start of this chapter.

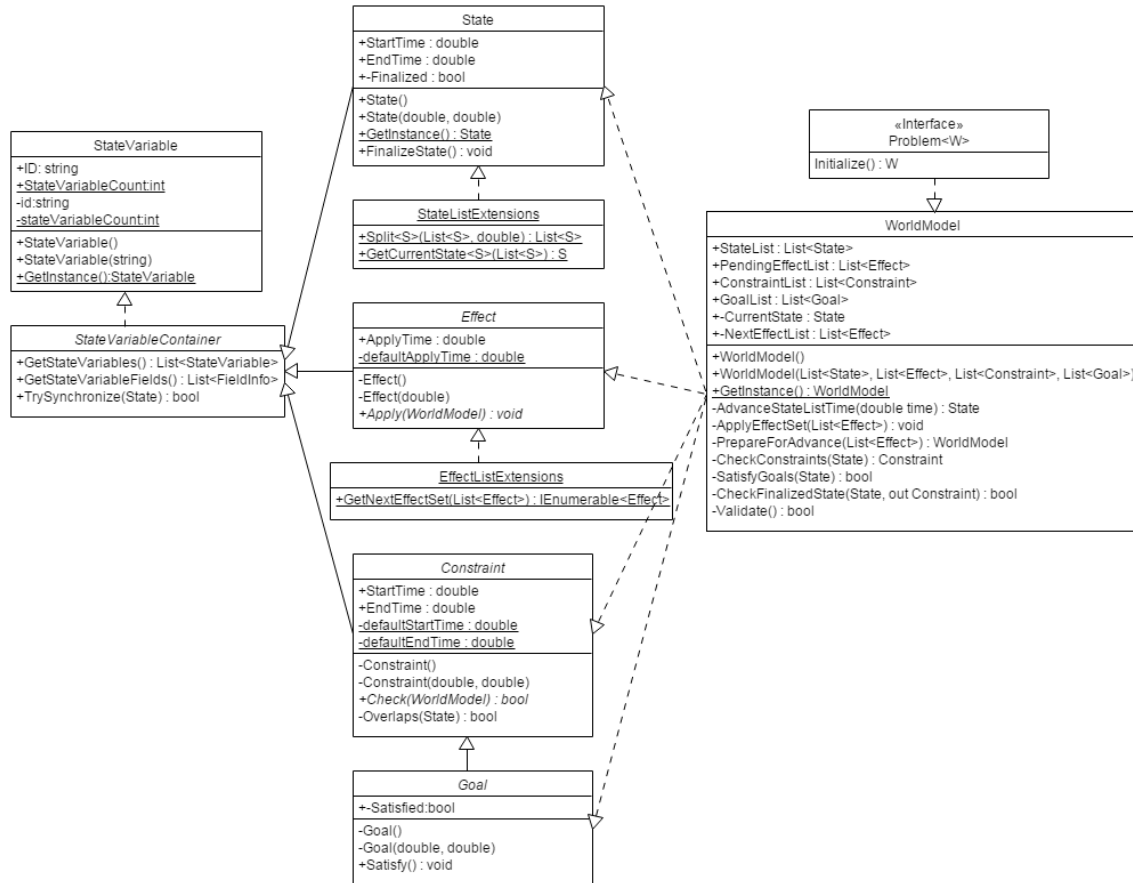


Figure 10: Modeling Library UML Overview

Several of the methods that may seem unfamiliar are actually the implementation for the World Model advancement algorithm from Chapter 4. Detailed msdn style documentation is available from the XML code comments for the entire planning library. Below is a brief description of some of the less immediately obvious methods.

StateVariableContainer contains several methods used for synchronization which can be overridden by domain authors. The default implementation of GetStateVariables is invoked on the State being synchronized to to retrieve a list of State Variables on fields, arrays, and enumerable collections which are considered for assignment to whatever is being synchronized to.

GetStateVariableFields is invoked on the Container being synchronized to get the public fields which hold the State Variables on the container. It ignores properties and any fields which are not a subclass of StateVariable or are private. TrySynchronize uses the fields returned by the latter and compares each object from the former against the list. When an id match is found, the field of the latter is assigned to the former. As a result the apply or check method invoked later doesn't have to sift through the World Model to find the appropriate object to check. However, this comes with the usual costs associated with using reflection. If these costs are a concern the method can be overridden to manually synchronize and ignore all of the reflection code.

StateListExtensions is a static class that contains extension methods for lists of State. Split is formally covered in the World Model advancement algorithm. The main point of interest is that split deep clones the current state.

All deep cloning in this library is accomplished by the DeepCloneExtension library in a different dll. This library provides the necessary flexibility to deep clone very simply with a DeepClone extension method with no prior overhead. Each object encountered is cloned depending on whether it implements the ICloneable interface or has an override behavior specified in the deep clone call. Cloning is a very complex topic which varies by programming language with a wide variety of approaches. I have invested significant effort adding flexibility and speed to the cloning process as this is one of the most expensive overheads of using the planning library. In particular, the fields to clone of a type are only gathered by reflection once and this behavior can be overloaded by type, the ICloneable interface can be implemented for the highest possible speed trading off programmer cost, a dictionary of replacement objects can be used to selectively deep clone only parts of an object web, and replacements objects can be specified for instantiation if the parameter-less constructor would cause problems or slowdown. The process even deep clones private fields all the way up the object hierarchy, it supports generics, arrays, every base type. And while I am rather proud of the result, it is possible though not convenient to change the deep clone process to use another library if the result is inadequate for some reason.

EffectListExtensions is another static class that helps with World Model advancement. When advancing the World Model, all Effects at the lowest time step are applied in a single algorithmic step. These Effects are called the next Effect set and the extension method creates this set from the pending Effect list.

Finally, World Model contains numerous helper methods to facilitate advancement. CheckFinalizedState calls CheckConstraints and SatisfyGoals after a new State is finalized in the World Model advancement algorithm. AdvanceStateListTime splits the StateList if the time provided from the next Effect set is ahead of the start time of the current State and returns the finalized State for CheckFinalizedState. Validate basically finalizes the State List up to the current State during problem loading. Several more methods exist to help the PlanningController covered in Section 6.

Example snippets of the BlocksWorld domain are available in the appendix. The full domain example of the data center and BlocksWorld are available alongside the planning library.

Remaining Concerns

In response to my frustration with using PDDL this library was designed with an emphasis on usability. Examples of this are prevalent in its design.

- 1) The concept of synchronization prevents the Effect method from needing to dig through the State to find State Variables that need changing.
- 2) The identity naming convention implementation of State Variables makes it easy to look at and verify plans without having to specifically name each object.
- 3) Every class in the library exists deliberately and has a meaningful semantic importance to planning. Clever uses of .Net libraries and the implementation of extension methods have reduced the number of data structures unique to the library.

I am uneasy whenever tradeoffs were made for usability that significantly impact performance or memory because ultimately the reason PDDL failed for the domains I encountered were unsurmountable performance and memory limitations. The usability difficulties I experienced were

a secondary concern even though in practice they were more frustrating and time consuming.

The chief performance concerns that remains are

- 1) How the library behaves when Effect descriptions are Markovian and
- 2) How State Variables are bound to States and used by Effects.

Addressing the first point, producing a timeline necessarily requires storing information about the WorldModel at previous times. If all Effects only address the current State then the timeline aspect adds nothing for the domain author but it still consumes significant memory and adds complexity to the cloning process. Different subclasses of WorldModel could be made which hold State Lists versus the current State. This would require substantial refactoring of the planning library and perhaps defining an entirely new branch of language independent semantics but it would help optimize Markovian domains significantly.

Addressing the second point, State Variables are bound to States the way they are so that a synchronized Effect can directly change a State Variable and the change will be local to the current State. This allows Effect apply methods to be extremely brief.

```
StateVariableSubclass myStateVariable;  
double v;  
  
public void Apply(WorldModel worldModel)  
{  
    myStateVariable.i1=v;  
}
```

No programmer effort is required to ensure that myStateVariable points to the State Variable instance whose values hold at the time of the desired change. No effort is required to ensure that changes made to the field will be persisted and visible to other Effects, Constraints, or Goals. Furthermore, the JPDL semantics ensure that changing myStateVariable directly causes no inadvertent changes to its value at previous times. The apply method has the luxury of assuming its fields and the State look like this:

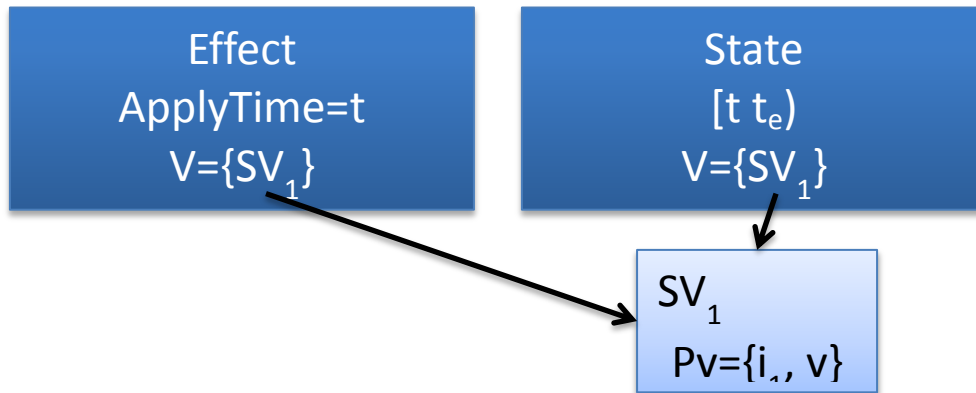


Figure 11: Shared Pointer Structure between State and Effects

These assumptions are luxurious compared to having to find the State Variable on the World Model, modify its time bounds, and potentially deep clone it if its start time happens to be before the Effect apply-time to prevent changing past values unintentionally. These repetitive tasks are implied by the planning semantics and conveniently handled for the domain author. These luxuries are enabled by synchronization and splitting respectively. Both of these luxuries are computationally expensive. Synchronization is a library concept, not a JPDL semantic concept, which relies on .Net reflection. The library includes an overloadable implementation for synchronization to preserve flexibility while providing usability. The overloadable nature of synchronization makes me confident that if the default implementation and the assumptions it makes are poorly suited to a new domain, that domain can create a better solution for itself. More problematic and less flexible is how splitting works.

Consider the case where the State Variable of an Effect resides on a State that does not begin at the apply-time of the Effect. If the Effect changes the State Variable instance it is semantically changing the value of the State Variable before the current time which violates the finalization semantics. There are two classes of solutions to this which are discussed below.

The first class of solutions defensively clones the data prior to allowing changes. The problem vanishes as a matter of convention. However, unless what will be changed can be predicted at the time of cloning, everything will need to be cloned. Without such prediction the

solution is memory wasteful. Each State will have an independent copy of all State Variables, the majority of which may be identical.

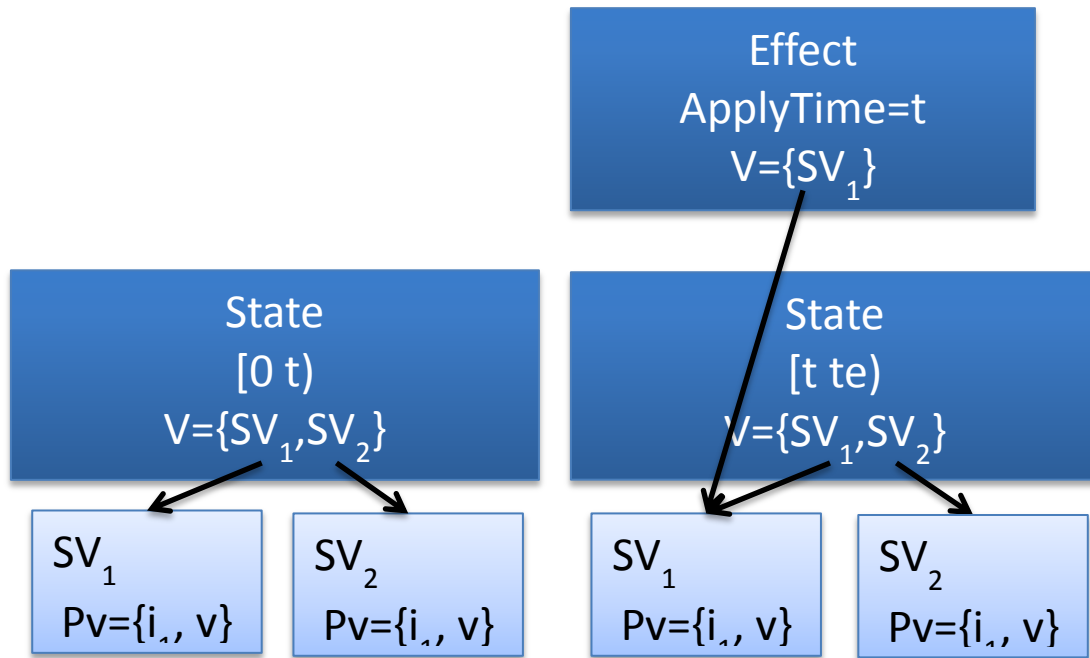


Figure 12: Memory Waste Caused by Defensive Cloning

There are several possible ways of implementing a prediction function. I offer several alternatives for thought. First, using .Net reflection it is possible to add Attributes to fields, methods, and classes with additional metadata. The most common example of this is in .Net unit testing. A basic test class looks something like this:

```
[TestClass]
public class TestClass
{
    [TestMethod]
    public void TestMethod()
    {
        ...
    }
}
```

The [TestClass] and [TestMethod] attributes are visible to the .Net unit test framework which is run on the resulting test dll. Similarly, the fields of the Effect could have Attributes added if the Effect plans to change that StateVariable.

```
[DeepCloneAttribute]
StateVariableSubclass myStateVariable;
double v;

public void Apply(WorldModel worldModel)
{
    myStateVariable.i1=v;
}
```

While there is some ambiguity, lists and arrays of StateVariables could similarly be attributed. Either the set itself would be deep cloned if it contained matching State Variables, or all State Variables in the set would be deep cloned.

A more direct approach is to add a method to Effects which would be executed prior to their application which builds the set of StateVariables on the current State which will need to be deep cloned. This would have a consistent look and feel with synchronization. This addresses the problem of interpreting how sets need to be cloned and even how nested State Variables need to be cloned but it requires more consistent effort from the user. In principle, just like with synchronization a default implementation could be provided by the library which domain authors could override. For example, an Attribute metadata approach could be the default implementation which domain authors could override to handle more nuanced cases like sets.

All approaches of this type become complicated if multiple Effects share the same apply time. If the State is cloned as part of splitting and this memory saving mechanism is included as part of the cloning process then the decision of what to clone needs to account not only for all Effects independently but also their potential interactions with one another. So, for example, if an Effect is intended to change every State Variable in a set and a State Variable added to the set by another Effect there would be no way of capturing this intent accurately at split time. Worse still is if complex Effects exist which will create new pending Effects at the same apply time. Such an Effect does violate the principle that Effects should be order-less within an apply time but I still think it is worthy of further consideration.

The inverse of this approach is possible as well, building a list of State Variables which would not be modified. The inverse approach simplifies cloning a bit but seems awkward from an Effect authoring perspective.

Regardless of the approach, the result of a change prediction function would be for V on each State to contain pointers which are potentially shared across State, reducing memory waste and the performance hit of deep cloning.

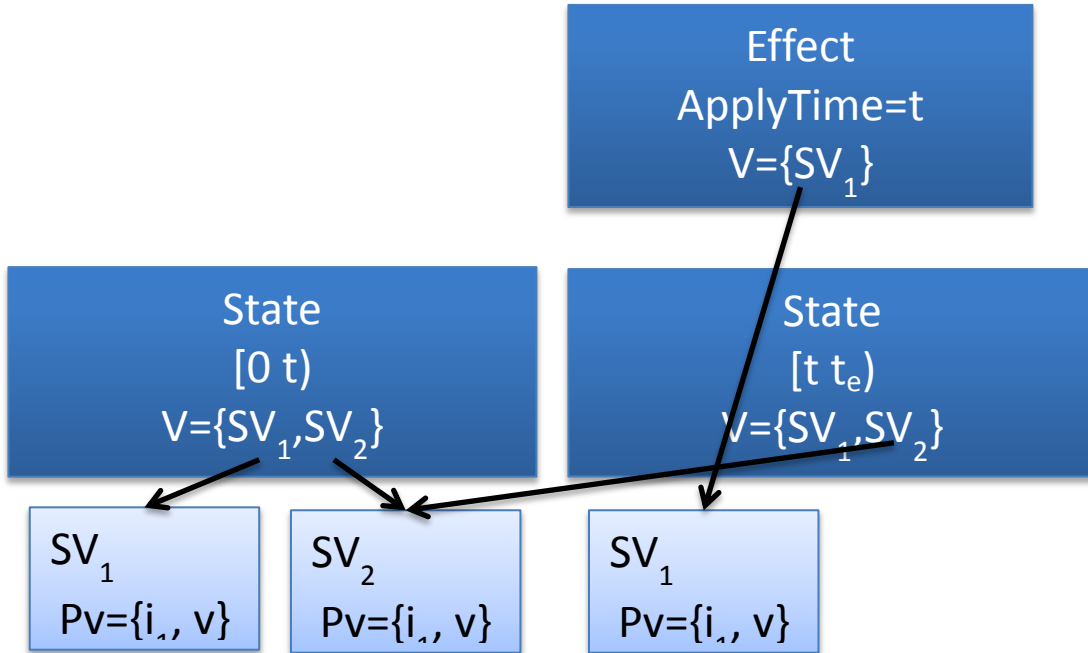


Figure 13: Alternative Partial Cloning Method

The second class of solutions reactively clones the data as changes are attempted. However, accomplishing this in a library solution is not trivial. The library would need to somehow limit the structures that the domain author can implement to ensure that the reactive clone occurs. For instance, States and State Variables could be made final. This would prevent changing references and value types on them and therefore any references past States hold could never violate the finalization semantics. The only way to make a change during an Effect would be to instantiate a new State to replace the old State with an updated set of State Variables.

A less disruptive method is to somehow disallow the default assignment operator on P_v and V of a State Variable and migrate the time bounds from the State to the State Variable definition. If all changes to a State Variable were gated by a library method then it could ensure the defensive clone is made. The function could have a signature like:

```
void SetValue(string identity, Object value)
```

I am skeptical of this approach for two reasons. First, I think domain authors may grow weary of always having to invoke a method rather than using the standard assignment operation. Second, it's unclear to me how to block it exactly. This is particularly true because the types actually being assigned to aren't necessarily State Variables, but are often just value types of fields of State Variables. How do you systemically prevent a domain author from assigning a double to a field of a class he himself authors? Even if you could, does that seem like an optimal solution?

In summary, the abstract design goal is to provide a programmer-friendly method for changing facts about the current world while ensuring that value changes are persisted, visible to other Effects, and don't break the finalized nature of the timeline. Organizing things in a way that values are persisted and visible is accomplished by sharing pointers and having a global store. Ensuring that only facts about the current world are changed is harder and often conflicts with the desire to be programmer-friendly.

CHAPTER 5 – MODELING LIBRARY SPECIFICATION

CONCLUSIONS

The simplicity of the JPDL semantics was not easy to achieve. Many alternative semantics were considered and eventually discarded for the simplicity and efficacy of this approach. For example, data was not originally structured onto States and State Variables but instead predicate lists were used with a PDDL-esq Markovian solution. I was advised that timeline style solutions had been tried and failed performance and memory benchmarks and it took serious analysis before I concluded it should be feasible. Once creating a timeline style solution, variable State time bounds were considered to generalize better than PDDL's awkward change mechanisms. Continuous, probabilistic, and hierarchical actions were considered and eventually discarded when I showed all of these could be solved with just the basic Effect definition. State Variables of various types were considered which would flag heuristics on how to use them before I gave up on trying writing just a planner and generalized to a planning library. In summary, it has taken several years to reduce the library to where it resides now. These semantics can be implemented in just about any language and can be used to model all domains from the international planning competition I have investigated as well as several cyber-physical domains I encountered which PDDL has been shown to fail at modeling.

CHAPTER 6

PLANNING LIBRARY SPECIFICATION

The previous section focused on the modeling portion of the planning library. Using the modeling classes outlined, domain modeling logic can be structured using meaningful planning semantics. Those semantics are meaningful to the planning controller portion of the planning library. The planning controller facilitates control flow of the planning process. The planning library provides the following to domain authors:

- 1) A structure for the solution space created by the planning process that can be reviewed or integrated into surrounding code.
- 2) A mechanism for deciding conditions of when additional planning is required.
- 3) An organization for managing and deciding between planning alternatives.
- 4) A mechanism for backtracking.

What this library does not offer to domain authors are the planning heuristics themselves. Due to the open-ended representational possibilities and the ability to invoke foreign code, general planning heuristics are impossible without the domain author specifying additional information. PDDL specifies “physics not advice”. By contrast, a planning library that cannot do code interpretation will need to be told both “physics and advice”. Because an Effect can be instantiated in infinite ways, additional information is required to specify which ways to consider. Because an Effect can potentially modify anything in the entire domain, additional information is required to specify the usefulness of an Effect. Because the heuristic data structure must be integrated with the heuristic process that uses it, the domain author must provide both, either custom to a specific domain or a means of extracting one in a general way for use with the other. I have considered offering a default heuristic as part of this library but deemed it too immature for publication at the time of this publication. Instead, the library includes an example domain and a heuristic which solves it. However, I have done diligence in considering that this library would be well suited to solving many of the domains from the international planning competition, the cyber-

physical systems from my background, and non-traditional planning domains such as constraint satisfaction problems.

CHAPTER 6 – PLANNING LIBRARY SPECIFICATION

LANGUAGE INDEPENDENT SEMANTICS

Planning semantics need to be defined independently of the library that implements them. The library has the responsibility of defining an API that maps to these semantics. This section reviews the design decisions made in mapping JPDL planning semantics to an object oriented paradigm and discusses alternative options and the tradeoffs considered.

Symbol	Meaning
P	Decision Epoch Pattern
C	Checkpoint
L	Checkpoint Link
T	Link Tree
B	Planning Branch
H	Planning Heuristic

Table 5: Planning Library Symbols

A **Decision Epoch Pattern** P is a function $f(P_v, V, W)$, that uses a paired Identity and Primitive set P_v , a set of State Variables V , and a World Model W , to evaluate a whether a pattern contained within f is matched by W . When this pattern is matched a decision epoch occurs as described in the Planning Branch algorithm below.

The basic structure is consistent with Constraints and Goals from the JPDL modeling semantics except that Decision Epoch Patterns have no time bounds. Because of this it is not immediately obvious when they should expire. Similarly it was not immediately obvious where they should reside. My final decision was for them to exist independently of the World Model on the Planning Branch and to expire when the next decision epoch occurs. This localized the checking process to a single Planning Branch rather than requiring the algorithm to navigate the Link Tree's parents. Alternative approaches include persisting the set of patterns from epoch to epoch or persisting each pattern until it's matched. Ultimately, any choice would serve because the parent Checkpoint Links are accessible during the decision epoch. Any of the above approaches are possible to implement by unioning whatever set of Decision Epoch Patterns the domain author wishes into the new planning branch he is building.

Another design decision was when to check the pattern. Goals, and Constraints are only checked after a State is finalized. Doing likewise here lead to a lot of stalling where there would

be no pending Effects remaining but the Goals were not yet Satisfied. This problem was encountered frequently enough that I decided to synchronize the patterns to the current State and check them after each Effect Set is applied.

At first it wasn't clear where the responsibility would lie to create the Decision Epoch Pattern. Previous research on temporal domains has concluded that deciding when to plan prior to deciding what to plan often led to a loss of planning completeness [65].

Because it proved troubling to separate the responsibilities and because it often made sense to consider when to next plan in terms of the heuristic model I integrated both decisions into a single data structure, the Checkpoint Link.

A **Checkpoint** C consists of a World Model W , a Decision Epoch Pattern P which triggered it, and domain specific heuristic information relating to the decision epoch. During the first decision epoch of the planning algorithm P is a reserved type of Decision Epoch Pattern. Otherwise, Checkpoints are created when P is matched on W .

The translation of the tuple to an object oriented approach is straightforward. However, the structure of the heuristic information varies from domain to domain and can't exist on the default implementation without somehow limiting the flexibility of that structure. As there is nothing to gain by limiting it, it's left open to the domain author. I expect a common addition to be some notion of prioritized reasons that the Goals are not satisfied. Storing the heuristic information separately from the World Model eliminates the need to clone it and is consistent with having a separate layer for planning versus modeling.

A* search can be implemented by storing a set of additional Planning Branches to try per Checkpoint alongside the perceived value of each Planning Branch. If this information was not available on the Checkpoint it would need to be stored on the Planning Heuristic itself or reprocessed each decision epoch.

Facilitating this custom heuristic structure required adding a factory because Checkpoints are constructed by the Planning Branch algorithm and not instantiated by the domain author. A domain author provided factory allows the planning controller to provide a Checkpoint to the heuristic that is immediately compatible with storing that heuristics information.

A **Checkpoint Link** L consists of a set of Effects E_L which represent planned actions and a set of Decision Epoch Patterns P_L which are used to start the next decision epoch. E_L enables advancing W by unioning E_L with the pending Effect List from W . Checkpoint Links are created by Planning Heuristics during decision epochs.

Checkpoint Links are the basis for describing what would traditionally be called a plan. Plans traditionally consist only of an ordered series of actions. However, actions alone are insufficient for the planning process without decision epoch triggers. This leads to several alternatives. The first alternative has the Checkpoint Link contain both the actions and these triggers. This approach is unintuitive because traditionally planning triggers are not part of the plan itself. The second alternative has the triggers exist on the planning branch rather than the Checkpoint Link. This would eliminate the Checkpoint Link concept entirely and make the Link Tree look like $\text{LinkTree}\langle\text{List}\langle\text{Effect}\rangle, \text{Checkpoint}\rangle$. The downside is that unless Planning Branches are stored as well the Decision Epoch Patterns would be impossible to audit. Lastly, the World Model could be restructured to include a list of pending Decision Epoch Patterns and simply have the Effects add Patterns as well. This would almost certainly coincide with other changes to Decision Epoch patterns. However it seemed odd to have Effects whose sole purpose was to aid the Planning process and would be meaningless to execute during plan validation, yet would still be part of the returned plan. Because a point of value for the library is the auditability of the solution space, I chose the first approach. This allows the heuristic to inspect the Decision Epoch Patterns of previous planning branches if that would be useful.

Originally Checkpoints and Checkpoint Links both contained a parent. This hierarchical information was eventually moved to the Link Tree structure which improved class cohesion and made the information more accessible to the planning controller which uses it.

A planning **Link Tree** T consists of a tree of Link Tree nodes each consisting of a Checkpoint value C , a set of children Link Tree nodes associated to a Checkpoint Link $\{L, T_C\}$, and a parent Checkpoint Link and Link Tree node which enables the construction of an ancestral ordered List of planning decisions that can construct a plan when all Goals are satisfied or allow

the planning heuristic to advance from other nodes in the tree. The Link Tree encodes the solution space both for the planning heuristic and for any surrounding code to analyze.

Link Trees are the most complicated structure of the implementation. They need to be able to navigate both upwards and downwards, and not only find their parent but efficiently find how the parent was linked to the child. Originally, I tried to associate the Link with the child as a field but sorting through the links required accessing each child and then its parent link field which was awkward. The implementing class inherits from Dictionary so an extra field is not needed for addressing the children. A Value field exists to address the Checkpoint. Multi-inheritance isn't allowed in C#, so the alternatives are to drop the inheritance and have a children field instead or to inherit from Checkpoint instead of Dictionary and remove the value field. All of the options above are about the same with some tradeoff of memory efficiency and generality for simplicity.

A **Planning Branch** B consists of the Link Tree node T to branch from and the Checkpoint Link L to branch with. This encapsulates the entire result of a decision epoch.

Originally, the heuristic returned a Checkpoint Link. Since the Checkpoint Link at the time held a pointer to a parent Checkpoint, the planning controller could advance the World Model of that Checkpoint. However, once the next decision epoch occurred, there was no easy way to update the Link Tree prior to invoking the heuristic. It required searching the tree for the node whose value matched the Checkpoint in question. Having the Checkpoint Link or Checkpoint contain a pointer back to the Link Tree is convoluted. A more elegant solution is to pass the heuristic the most relevant node to the last advancement and have it return the relevant node for the next advancement. Once that change was made there was no more need for the Checkpoint Link to contain a parent pointer.

A **Planning Heuristic** H is a set of functions $f_C(T)$ and $f_B(B, W, C)$ that return a Planning Branch to explore. $f_C(T)$ is a Checkpoint function that takes a Link Tree whose value is the Checkpoint created in response to a Decision Epoch Pattern being matched during World Model Advancement or initialization. $f_B(B, W, C)$ is a Backtrack function that takes the Planning Branch B which was explored, a World Model W which was advanced using B , and a Constraint C which was violated by W during World Model Advancement.

The heuristic itself is just an interface. The planning controller integrates the planning decisions from the heuristic into a general planning algorithm and maintains a complex control flow to advance the World Model safely, trigger decision epochs, maintain the Link Tree structure, and return a plan from the solution space when one is found.

All of the above capture the needs of a general planning process.

The planning controller orchestrates the generation of plans. A successful plan is a series of Effects to add from an Initial WorldModel to create a StateList which satisfies the Goal Constraints before an eventual time without violating any Constraints. The planning library acts as a controller that facilitates managing the search tree, constructing checkpoints, heuristic calls, applying Effects, and checking Constraints.

This approach is the most general solution I have conceived of so far. There is not a planning paradigm I am familiar with that could not be implemented into this framework. It allows for arbitrary definitions of decision epoch selection, planning heuristics, search heuristics, and goal criteria. The domain and the heuristics employed to solve it can be arbitrarily complex and can literally be used to solve any Turing complete predictive model.

CHAPTER 6 – PLANNING LIBRARY SPECIFICATION

PLANNING BRANCH ALGORITHM

This section defines how a planner can use these semantics to produce a plan. This algorithm uses symbols defined Chapter 5 and the previous section.

- 1) A World Model W is initialized by a problem P and used to create a Checkpoint C . Use C to seed an initial Link Tree T .
- 2) An initial decision epoch occurs. Set Planning Branch $B=H.f_C(T)$.
- 3) Initialize W to explore B . If B is null, fail.
 - a. Set $T=B.T$.
 - b. Set $W=$ a Deep clone of $T.C.W$.
 - c. Set $W.E_L=$ the union of $B.L.E_L$ and $W'.E_L$.
 - d. Select a next Effect set E_n with time t .
- 4) Explore B .
 - a. Advance W to t .
 - b. Apply E_n to W .
 - c. Check $L.P_L$ against W . If a pattern P is matched add a child Link Tree T' to T using a new Checkpoint built with W and P . Set $T=T'$. Set $B=f_C(T)$. Go to 3.
 - d. Select a next Effect set E_n with time t . If $E_n \neq \{\}$ Go to a.
 - e. Finalize current State.
- 5) Stalling has occurred. Set $B=H.f_B(T, W, C)$ where C is an implicit stall Constraint and go to Step 3.

When a State is Finalized:

- 1) Check Constraints in C_L that overlap S . If a Constraint is violated, set $B=H.f_B(B, W, C)$ and go to Step 3.
- 2) Satisfy Goals that overlap S . If all Goals in S_L are satisfied return a plan of Checkpoint Links built using T and L .

CHAPTER 6 – PLANNING LIBRARY SPECIFICATION

PLANNING DATA STRUCTURES

Figure 14 shows a UML diagram of an object oriented set of classes which implement the domain independent semantics introduced at the start of this chapter.

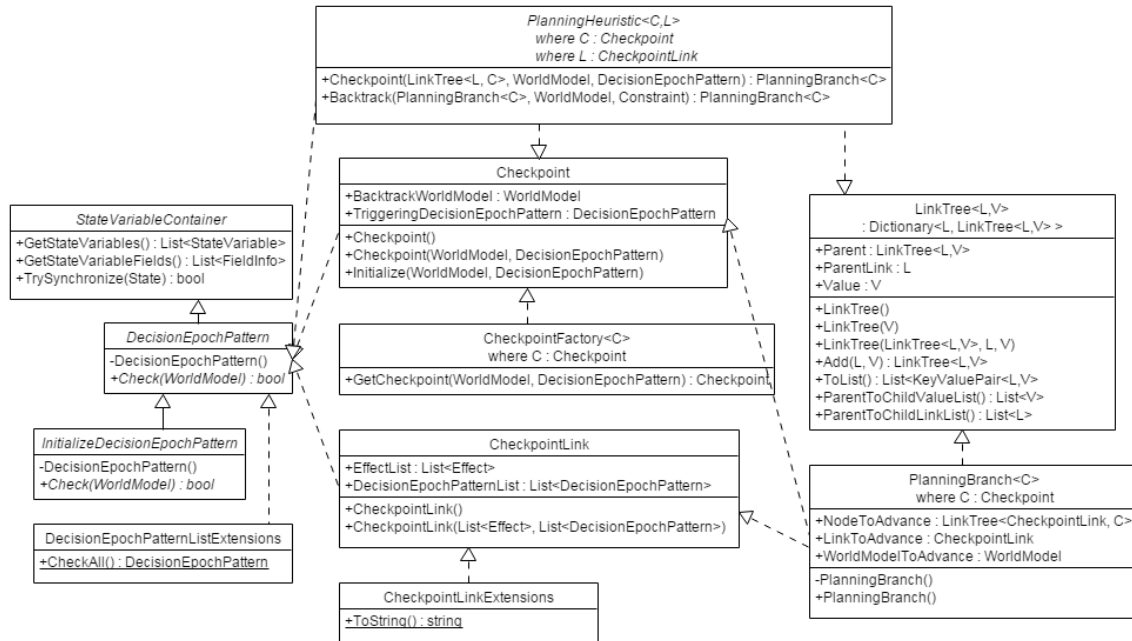


Figure 14: Planning Library UML Overview

This diagram is fairly complicated and even this large picture is incomplete so this will be explained one piece at a time in the same order as the semantics were defined.

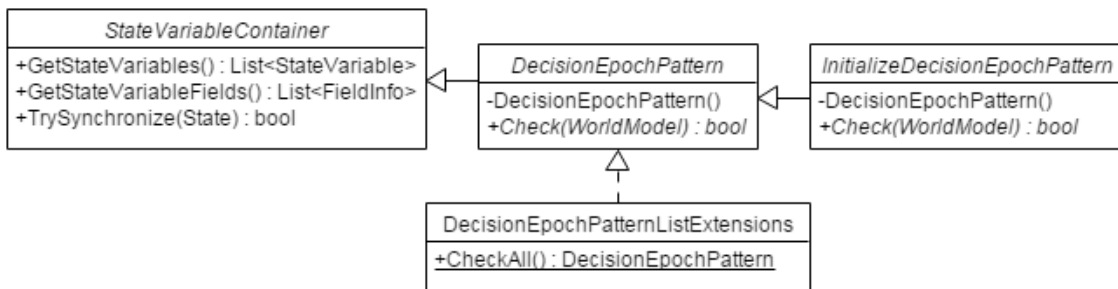


Figure 15: DecisionEpochPattern UML Diagram

DecisionEpochPatterns are constructed by the domain author as part of the PlanningBranch in the PlanningHeuristic. Each pattern is synchronized to the current State and checked each time a next Effect set is applied by the CheckAll extension method during step 4.c

of the PlanningBranch algorithm. DecisionEpochPattern inherits from StateVariableContainer to facilitate this synchronization and the functionality can be overridden by domain authors in the same way as Effects. If a match is found the Checkpoint function of the heuristic is called and the first matching pattern is passed as a parameter. The reserved type InitializeDecisionEpochPattern is passed during the first Decision Epoch.

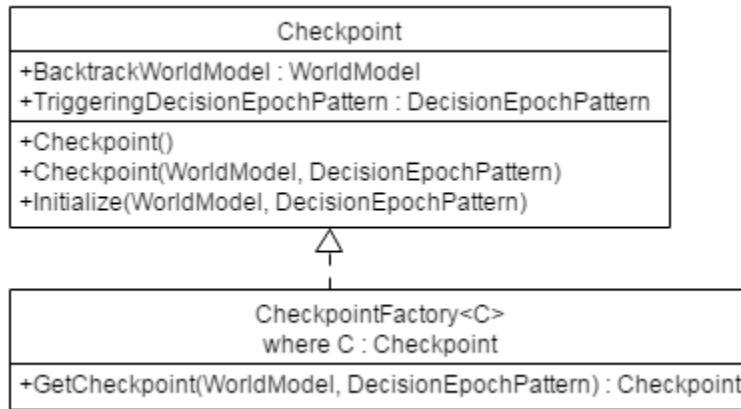


Figure 16: Checkpoint UML Diagram

Checkpoints are constructed by the PlanningController each time a DecisionEpochPattern is matched during step 2 and step 4.c of the PlanningBranch algorithm. The controller invokes the domain author provided factory and calls Initialize on the Checkpoint it creates. BacktrackWorldModel is used to prepare the WorldModel to advance each time a new PlanningBranch is being explored in step 3.b of the PlanningBranch algorithm.

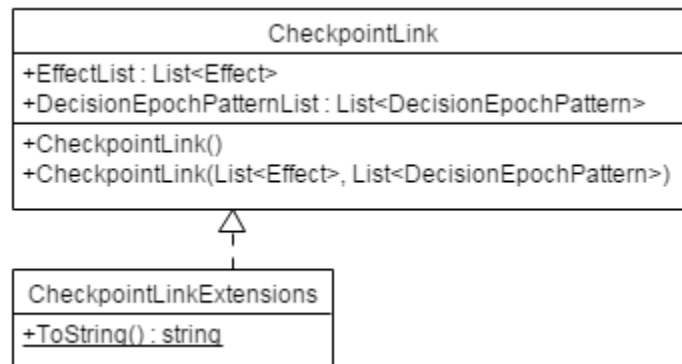


Figure 17: CheckpointLink UML Diagram

CheckpointLinks are created by the domain author as part of the PlanningBranch in the PlanningHeuristic during step 2 and 4.c of the PlanningBranch algorithm and step 1 of state finalization. The Effects of EffectList are appended to the PendingEffectList of the WorldModel to advance during step 3.c of the PlanningBranch algorithm. The DecisionEpochPatternList is checked each time a next Effect set is applied in step 4.c.

The ToString method is primarily intended to help with auditing. It returns a tab formatted string of the plan up to and including the CheckpointLink.

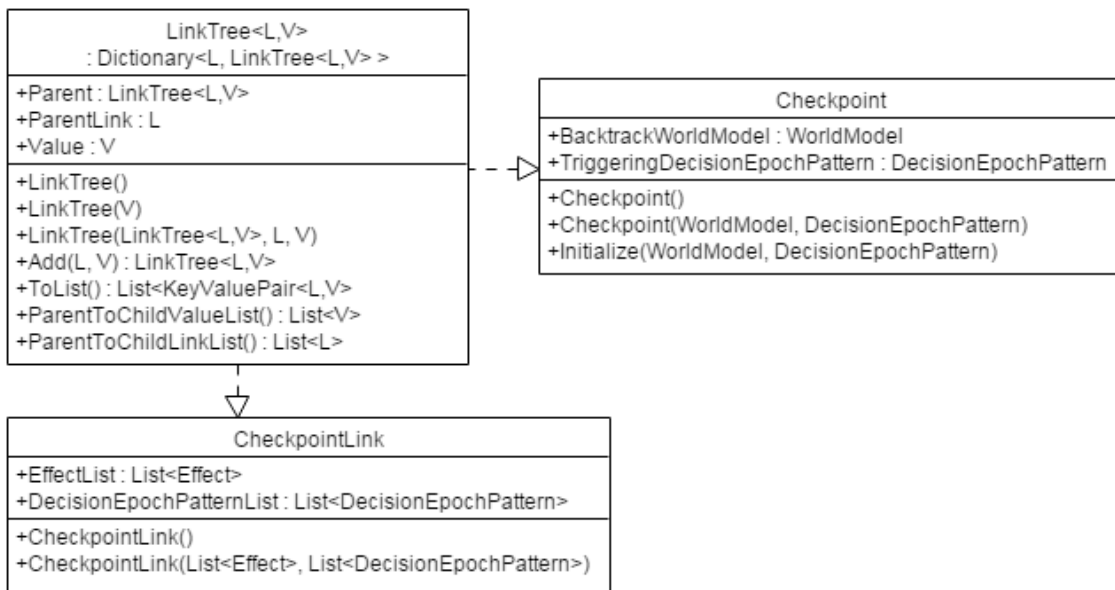


Figure 18: LinkTree UML Diagram

LinkTrees are created by the PlanningController prior to invoking the Checkpoint heuristic function during step 4.c of the PlanningBranch algorithm. The structure itself requires a bit of elaboration. Their purpose is to structure the solution space while providing strong class cohesion. Having a separate structure for the hierarchical information simplifies the data structures the heuristic builds and allows the planning library to handle the responsibility of managing the hierarchy for the domain author.

Without a LinkTree structure the hierarchical information would conceptually look like this:

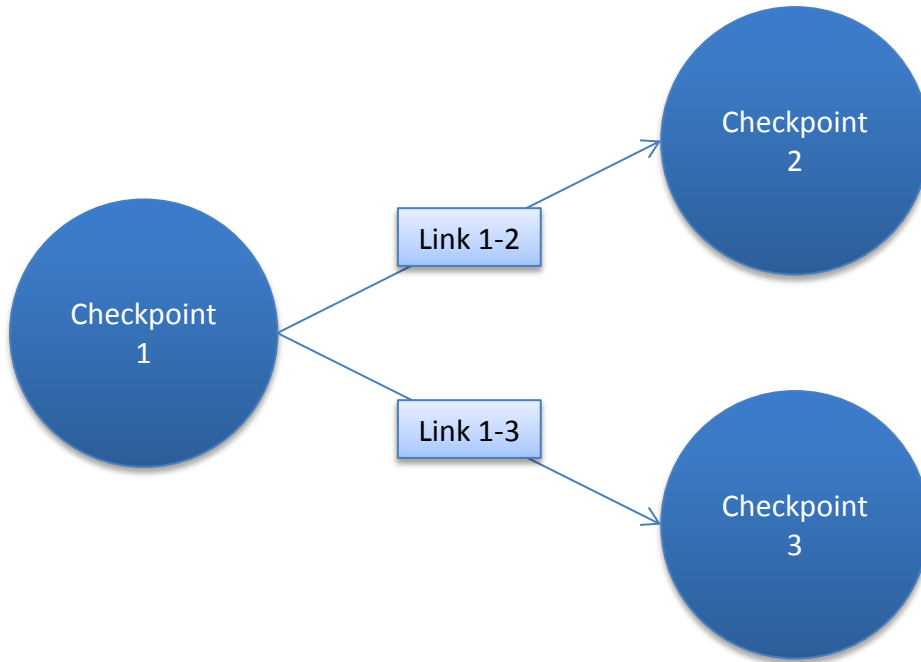


Figure 19: LinkTree Conceptual Diagram

This structure is similar to what would be used for A* search except that instead of each node being a State and the graph being a series of State transitions each node corresponds to a WorldModel finalized up to a particular time that progresses as you go down the tree and the difference from node to node can constitute many State transitions. Also, the association from one node to the next includes the planning decisions as Checkpoint Links which structures and publishes the solution space.

There are several things missing from this structure which motivate a LinkTree. First, there is no easy way of constructing and returning the plan once the Goals are satisfied. Either the entire plan up to each Checkpoint needs to be included in each Checkpoint directly or as an ancestral chain, or a pointer to the parent is required which can recursively construct the plan. Second, there are several possible implementations of this, most of which unnecessarily burden the domain author in some way. Does the Checkpoint point to the CheckpointLink or hold key value pairs of Checkpoint Links and Checkpoints? If the pointer to the parent is added, do you include a pointer to the parent CheckpointLink, the parent Checkpoint, or both? If you only include the Checkpoint Link, then the Checkpoint Link needs a pointer to the parent Checkpoint and

navigating up the tree requires a rather awkward `.ParentLink.ParentCheckpoint`-esq call. If you only point to the parent Checkpoint the child has to sort through the set of child CheckpointLinks of the parent to find a Checkpoint Link that points to or associated with itself in order to construct the plan. Including a parent pointer on the Checkpoint Link burdens the domain author since he is constructing the Checkpoint Link in the heuristic function. Having the Checkpoint Link point to the Checkpoint is poor practice because when constructing the Checkpoint Link the Checkpoint it should point to does not exist yet. Furthermore, a more cohesive solution places the hierarchical functionality in its own class for the same reasons we use generic Lists and Dictionaries. The LinkTree structure addresses all of these problems. It removes all of the hierarchy burden from the planning data structures and the burden of maintaining the hierarchy from the domain author.

Link Tree<CheckpointLink, Checkpoint>

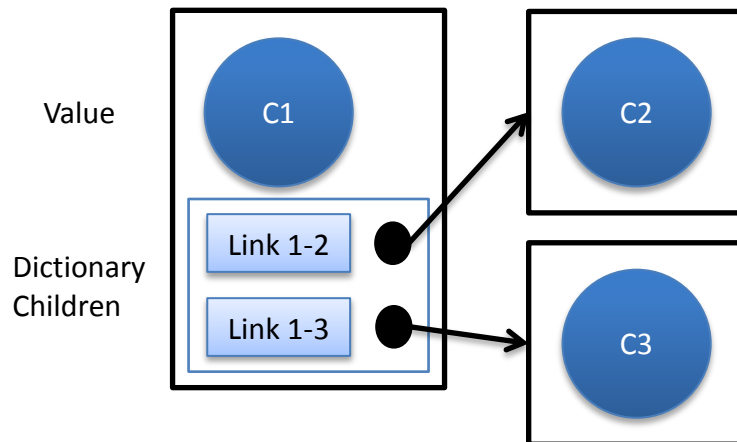


Figure 20: LinkTree Structural Diagram

It's easy to see how this structure would be compatible with other modeling work such as interactive model steering while also being an implementation of the planning semantics covered above [21]. While interactive model steering essentially replaces a planner with a human knowledgeable about a domain, integrating LinkTree solution space with a model steering approach would enable a planner to be part of the equation as well. A plausible application of this would be to use software such as WorldLines to debug and visualize a plan as it was being built in order to aid in the creation of a domain dependent planner.

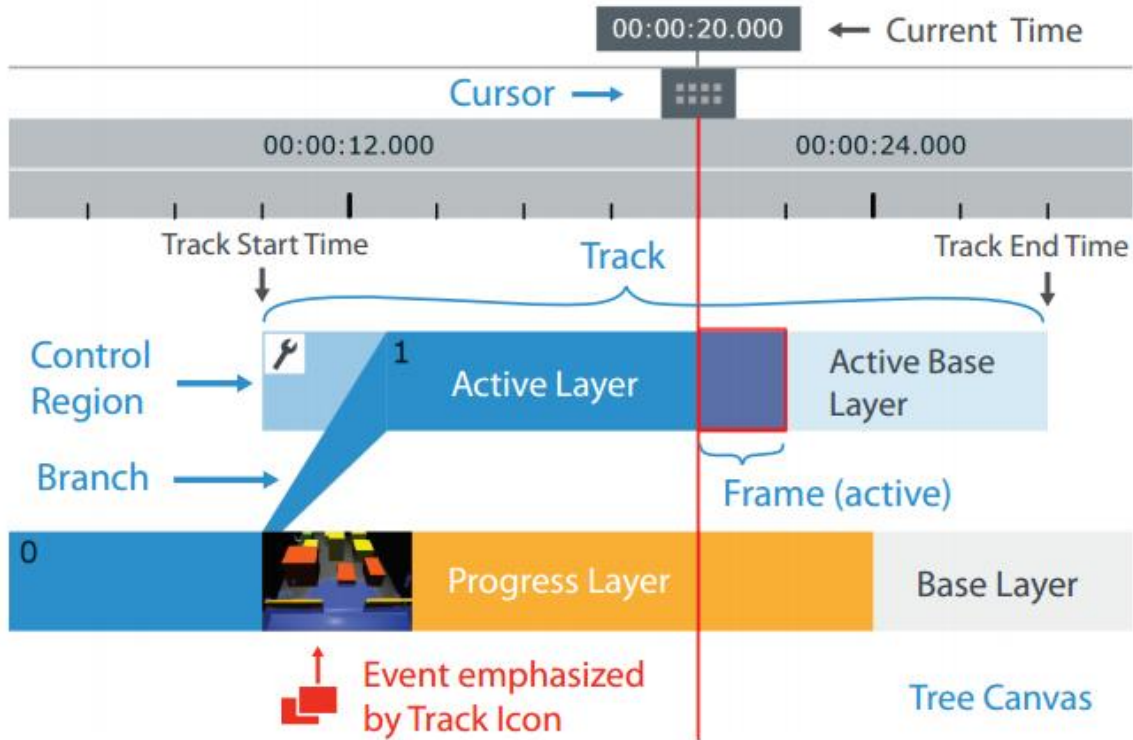


Figure 21: World Lines Visualization [21].

A reasonable parallel to the WorldLines model would include LinkTree at every place where branches occur and CheckpointLinks to connect tracks. The active track progresses as the WorldModel of the PlanningBranch is advanced. With a few modifications WorldModel advancement could be paused and tracks could be switched using interactive model steering. Potential branches could be viewed and selected between using the UI. It's even plausible to have a learning agent create a heuristic using this manual intervention over time.

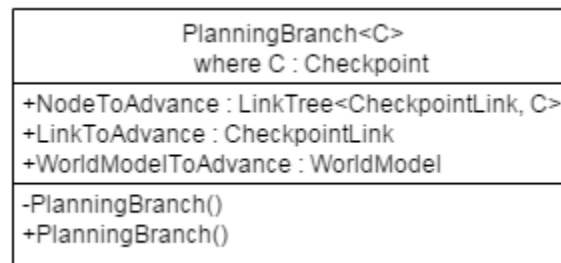


Figure 22: Planning Branch UML Diagram

PlanningBranches are created by the domain author in the PlanningHeuristic functions in step 2 and step 4.c of the PlanningBranch algorithm. They contain information about what WorldModel to advance from and what actions planning resulted in. The NodeToAdvance contains a Checkpoint with the WorldModel to clone prior to advancing. The LinkToAdvance contains the set of Effects to add to the WorldModel and the set of DecisionEpochPatterns to check while advancing. The WorldModelToAdvance property creates a backtrack-safe WorldModel from the NodeToAdvance with the Effects from LinkToAdvance appended to the pending Effect List. Once the next Checkpoint is reached, the resulting Checkpoint is added to the children of Node to advance prior to calling the PlanningHeuristic Checkpoint function.

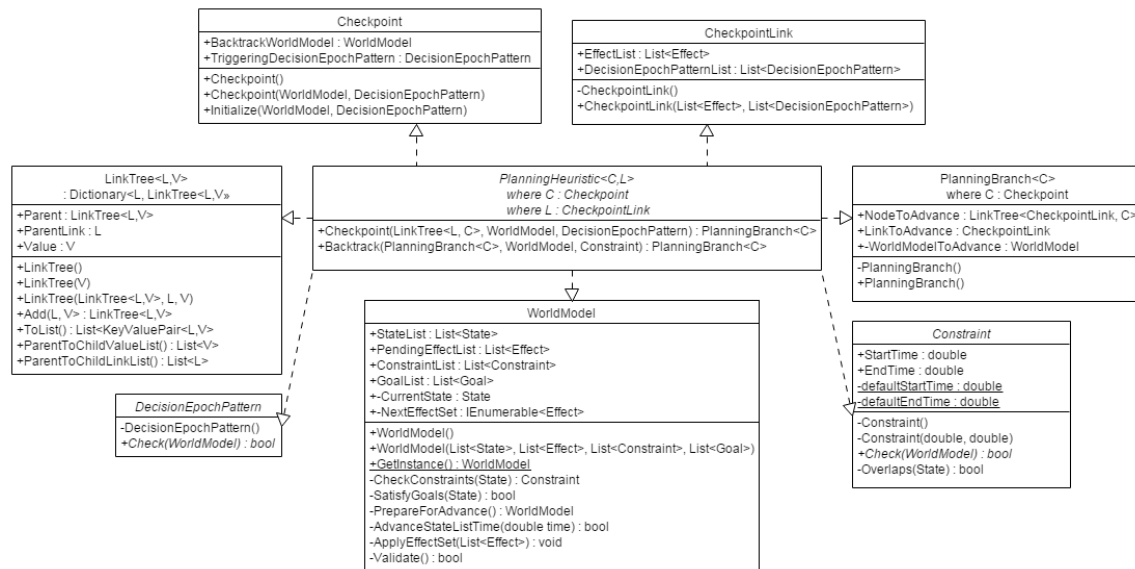


Figure 23: PlanningHeuristic UML Diagram

The PlanningHeuristic is a service object provided to the planning controller during setup. It consists of two functions used to handle four cases.

- 1) Initialization occurs in step 2 of the PlanningBranch algorithm. This calls Checkpoint with a reserved type of DecisionEpochPattern called InitializeDecisionEpochPattern, a root node LinkTree, and the WorldModel initialized from the problem provided to the PlanningController.
- 2) Standard checkpoints occur in step 4.c of the PlanningBranch algorithm. The function is passed the newly created LinkTree node whose value is the Checkpoint containing a

WorldModel on which the DecisionEpochPattern parameter matched. The WorldModel and DecisionEpochPattern parameters to Checkpoint are redundant programmer conveniences because they can be fetched from the LinkTree parameter.

- 3) Standard backtracking occurs when a Constraint is violated in step 1 of State finalization. The function is passed the PlanningBranch being expanded at the time which gives it access to the original WorldModel, the WorldModel at the time the Constraint was violated so that it can compare the difference, and the Constraint that was violated.
- 4) Stalling occurs in step 5 of the PlanningBranch algorithm when there are no more Effects to apply before the State List eventual time but at least one Goal remains unsatisfied. This calls the backtrack function with the usual parameters except for a reserved type of Constraint called PlanningStalledConstraint.

These functions could have been implemented by 4 methods instead of 2 to remove the need for the reserved types of DecisionEpochPattern and Constraint but in practice there seemed to be enough overlap between the code implementing the two functions to merge them as I have done.

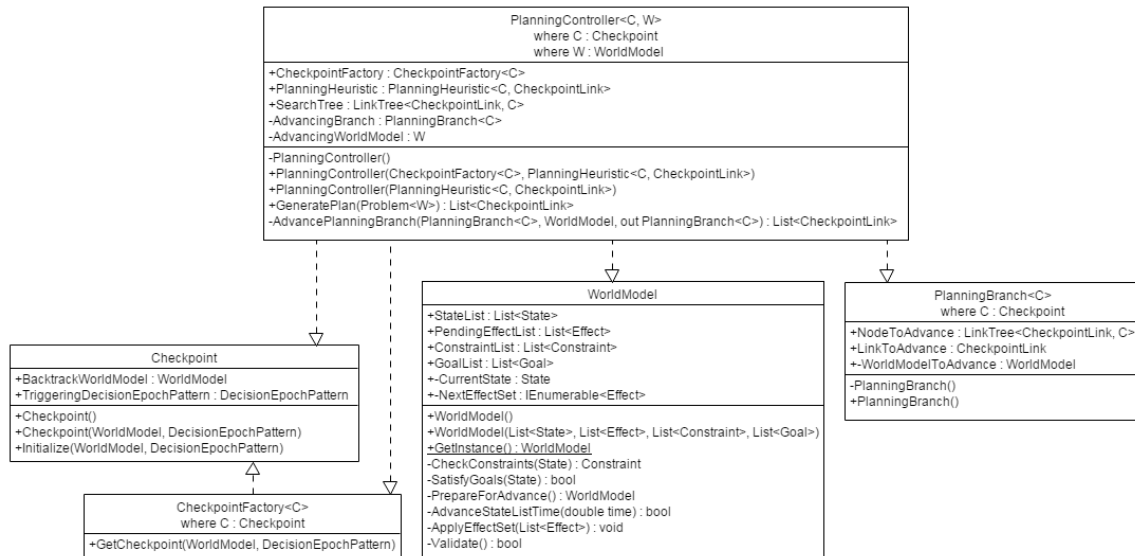


Figure 24: PlanningController UML Diagram

Finally, there is the PlanningController. This is not part of the language independent semantics but the part of the library that implements the PlanningBranch algorithm. The

constructor requires a heuristic and an optional checkpoint factory if the default checkpoint isn't used. The problem is passed to the GeneratePlan call on the controller instance and returns a List of CheckpointLinks which can be used to view the Effects added by the heuristic. The design allows domain authors to use the library with a minimum of fuss.

```
PDDLProblem problem = new PDDLProblem("PDDLProblems\\" + problemFile);
```

```
BlocksWorldCheckpointFactory factory = new BlocksWorldCheckpointFactory();
```

```
PlanningController<BlocksWorldCheckpoint, WorldModel> controller = new  
PlanningController<BlocksWorldCheckpoint, WorldModel>(factory, heuristic);
```

```
List<CheckpointLink> plan = controller.GeneratePlan(problem);
```

Remaining Concerns

There are several features that would be useful to domain authors and several remaining inefficiencies that could still be addressed.

When all Goals are satisfied there is no further heuristic call prior to returning. If the first plan found is not optimal there is no chance for the heuristic to keep searching. This works well if the heuristic is admissible but can be problematic otherwise. This can be worked around by detecting that Goals would be satisfied by the Effect set prior to State finalization but this violates intuitions about the function of Effects.

Also related to Goals, the only time Goals are checked is after a State is finalized or advancement stalls. In the former case this means that an unnecessary split (including the deep clone) already occurred. In the latter case this can mean either that a decision epoch unnecessarily occurs where all Goals could be satisfied on the current State or that the pending Effect List was emptied, even if doing so was artificial. From a modeling semantic perspective it does make sense to only satisfy Goals on finalized States to prevent the Goal from being immediately unsatisfied but in principle some mechanism could be added between applying the next Effect set and checking DecisionEpochPatterns to force the current State to finalize without stalling or advancing.

When backtracking occurs there is no LinkTree node that tracks the advancement of that PlanningBranch. To an auditing mechanism there is no sign that the backtrack ever occurred.

The heuristic has to keep track of the useful CheckpointLinks from the parent node and eliminate

the CheckpointLink that led to the Constraint. If it wants to make backtrack calls visible it needs to store that information on the Checkpoint itself somehow as well. This in and of itself is just some extra bookkeeping; more wasteful is that there can be any number of States and applied Effects between the last Checkpoint and when the Constraint was violated. Any portion of the advanced WorldModel that was not responsible for the eventual Constraint violation will need to be regenerated from the last Checkpoint by redundantly advancing. There is no mechanism to resume advancement from any point other than the last Checkpoint.

While significant effort has been put into making the defensive cloning process efficient there is an irreducible cost that scales with the size of the domain. Each time a State is split a deep clone of that State occurs. Each time a PlanningBranch is explored from a Checkpoint, the WorldModel is cloned again. Fortunately, the planning semantics ensure that only the current State needs to be deep cloned, but the WorldModel clone still needs to deep clone the current State, shallow clone the State List, deep clone the Effect List, Constraint List, Goal List and any variables that exist on the WorldModel object itself each time a Decision Epoch occurs. These costs could be reduced if the library had a programmer guarantee that an Effect, Constraint, or Goal would not be modified in the same way as finalized States. Another technically possible optimization would be for the WorldModel's StateList to fetch previous States from up the LinkTree rather than shallow cloning the StateList. The interface for interacting with the StateList would need to be revisited and parts of the PlanningLibrary would require significant rework to pull this off.

The modeling semantics make no guarantee about the order of Effect application. The library takes a "buyer-beware" approach similar to finalized States; don't write Effects that will be applied at the same time if their order of application produces different State Lists. Doing so means that a different implementation of the WorldModel advancement algorithm would determine a plan to fail.

Decision Epochs add an ordering to Effects which doesn't exist during plan validation. This makes it possible for successful plans to be generated which will fail validation.

In many planners, the problem of choosing decision epochs is solved independently of solving for actions. William Cushing convincingly proves the inevitable problem with these approaches; the context of the actions is necessary to ensure planning completeness [65].

It is the domain author's responsibility to account for the context of all domain Effects when specifying the decision epoch patterns. Failing to do so can result in a loss of planning completeness. A decision epoch pattern is specified as a Boolean method on a DecisionEpochPattern interface. The planning heuristic method which generates Checkpoint Links specifies a set of patterns which are used until the next decision epoch. A few common patterns are included in the planning heuristic library such as triggering when the StateList has been finalized up to a predetermined time or when a constraint associated only with the DecisionEpochTrigger is violated by the WorldModel.

CHAPTER 6 – PLANNING LIBRARY SPECIFICATION

CONCLUSIONS

Refining the planning semantics to their current level of simplicity was no easy feat. My first attempt at the planning solution included no tree for the solution space. The planning heuristic returned a time for the next heuristic call and the Effects to append. Either the attempted solution would succeed adequately and progress or violate a Constraint and fail. This was obviously insufficiently general.

My second attempt at the planning semantics included a tree for the solution space consisting of Checkpoints and what became Checkpoint Links. DecisionEpochPatterns generalized the notion of when to plan from a time to a pattern. The domain author now also had to provide a Checkpoint selector mechanism. Each Checkpoint was also associated with an enumerable set of Checkpoint Links which was created at the time when the Checkpoint was constructed. The Checkpoint selector reduced the solution space to a single node and that node output the Checkpoint Link to expand with. This had the significant drawback that all Checkpoint Links had to be inferable at the time when the Checkpoint was constructed. It was difficult to learn from a Constraint violation and update your Checkpoint Links. Also, the ranking mechanism for Checkpoint Links was unclear. Presumably the Checkpoint Link enumerator would be able to use the WorldModel to rank its own Checkpoint Links but the Checkpoint selector would have to do this ranking again amongst the Links from each enumerator. I could imagine solutions where the Checkpoint Link, Checkpoint pair was reduced to some A* value and then the optimal value was chosen but the whole process felt repetitive and clunky.

This led to a model where the decision of where to expand from and what to do were packaged into a single concept called a PlanningBranch. This prevented the domain author from needing to analyze each Checkpoint for an optimal CheckpointLink and opened up simpler approaches such as breadth first search or depth first search which can select a Checkpoint not based on a Checkpoint Link but it's position in the solution space. It also removed the seemingly repetitive definition of ranking Checkpoint Links within and amongst Checkpoints.

At this point the library has reached a balance that doesn't constrain the form of the solution any more than is necessary to provide a structure for the solution space and implement a general planning algorithm on behalf of domain authors.

One of the more substantial complaints I received to my original JPDL publication was that while the reviewer could understand how to validate a plan using my the WorldModel advancement algorithm, he could not understand how to make a plan from the domain definition. My answer at the time was to prove that JPDL was translatable to PDDL, and therefore could be used to build the same planning structures that heuristics currently use. I have kept this feedback in mind when changing to a planning library that can support arbitrarily expressive implementations that are beyond the reach of code interpretation techniques. Are the definitions provided sufficient to author a domain independent heuristic?

The answer is certainly no, but by how much? Because it cannot be built directly from the code, whatever structure a general heuristic acts on needs to be built using metadata or by functions the domain author provides. These heuristic structures invariably act on limited models, though the limits of each heuristic may vary. However, there is nothing in the semantics provided that prevents the construction of a heuristic structure. For example, a subclass of Goal could be authored which specifies a conjunctive normal form of small Boolean patterns each of which is mapped to one or more State Variables. The types of these patterns could be understood by a heuristic which knows how subclasses of Effects can be used to satisfy them. Metadata on each P_v and V element of a State or Effect could be used to extract reachability information for these variable. A reverse function associated with each apply method could be used to do backward-chaining. Each of these pieces would substitute something that general heuristics already extract from a PDDL description until the entire heuristic structure has been equivalently replaced.

The forms that these heuristic structure replacements could take are very interesting to me and I think would be of great interest to the planning community. The underlying semantics that the planning library provides would also provide a lexicon for discourse. Analyzing why domains exceed these descriptions will be much easier and more productive when that domain is implemented using a well understood semantic base. This could in turn motivate the expansion of

PDDL to support more complex domains. I don't think that this approach is necessarily superior to the research approach, but rather it is necessary to solve some domains, and largely unexplored to date.

PDDL is ill-suited to represent entire categories of domains, many of which are quite common and for which better solutions are urgently needed. Spatial-temporal domains are one such class which commonly applies to both Cyber-Physical Systems and Video Games [66].

CHAPTER 7

BLOCKS WORLD EXAMPLE DOMAIN

In order to illustrate how a domain author interacts with the planning library detailed in Sections 5 and 6 and in order to evaluate the performance of the library objectively I have implemented a solution for the Blocks World domain from the planning literature and the planning competition.

The general form of this solution is applicable to many domains. The underlying pattern of the heuristic is to identify a list of candidate problems that are preventing the Goals from being satisfied, select a problem from either the most recent Checkpoint or an unexplored problem from a previous Checkpoint to solve, and build an Effect List that the heuristic anticipates will correct the issue. This methodology is simple, intuitive, and serves as a good example of how other domain authors can utilize the planning library.

This solution is evaluated by comparing the resulting plan and solution times against the results of the planning competition. The plans produced by the library satisficing solution are of similar length and solution time to the competition results. This makes a strong case for the feasibility of this approach.

The Blocks World domain is one of the oldest and most quintessential planning domains. Because it is clear and simple, it has been by far the most frequently used example in the AI planning literature since the 1960s and is still used by introductory AI courses [67]. It has since been adapted to the planning competition and used as recently as 2011 in the IPCC 2011 Learning track.

The domain consists of Blocks which can be moved to form stacks. The initial problem and goal both describe a stack configuration and the problem consists of finding a series of moves to convert the initial configuration into the goal configuration. A solution provably exists in less than $4n$ moves and can be found in n time. However, finding an optimal solution has been shown to be NP hard [68].

PLANNING LIBRARY MODEL

States and Effects

Figure 25 shows the UML diagram capturing the State Variables for this domain.

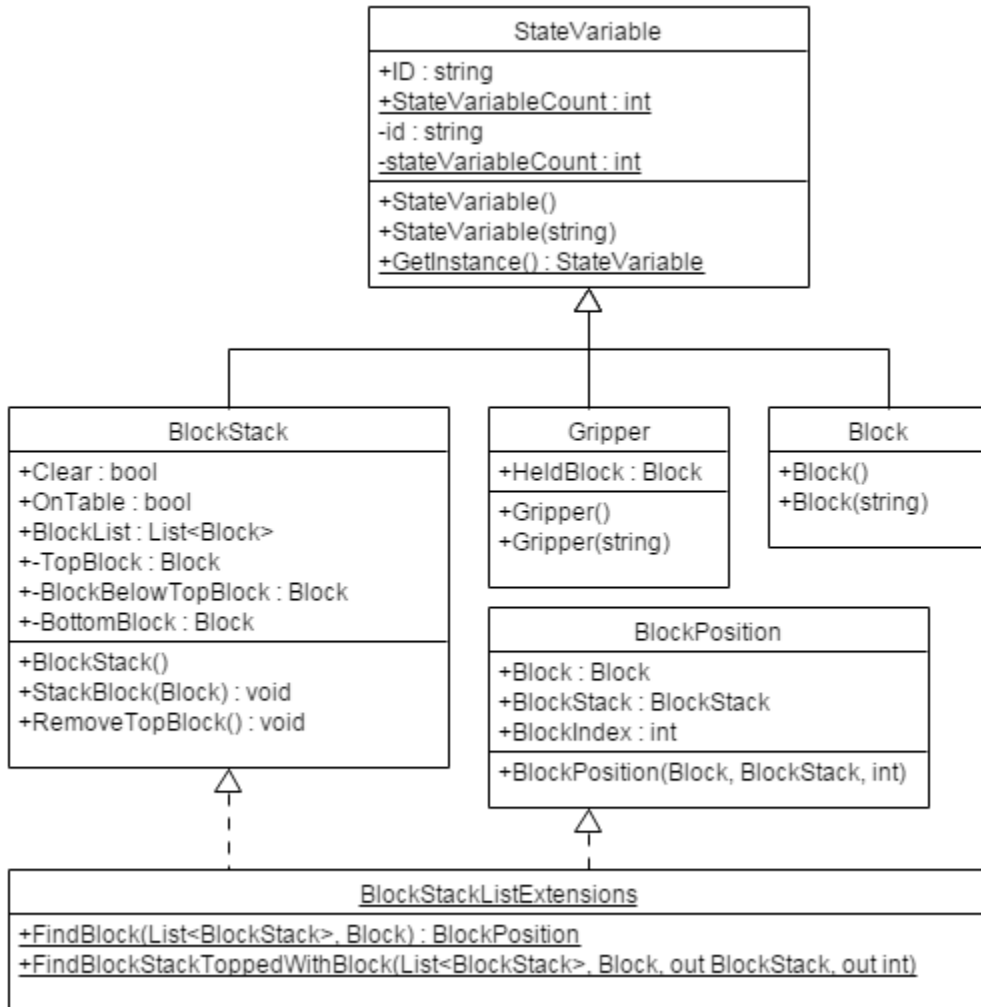


Figure 25: BlocksWorld StateVariable UML Diagram

A significant aspect of this domain is the notion of block stacks. In the PDDL domain, the stack of blocks can be inferred by iteratively processing the ‘on’ predicate list. However, there is no structure to represent the stack itself. When considering the set of actions which meet the PDDL preconditions the only blocks which can be targeted by the actions are blocks in the gripper and blocks on top of stacks. When considering the Goal function, a stack structure

prevents excessively searching the 'on' predicate list for comparison and reduces that to comparing IDs. Lastly, when modeling the Block itself, it is pointless to have an OnTable and Clear field for all Blocks except for the bottom Block and top Block. Having a stack structure eliminates the need for these fields or other workarounds that attempt to preserve memory such as having a separate type for the bottom and top Blocks. It also demonstrates differences in effective modeling using predicate lists and object webs.

In addition to purely structural information, the BlockStack class contains several programming convenience properties and methods used by Effects. Lists of BlockStacks have search oriented extension methods to brute force search the position of a Block (used by the Goal) and the position of a BlockStack amongst its siblings topped with a Block (used by Effects). The Block and Gripper objects are simple enough to be self-explanatory.

The BlocksWorldState implementation contains StateVariables for the Gripper and a List of BlockStacks which is fairly straightforward.

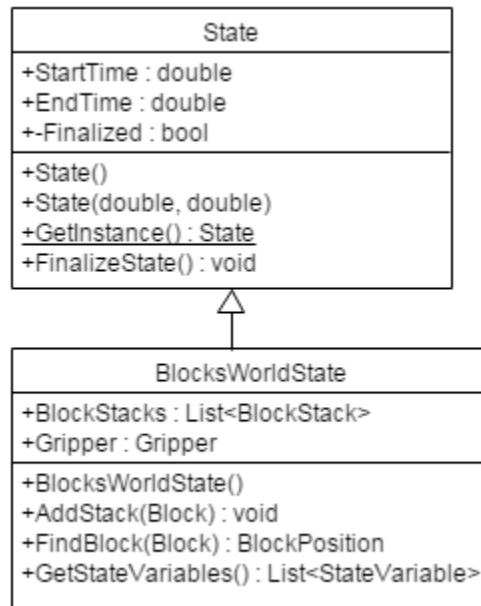


Figure 26: BlocksWorld State UML Diagram

The AddStack method is used by the PutDown Effect to create a new BlockStack with the provided Block as a base. FindBlock invokes the equivalent extension method from BlockStackListExtensions for programmer convenience. Less obvious is the GetStateVariables

method which is used for synchronization. The default implementation from StateVariableContainer returns StateVariables which are fields or part of an enumerable collection. In this case that consists of the Gripper and the BlockStacks. However, the Effects need to synchronize to the Blocks themselves to be PDDL equivalent, and the default implementation does not recursively iterate nested State Variables. The override implementation ensures that the List of Blocks contained in the BlockStacks and the Block held by the Gripper are included in the StateVariables considered for synchronization. An alternate solution would be to include a List of Blocks on the State itself. The override implementation is quite small as shown below.

```
public override List<StateVariable> GetStateVariables()
{
    List<StateVariable> returnSV = new List<StateVariable>();
    returnSV.AddRange(BlockStacks);
    returnSV.Add(Gripper);
    if (Gripper.HeldBlock != null)
    {
        returnSV.Add(Gripper.HeldBlock);
    }
    returnSV.AddRange(this.BlockStacks.SelectMany(s => s.BlockList));
    return returnSV;
}
```

The domain requires four Effects which are equivalent to the PDDL domain. The UML for these four methods is shown below. This equivalence enables plan comparison and a small conversion method enables the plans returned to be checked by VAL [69].

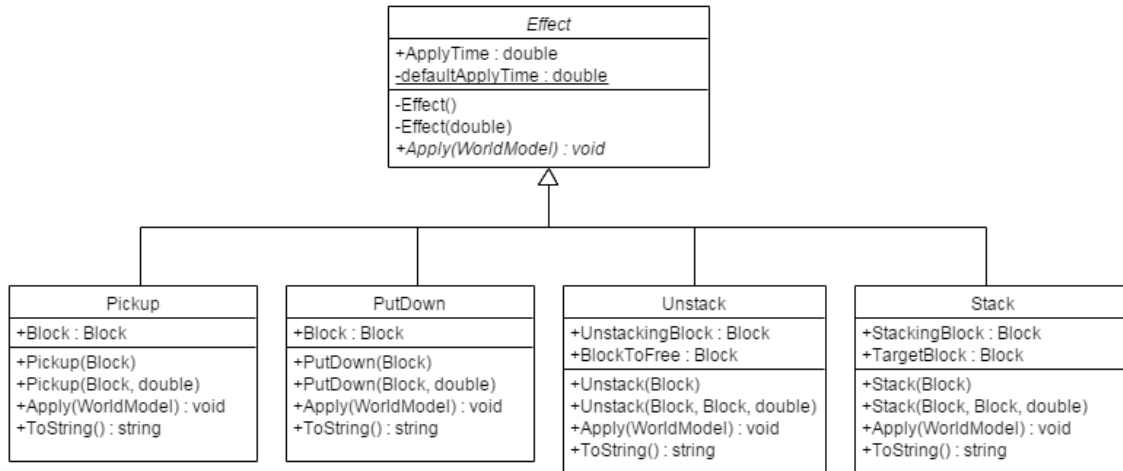


Figure 27: BlocksWorld Effect UML Diagram.

Each Effect implements the PDDL preconditions using an if-statement which throws an Exception if not met. The intent is for only valid Effects to even be instantiated. Any invocation of apply that would be meaningless to run is a bug. The Apply method for PutDown is shown below.

```

public override void Apply(WorldModel worldModel)
{
    BlocksWorldState currentState = (BlocksWorldState)worldModel.CurrentState;
    List<BlockStack> blockStacks = currentState.BlockStacks;
    Gripper gripper = currentState.Gripper;

    if (gripper.HeldBlock == Block)
    {
        gripper.HeldBlock = null;
        currentState.AddStack(Block);
    }
    else
    {
        throw new InvalidOperationException();
    }
}
  
```

A more natural Effect definition would probably refer to the BlockStack directly rather than name the Blocks being stacked. This would be slightly more efficient because the apply method would not need to search the stacks for the block in question to modify the appropriate stack. However, I wanted the actions to have a certain amount of equivalence so they could be easily compared with PDDL results.

Goals and Mismatches

As PlanningBranches are explored the Goals are checked as part of WorldModel advancement. The PDDL BlocksWorld domain used in the competition has a Goal consisting of a single conjunctive normal form expression of predicates. For example:

```
(:goal (AND (ON D C) (ON C B) (ON B A)))
```

This describes a single stack of four Blocks ordered alphabetically bottom to top, A to D.

Goal State



Figure 28: Example BlocksWorld Goal.

I have implemented an equivalent Goal called StackedBlocksGoal shown in Figure 29.

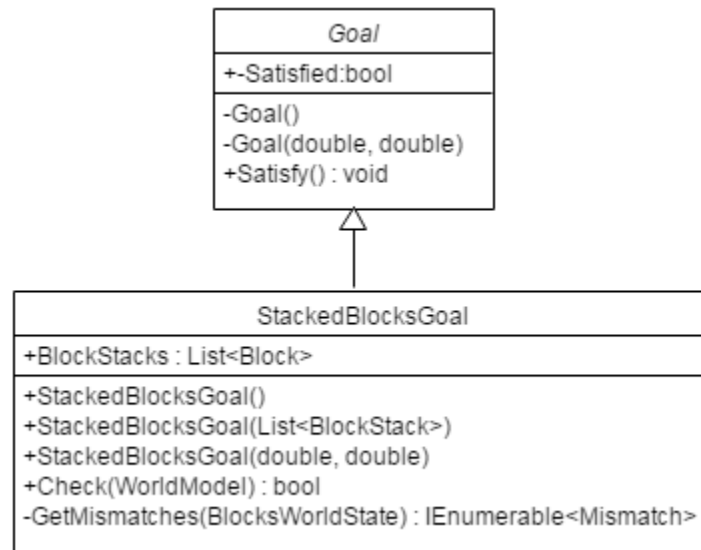


Figure 29: BlocksWorld Goal UML Diagram.

This Goal is intended to meet or exceed the descriptive power of the PDDL equivalent domain. The Check function compares each BlockStack on the Goal to the corresponding BlockStack on the current State. The corresponding stack of a BlockStack on the Goal is the BlockStack on the current State which contains the bottom Block of the BlockStack on the Goal. A BlockStack on the current State may correspond to multiple Goal BlockStacks. A BlockStack on the Goal matches its corresponding Stack on the current State iff it meets these criteria:

- 1) Each Block in the Goal BlockStack exists on the current State.
- 2) Each Block in the Goal BlockStack that is not the BottomBlock is above the same Block on the current State.
- 3) If the Goal BlockStack is OnTable then the BottomBlock of the Goal is also the BottomBlock of the corresponding Stack.
- 4) If the Goal BlockStack is Clear then the TopBlock of the Goal is also the TopBlock of the corresponding Stack.

If a BlockStack is not matched it is because the State is inconsistent one or more of these criteria. Each inconsistency produces a distinct Mismatch in the GetMismatches method of the StackedBlocksGoal. These Mismatches are smaller unsolved pieces of the Goal which heuristics can attempt to solve. The alternative would have been to create many different Goals from the PDDL description and have the heuristic select from the unsatisfied ones for solving. Many domains contain “implied goals” which are necessary to solve but are inferred rather than explicit. This type of representation is suitable for such goals because it draws a clear distinction between explicit goals of the model and inferred subtasks useful to the heuristic. The Mismatches used for my heuristics are shown below:

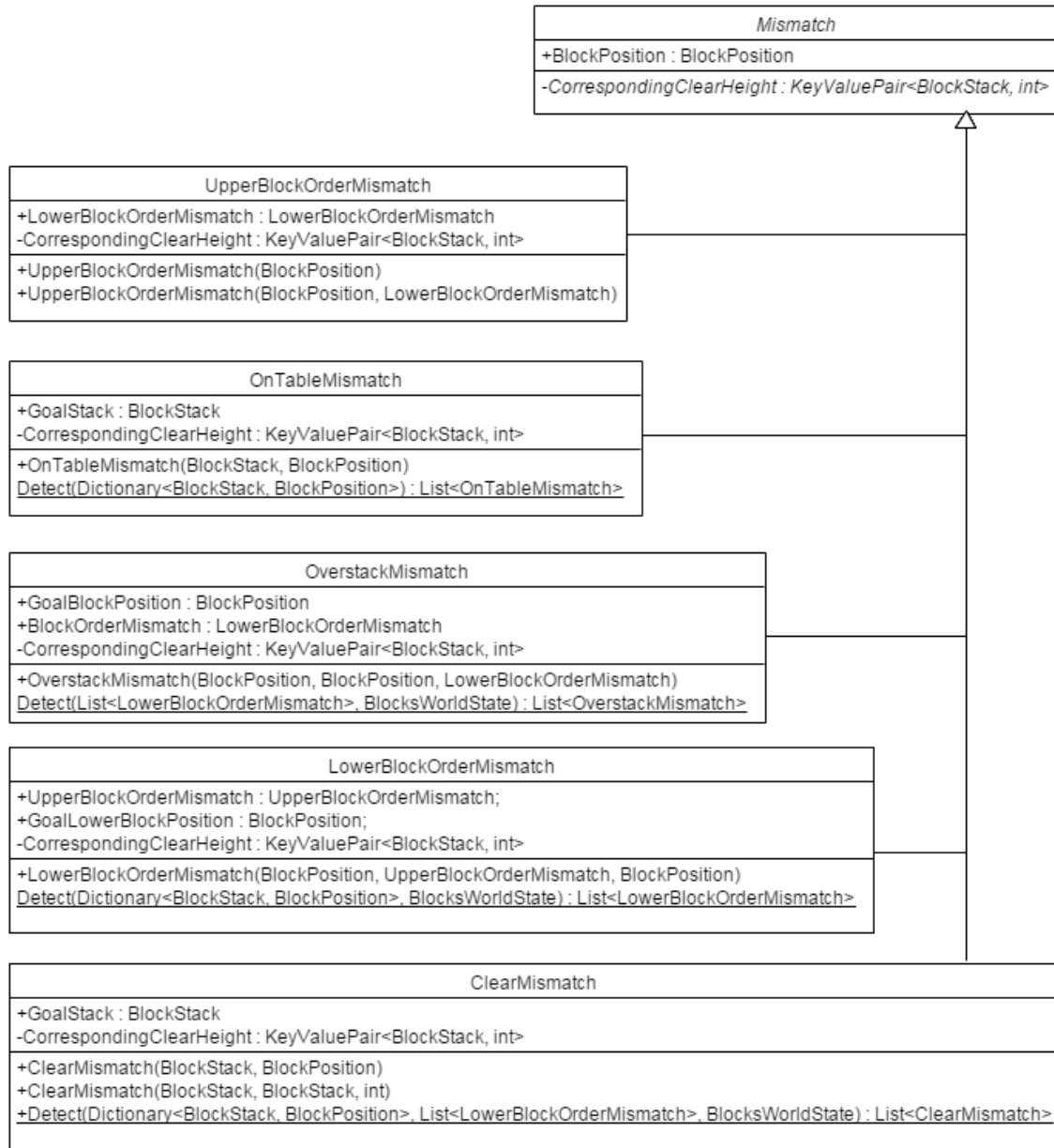


Figure 30: BlocksWorld Mismatch UML Diagram

The OnTableMismatch and ClearMismatch are fairly obvious and correspond to criteria 3 and 4 respectively. The LowerBlockOrderMismatch, UpperBlockOrderMismatch, and OverstackMismatch are all optimizations that capture criteria 2 and merit more explanation.

Consider the problem below.

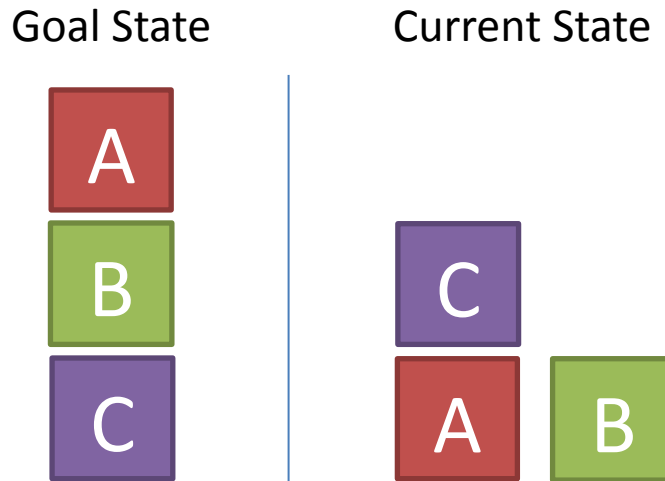


Figure 31: BlocksWorld Overstack Problem

There are two PDDL style predicates which are mismatching the current state.

- 1) (ON B C)
- 2) (ON A B)

When an ON predicate does not match this can consist of two subtasks:

- 1) The lower Block to be stacked on is not clear.
- 2) The upper Block to stack is not clear or is not on the lower Block.

These subtasks are the LowerBlockOrderMismatch and UpperBlockOrderMismatch respectively. There are two useful domain-specific pieces of information which are useful in explaining Overstack Mismatches.

- 1) It is only worthwhile to consider solving the lowest Mismatch in a Goal stack. Solving any higher mismatches will only lead to undoing that work later to resolve a lower Mismatch in the stack.
- 2) Similarly, it is useless to stack on or with any Block which is needed later by the Goal. Block A is the simplest possible example of this. Despite the fact that we *could* solve (ON B C) immediately, it is suboptimal to do so because it will need to be unstacked later to access Block A.

We could account for this information using a large number of Lower and Upper BlockOrderMismatches however it is semantically confusing to use a BlockOrderMismatch to track if two Blocks are correctly ordered but still need to be moved later. For this reason we have a separate type of Mismatch called an Overstack mismatch which tracks this fact. Each Block above the lowest BlockOrderMismatch in the Goal BlockStack is overstacked if that Block is not clear. Together these observations and structures are sufficient to capture all the ordering criteria of the Goal.

The set of Mismatches can be filtered before a heuristic selects a Mismatch to pursue using the following criteria:

- 1) Filter any Mismatch that is not the bottom Mismatch of its Goal BlockStack.
- 2) Filter any Mismatch that is not the top Mismatch of its State BlockStack.
- 3) Filter any Overstack Mismatches because they eventually become an UpperBlockOrder Mismatches before they are solved.
- 4) Filter any BlockOrder Mismatches in a Goal BlockStack with an OnTable Mismatch.
- 5) Filter any Clear Mismatches above that State BlockStacks clear height.

By filtering on these criteria the branching factor of search is greatly reduced. The filtered mismatches cannot be entirely ignored however. Any Blocks involved in a mismatch provably need to be moved at some later point and should not be stacked on according to observation 2) above. To move them, their stacks need to be cleared to at least the height of the Block related to the mismatch. This height is called the clear height and is essentially captures the constraints of all filtered Mismatches. The clear height of a State BlockStack is the minimum clear height of all Mismatches which pertain to that BlockStack. This value is calculated at the beginning of each Decision Epoch. Counting from 0, a clear height of 1 indicates that the corresponding Stack must be reduced to 2 Blocks before the problem will be solvable.

This description is sufficient to capture positive conditions, such as example Goal above. This description is complete because the PDDL domain never specifies negated condition such as '(not (onTable A))' or any type of general qualifier (exists ?x (onTable ?X)).

In the example problem above GetMismatches will yield a LowerBlockOrderMismatch for Block C, an UpperBlockOrderMismatch for Block B, and an OverstackMismatch for Block A. The clear height of both BlockStacks is the table level (0). Next the Checkpoint filters the lowest Mismatch of each Goal BlockStack and the highest Mismatch of each State BlockStack. This yields a Dictionary of State BlockStack to Mismatch which the heuristic needs to select amongst for solving. For the example problem above filtering reduces the Mismatch set to only the LowerBlockOrderMismatch of Block C. This indicates that C needs to move because it is above the clear height of its BlockStack. The eventual plan will be (Unstack C A) (PutDown C) (Pickup B) (Stack B C) (Pickup A) (Stack A B).

CHAPTER 7 – BLOCKS WORLD EXAMPLE DOMAIN

BLIND HEURISTIC

To demonstrate LinkTree usage and as a means of verifying correctness for solutions produced by my satisficing heuristic I created a brute force breadth-first search baseline heuristic. This solution is admissible; if a solution exists at depth n , this heuristic is guaranteed to find it because it is attempting all potentially useful actions. The heuristic uses all of the Mismatch identification, filtering logic, and clear height logic described in the Goals and Mismatches section when building the Checkpoint but it is otherwise naïve and wasteful, particularly when a State is encountered multiple times from different paths.

The UML for the Blind Heuristic is shown below.

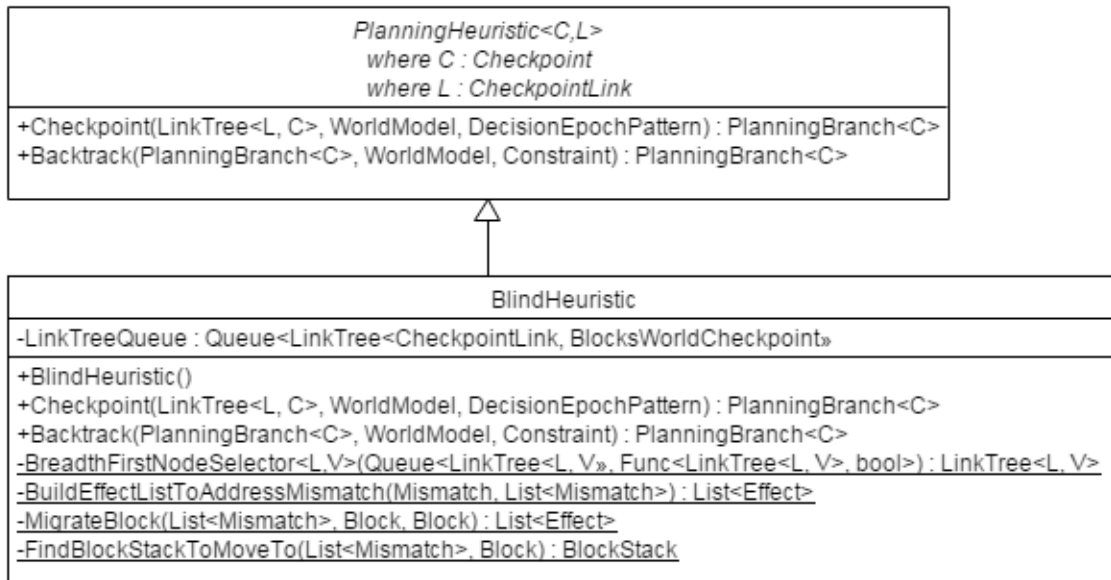


Figure 32: BlindHeuristic UML Diagram

The Backtrack function is a no-op planning failure because the domain has no Constraints and if stalling occurs it is a bug. The Checkpoint function consists of a few simple tasks:

- 1) Select a LinkTree node to expand using breadth first search.
- 2) Select a top mismatch which is as yet unexplored from that node.
- 3) Build an Effect List to migrate one Block to a better position.

- 4) Return a Planning Branch that executes this Effect List and triggers the next Decision Epoch.

The BreadthFirstNodeSelector function addresses point 1. The other three static functions address point 3. By way of example, consider the problem below:

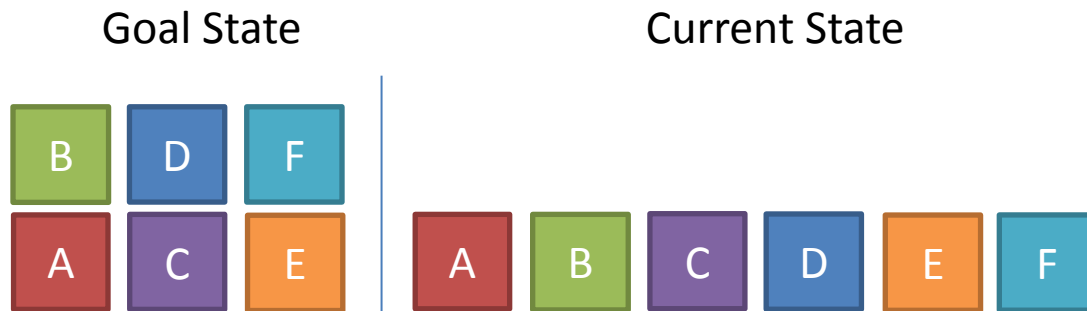


Figure 33: BlocksWorld BFS Problem

At the start the Goal will produce three UpperBlockOrderMismatches, one for each unstacked pair. At each Checkpoint the heuristic will choose one unsolved mismatch, build a pair of Effects that relocates one Block, and return a PlanningBranch which advances the WorldModel by this move.

Because there is only one sensible location to move a Block to all optimal solutions are considered by the heuristic. However, this heuristic does produce an extremely wasteful number of States.

The full search tree for this problem using the blind heuristic is shown below.

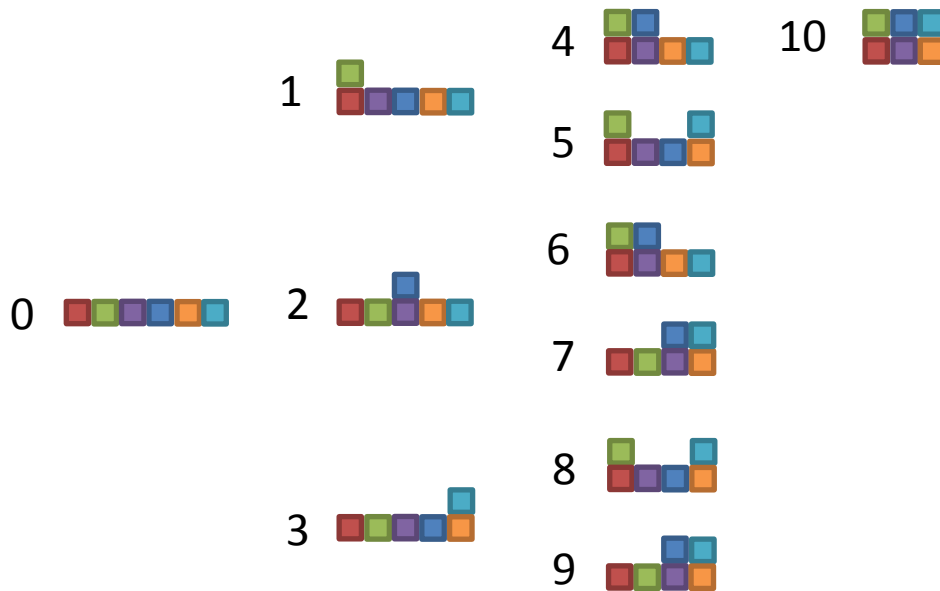


Figure 34: BFS Problem Search Tree

Each State above is numbered according to the order of the Checkpoint that was created for it. The node selection function uses level order queue based tree traversal with the modification that the selection of a node is repeated until it has no more children to explore. This modification allows node 0 to be selected for expansion three times prior to expanding node 1. Each time a new Checkpoint is created that checkpoint is queued into the LinkTreeQueue property of the heuristic. Once the exploration of a node is exhausted it is dequeued and the next node is selected.

In the search tree above you can see that States 4 5 and 6 are each repeated. This is a significant source of waste in the blind heuristic as the same State can be reached from different paths, and there is no built-in mechanism to detect this repetition and cull further search down that path. In the example above, this leads to three unnecessary Checkpoints. However, the total number of states in just the final layer of a problem of this shape is $(n/2)!$ If this problem was expanded to include 2 more blocks, the depth of the tree would increase by 1, and the state count would increase to 42 from 10. The sheer number of states produced by this solution makes it

entirely impractical for large problems. A typical 15 Block problem such as those used in the planning competition produces multiple thousand Checkpoints and can run for several minutes. The sheer number of states makes this approach infeasible for large problems. A more realistic approach is provided in the satisficing heuristic.

CHAPTER 7 – BLOCKS WORLD EXAMPLE DOMAIN

SATISFICING HEURISTIC

In contrast to the blind heuristic the satisficing heuristic only considers a single path and provides no guarantee of an optimal solution. It essentially explores a planning probe [76]. Despite providing no guarantee we will show that in many complex cases this heuristic produces an optimal solution or at least solutions comparable to results of the international planning competition in the years where this domain was used. Our comparison serves as a proof of the feasibility of this approach.

Due to the age of and quantity of literature involving this domain, many satisficing heuristics already exist. Slaney et. al. provide a review of several linear time near-optimal algorithms which solve the domain and provide their own constant time algorithm for selecting each move [77]. The purpose of this work is to prove feasibility of the library not to make the fastest possible domain specific planner for BlocksWorld to date.

The section above has already defined the various mismatches. This section will outline the heuristic algorithm and provide several problems to illustrate the algorithm using the Mismatch definitions and clear height concepts previously defined.

Heuristics of this library have a few elemental tasks to accomplish.

- 1) Identify subtasks that could be solved on the most recent Checkpoint.
- 2) Choose from the LinkTree a single subtask to solve this iteration.
- 3) Construct a Planning Branch which progresses towards a solution

The subtask identification here consists of the Mismatch definitions already covered and is done in the Checkpoint factory. As already mentioned this heuristic explores a planning probe and therefore it never considers any Checkpoints except the current one. This reduces task 2) to choosing 1 subtask from the current Checkpoint to expand.

The satisficing heuristic selects a Mismatch based on the number of misplaced Blocks above the Block related to the Mismatch. These Blocks consist of “incidental” displacement, moves not directly related to solving the Mismatch, which can lead to suboptimal solutions.

Suboptimal solutions can only occur when a Block that is needed in a Goal BlockStack later is placed somewhere other than its final resting place. Each Block displaced in this way increases the chance of a suboptimal move occurring. At the very least, the problems with posted optimal solutions from the planning competition were all solved optimally using this technique.

Once a Mismatch is selected an Effect List needs to be created to address it. In the blind heuristic this Effect List was a single Block migration. In the satisficing heuristic this is a multi-step process which eliminates the Mismatch entirely. Any Blocks above the Mismatch Block are migrated to the table or their Goal position. From there, the Mismatch Block is moved if it is also above the stack's clear height or is the Block of an UpperBlockOrder Mismatch. A Decision Epoch is triggered every time the Effect List is emptied.

CHAPTER 7 – BLOCKS WORLD EXAMPLE DOMAIN

EVALUATION

A primary reason for using an existing well documented domain for evaluation is for comparison against other existing approaches. This section demonstrate the feasibility of this approach compared to existing domain-independent and domain-configurable planners which competed in this domain in previous planning competitions. There were three forms of evaluation:

- 1) Unit test elemental problems.
- 2) PDDL problems consisting of <15 Blocks solved in the optimal track of the IPC.
- 3) PDDL problems consisting of 100-500 Blocks solved in the satisficing track of the IPC.

A few of the unit test problems and their solutions are illustrated here. For the optimal and satisficing test problems we compare numeric data in runtime and plan length.

Sample Problems

The first sample problem considered here is the Overstack example problem from the Mismatch introduction.

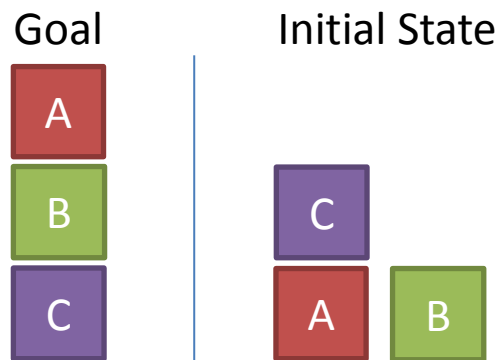


Figure 35: Overstack Problem Revisited.

This case is interesting because the most immediate thing to solve, (ON B C), shouldn't be solved without first moving Block C to table. To prove this a heuristic needs to look beyond the immediate task in some way and account for future problems.

The Mismatch and filter system which uses clear heights accomplishes this beautifully. Block A is above Block B and therefore is required later in the problem. Calculating the clear

height of each stack is sufficient to prove that solving LowerBlockOrder C requires first moving Block C to the table. From there the solution is fairly straight forward.





Checkpoint	Visual State	Mismatch List	Effect List
0		LowerBlockOrder C	Unstack C A PutDown C
1		UpperBlockOrder B	Pickup B Stack B C
2		UpperBlockOrder A	Pickup A Stack A B
3			

Table 6: Overstack Problem Solution

Each time the current Mismatch is solved the Effect List should be empty which triggers a new Decision Epoch. In this case, the filtering process always ensures that there is only one Mismatch to consider from because Mismatches are filtered based on the lowest Mismatch of a Goal BlockStack and the highest Mismatch of a State BlockStack. Once the LowerBlockOrder Mismatch is solved the UpperBlockOrder Mismatch for Block B is no longer filtered because it is now the bottom Mismatch of its corresponding Goal BlockStack. Once that is solved, the UpperBlockOrder Mismatch for Block A is no longer filtered. This process would continue proceeding up the Goal BlockStack regardless of how many Blocks it contained.

The next example shows a case with multiple Goal and State BlockStacks so that multiple Mismatches will exist after filtering that the heuristic has to choose between.

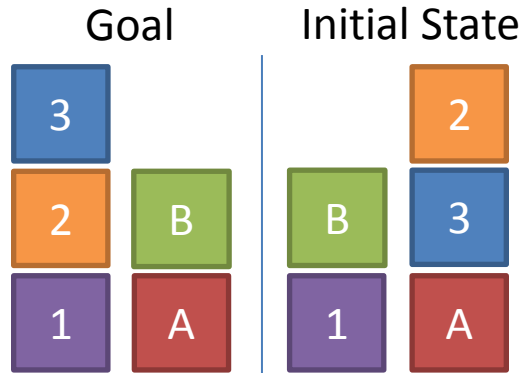


Figure 36: BlocksWorld Swapstack Problem

In the initial Checkpoint Block 1 and Block A need to be clear before they can be stacked on. This creates a LowerBlockOrder Mismatch for Block 1 and Block A. Next the Block above the LowerBlockOrder Mismatch on the Goal BlockStack needs an UpperBlockOrder Mismatch as a subtask to clear those Blocks and move them on to their corresponding lower Block. Next, every Block above these in their Goal BlockStacks are Overstacked if they are not clear. This creates an Overstack Mismatch for Block 3. Table 7 shows a list of all identified Mismatches and their clear heights. Table 8 compiles clear height by State BlockStack.

Mismatch Type	Mismatch Block	Corresponding Stack (Bottom Block)	Clear Height
LowerBlockOrder	1	1	0
UpperBlockOrder	2	A	2
Overstack	3	A	1
LowerBlockOrder	A	A	0
UpperBlockOrder	B	1	1

Table 7: Swapstack Initial Problem Mismatches

Corresponding Stack (Bottom Block)	Clear Height
A	0
1	0

Table 8: Swapstack Initial Problem Clear Height Dictionary

In the initial Checkpoint the only necessary filter is to take the lowest Mismatch of each GoalStack. This yields two mismatches shown in the table below. At this point the heuristic needs to select which to solve in the first Decision Epoch. The satisficing (non-optimal) criteria for this preference is to select the mismatch with the minimum displacement. LowerBlockOrder 1 requires displacing 1 Block, whereas solving LowerBlockOrder A requires displacing 2 Blocks.

Therefore LowerBlockOrder 1 is selected by the satisficing heuristic. In this example either choice produces an optimal result. After the initial Checkpoint there is only 1 Mismatch each Decision Epoch and so search becomes trivial.






Checkpoint	Visual State	Mismatch List	Effect List
0		LowerBlockOrder 1 LowerBlockOrder C	Unstack B 1 PutDown B
1		UpperBlockOrder 2	Unstack 2 3 Stack 2 1
2		UpperBlockOrder 3	Unstack 3 A Stack 3 2
3		UpperBlockOrder B	Pickup B Stack B A
4			

Table 9: Swapstack Problem Solution

IPC Optimal Track Problems

The performance of the blind heuristic and satisficing heuristic is compared using test problems with a known optimal length from the optimal planning track of the AIPS 2000 planning competition. In all of these cases the Satisficing Heuristic returned an optimal plan. The results are shown in Table 10:

Problem Name	Blind Heuristic Runtime	Satisficing Heuristic Runtime	Plan Length
blocks-4-0	<1ms	<1ms	6
blocks-4-1	<1ms	<1ms	10
blocks-4-2	<1ms	<1ms	6
blocks-5-0	<1ms	<1ms	12
blocks-5-1	<1ms	<1ms	10
blocks-5-2	<1ms	<1ms	16
blocks-6-0	<1ms	<1ms	12
blocks-6-1	1ms	<1ms	10
blocks-6-2	1ms	<1ms	20
blocks-7-0	1ms	<1ms	20
blocks-7-1	2ms	<1ms	22
blocks-7-2	1ms	1ms	20
blocks-8-0	1ms	<1ms	18
blocks-8-1	2ms	1ms	20
blocks-8-2	1ms	<1ms	16
blocks-9-0	2ms	<1ms	30
blocks-9-1	2ms	1ms	28
blocks-9-2	2ms	1ms	26
blocks-10-0	3ms	1ms	34
blocks-10-1	3ms	1ms	32
blocks-10-2	3ms	1ms	34
blocks-11-0	2ms	1ms	32
blocks-11-1	3ms	1ms	30
blocks-11-2	3ms	2ms	34
blocks-12-0	3ms	1ms	34
blocks-12-1	3ms	1ms	34
All of Above	51ms	37ms	n/a

Table 10: Heuristic Comparison on IPC Optimal Track Problems

IPC Satisficing Track Problems

The table below compares the runtime and plan length of planners which competed in the satisficing track of the international planning competition against the satisficing heuristic. The claim of this work is not that this library produces the best solution for a BlocksWorld domain but rather that it produces feasible solutions on a comparable domain and can enable solutions for arbitrarily complex domains.

Table 11 shows IPC track 1 results compared with the satisficing heuristic. Track 1 contains only domain-independent planners.

Problem Name	Best IPC Plan Length	Satisficing Heuristic Plan Length	Best IPC Time	Satisficing Heuristic Runtime
blocks-28-0	102	92	47.83s	60ms
blocks-36-1	138	134	23.21s	48ms
blocks-39-0	144	136	34.35s	49ms
blocks-50-1	188	176	893.17s	79ms

Table 11: Satisficing Heuristic vs. Track 1 IPC Results.

Table 12 shows IPC track 2 Results compared with the satisficing heuristic. Track 2 contains domain-configurable planners such as TALPlanner and includes problems of much greater scale.

Problem Name	Best IPC Plan Length	Satisficing Heuristic Plan Length	Best IPC Time	Satisficing Heuristic Runtime
probblocks-100-1	370	372	309ms	147ms
probblocks-200-1	744	740	479ms	492ms
probblocks-300-1	1158	1136	699ms	1s
probblocks-400-1	1556	1562	999ms	2s
probblocks-500-1	1954	1948	1.409s	3s

Table 12: Satisficing Heuristic vs. Track 2 IPC Results.

Previous domain specific planners for this domain have solved over 10,000 Block problems in under a second as far back as 1995 [67]. More problems can be generated using the generator described in Blocks World revisited and available on their website [70].

Ultimately, the domain is proven to be NP Hard. Therefore, no heuristic can exist which is both optimal and scalable. Optimal heuristics exist which solve problems up to ~150 Blocks in reasonable time. Satisficing heuristics exist which solve 10,000 Block problems in under a second.

REFERENCES

- [1] Laird, J. (2012). *The Soar cognitive architecture*. MIT Press.
- [2] Anderson, J. R. (2013). *The architecture of cognition*. Psychology Press.
- [3] Langley, P. (2006). Cognitive architectures and general intelligent systems. *AI magazine*, 27(2), 33.
- [4] Kambhampati, S. (1997). Refinement planning as a unifying framework for plan synthesis. *AI magazine*, 18(2), 67.
- [5] Nau, D. S. (2007). Current trends in automated planning. *AI magazine*, 28(4), 43.
- [6] Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model Checking* The MIT Press. Cambridge, Massachusetts, London, UK.
- [7] Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivančić, F., & Gupta, A. (2013). Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s), 95.
- [8] Edelkamp, S., & Greulich, C. (2014, August). Solving physical traveling salesman problems with policy adaptation. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on* (pp. 1-8). IEEE.
- [9] Yannakakis, G. N. (2012, May). Game AI revisited. In *Proceedings of the 9th conference on Computing Frontiers* (pp. 285-292). ACM.
- [10] Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1), 125-130.
- [11] Edelkamp, S., & Plaku, E. (2014, August). Multi-goal motion planning with physics-based game engines. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on* (pp. 1-8). IEEE.
- [12] Ventures, M. D. (2006). Stanley: The robot that won the DARPA Grand Challenge. *Journal of field Robotics*, 23(9), 661-692.
- [13] Kovacs, D. L. (2011). BNF definition of PDDL 3.1. Unpublished manuscript from the IPC-2011 website.
- [14] Sanner, S. (2010). Relational dynamic influence diagram language (rddl): Language description. Unpublished ms. Australian National University.
- [15] Smith, D. E., Frank, J., & Cushing, W. (2008, September). The anml language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.
- [16] Bäckström, C., & Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, 11(4), 625-655.
- [17] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., ... & Wilkins, D. (1998). PDDL-the planning domain definition language.
- [18] Shah, M., Chrapa, L., Jimoh, F., Kitchin, D., McCluskey, T., Parkinson, S., & Vallati, M. (2013). Knowledge engineering tools in planning: State-of-the-art and future challenges. *Knowledge Engineering for Planning and Scheduling*, 53.

- [19] Jilani, R., Crampton, A., Kitchin, D. E., & Vallati, M. (2014). Automated Knowledge Engineering Tools in Planning: State-of-the-art and Future Challenges.
- [20] Wickler, G. Using Static Graphs in Planning Domains to Understand Domain Dynamics. *Knowledge Engineering for Planning and Scheduling*, 69.
- [21] Waser, J., Fuchs, R., Ribičić, H., Schindler, B., Blöschl, G., & Gröller, M. E. (2010). World lines. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6), 1458-1467.
- [22] Fox, M., & Long, D. (2003). PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.(JAIR)*, 20, 61-124.
- [23] Younes, H. L., & Littman, M. L. (2004). PPDDL1. 0: The language for the probabilistic part of IPC-4. In *Proc. International Planning Competition*.
- [24] Keller, T., & Eyerich, P. (2012, May). PROST: Probabilistic Planning Based on UCT. In *ICAPS*.
- [25] Fox, M., & Long, D. (2002). PDDL+: Modelling continuous time-dependent effects. In *Proc. 3rd International NASA Workshop on Planning and Scheduling for Space*.
- [26] Coles, A. J., Coles, A. I., Fox, M., & Long, D. (2012). COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research*, 1-96.
- [27] Long, P. G. D., & Beck, M. F. J. C. Planning Modulo Theories: Extending the Planning Paradigm. *PlanSIG2011*, 39.
- [28] Fox, M. (2014, January). A Modular Architecture for Hybrid Planning with Theories. In *Principles and Practice of Constraint Programming* (pp. 1-2). Springer International Publishing
- [29] Albore, A., Peyrard, N., Sabbadin, R., & Teichteil-Königsbuch, F. (2015, April). An Online Replanning Approach for Crop Fields Mapping with Autonomous UAVs. In *ICAPS* (pp. 259-267).
- [30] Maillard, A., Pralet, C., Jaubert, J., Sebbag, I., Fontanari, F., & L'Hermitte, J. (2015, April). Ground and Onboard Decision-Making on Satellite Data Downloads. In *ICAPS* (pp. 273-281).
- [31] Mersheeva, V., & Friedrich, G. (2015, April). Multi-UAV monitoring with priorities and limited energy resources. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*.
- [32] Riabov, A. V., Sohrabi, S., Sow, D., Turaga, D., Udrea, O., & Vu, L. (2015, April). Planning-Based Reasoning for Automated Large-Scale Data Analysis. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*.
- [33] Surovik, D. A., & Scheeres, D. J. (2015, May). Heuristic search and receding-horizon planning in complex spacecraft orbit domains. In *Eighth Annual Symposium on Combinatorial Search*.
- [34] Čáp, M., Vokřínek, J., & Kleiner, A. (2015, April). Complete Decentralized Method for On-Line Multi-Robot Trajectory Planning in Well-Formed Infrastructures. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*.
- [35] Pralet, C., Verfaillie, G., Maillard, A., Hébrard, E., Jozefowicz, N., Huguet, M. J., ... & Jaubert, J. (2014, May). Satellite Data Download Management with Uncertainty about the Generated Volumes. In *ICAPS*.
- [36] Cirillo, M., Pecora, F., Andreasson, H., Uras, T., & Koenig, S. (2014, June). Integrated Motion Planning and Coordination for Industrial Vehicles. In *ICAPS*.

- [37] Kumar, T. S., Jung, S. J., & Koenig, S. (2014, May). A Tree-Based Algorithm for Construction Robots. In ICAPS.
- [38] Levine, S. J., & Williams, B. C. (2014, May). Concurrent Plan Recognition and Execution for Human-Robot Teams. In ICAPS.
- [39] Lipovetzky, N., Burt, C. N., Pearce, A. R., & Stuckey, P. J. (2014, May). Planning for Mining Operations with Time and Resource Constraints. In ICAPS.
- [40] Bernardini, S., Fox, M., & Long, D. (2014, May). Planning the Behaviour of Low-Cost Quadcopters for Surveillance Missions. In ICAPS.
- [41] Awaad, I., Kraetzschmar, G. K., & Hertzberg, J. (2014, November). Finding Ways to Get the Job Done: An Affordance-Based Approach. In ICAPS.
- [42] Wang, H., Kurniawati, H., Singh, S. P., & Srinivasan, M. (2015, April). In-silico Behavior Discovery System: An Application of Planning in Ethology. In ICAPS (pp. 296-305).
- [43] Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., ... & Carreras, M. (2015, April). ROSPlan: Planning in the robot operating system. In Twenty-Fifth International Conference on Automated Planning and Scheduling.
- [44] Bercher, P., Biundo, S., Geier, T., Hoernle, T., Nothdurft, F., Richter, F., & Schattenberg, B. (2014, May). Plan, Repair, Execute, Explain-How Planning Helps to Assemble your Home Theater. In ICAPS.
- [45] Kumar, A., Singh, S. S., Gupta, P., & Parija, G. R. (2014, May). Near-Optimal Nonmyopic Contact Center Planning Using Dual Decomposition. In ICAPS.
- [46] Parkinson, S., Gregory, P., Longstaff, A. P., & Crampton, A. (2014, May). Automated Planning for Multi-Objective Machine Tool Calibration: Optimising Makespan and Measurement Uncertainty. In ICAPS.
- [47] Boselli, R., Cesarini, M., Mercurio, F., & Mezzanzanica, M. (2014, May). Planning meets data cleansing. In Twenty-Fourth International Conference on Automated Planning and Scheduling.
- [48] Chanel, C. P. C., Lesire, C., & Teichteil-Königsbuch, F. (2014, May). A robotic execution framework for online probabilistic (re) planning. In Twenty-Fourth International Conference on Automated Planning and Scheduling.
- [49] Khandelwal, P., Yang, F., Leonetti, M., Lifschitz, V., & Stone, P. (2014, June). Planning in Action Language BC while Learning Action Costs for Mobile Robots. In ICAPS.
- [50] Ruiken, D., Lanighan, M. W., & Grupen, R. A. (2014, May). Path Planning for Dexterous Mobility. In ICAPS.
- [51] Hernández, C., Baier, J. A., & Asín, R. (2014, May). Making A* Run Faster than D*-Lite for Path-Planning in Partially Known Terrain. In ICAPS.
- [52] Ondruska, P., & Posner, I. (2014, May). The route not taken: Driver-centric estimation of electric vehicle range. In Twenty-Fourth International Conference on Automated Planning and Scheduling.
- [53] Bevacqua, G., Cacace, J., Finzi, A., & Lippiello, V. (2015, April). Mixed-Initiative Planning and Execution for Multiple Drones in Search and Rescue Missions. In ICAPS (pp. 315-323).

- [54] Bacchus, F., & Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1), 123-191.
- [55] Kvarnström, J., & Doherty, P. (2000). TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4), 119-169.
- [56] Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). SHOP2: An HTN planning system. *J. Artif. Intell. Res.(JAIR)*, 20, 379-404.
- [57] Kautz, H. (1998). The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework.
- [58] Zhou, N. F., Bartak, R., & Dovier, A. (2015). Planning as tabled logic programming. *Theory and Practice of Logic Programming*, 15(4-5), 543-558.
- [59] Gebser, M., Kaufmann, B., Romero, J., Otero, R., Schaub, T., & Wanko, P. (2013, July). Domain-Specific Heuristics in Answer Set Programming. In *AAAI*.
- [60] Jonas, M., Gilbert, R. R., Ferguson, J., Varsamopoulos, G., & Gupta, S. K. (2012, June). A transient model for data center thermal prediction. In *Green Computing Conference (IGCC), 2012 International* (pp. 1-10). IEEE.
- [61] Jonas, M. (2011). JPDL: A fresh approach to planning domain modeling. *KEPS 2011*, 63.
- [62] Hoffmann, J. (2001). FF: The fast-forward planning system. *AI magazine*, 22(3), 57.
- [63] Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1), 127-177.
- [64] Chen, Y., Hsu, C. W., & Wah, B. W. (2004). SGPlan: Subgoal partitioning and resolution in planning. Edelkamp et al. (Edelkamp, Hoffmann, Littman, & Younes, 2004).
- [65] Cushing, W., Kambhampati, S., & Weld, D. S. (2007, January). When is temporal planning really temporal?. In *Proceedings of the 20th international joint conference on Artificial intelligence* (pp. 1852-1859). Morgan Kaufmann Publishers Inc..
- [66] Barthelemy, O., & Jacopin, E. (2010). Real-Time Planning for Video-Games: A Purpose for PDDL. In *ICAPS 2010 Planning in Games Workshop*. Προσπέλαση από: <http://skatgame.net/mburo/icaps2010-pg/ICAPSPG>.
- [67] Slaney, J., & Thiébaux, S. (1995). Blocks World Tamed--Ten thousand blocks in under a second.
- [68] Chenowet, S. V. (1991). On the NP-hardness of blocks world.
- [69] Howey, R., Long, D., & Fox, M. (2004, November). VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on* (pp. 294-301). IEEE.
- [70] Slaney, J., & Thiébaux, S. Web Resource <http://users.cecs.anu.edu.au/~jks/cgi-bin/bwstates/bwcgi>
- [71] Löhr, J., Wehrle, M., Fox, M., & Nebel, B. (2014, June). Symbolic Domain Predictive Control. In *Proceedings of the 28th National Conference on Artificial Intelligence (AAAI 2014)*.

- [72] Thiébaux, S., Gretton, C., Slaney, J. K., Price, D., & Kabanza, F. (2006). Decision-Theoretic Planning with non-Markovian Rewards. *J. Artif. Intell. Res.(JAIR)*, 25, 17-74
- [73] Gabaldon, A. (2002, July). Non-markovian control in the situation calculus. In *AAAI/IAAI* (pp. 519-525).
- [74] Gonzalez, G., Baral, C., & Gelfond, M. (2003, August). Alan: An action language for non-markovian domains. In *NonMon. Reasoning, Action and Change Workshop*.
- [75] Moore, J. D., Chase, J. S., Ranganathan, P., & Sharma, R. K. (2005, April). Making Scheduling "Cool": Temperature-Aware Workload Placement in Data Centers. In *USENIX annual technical conference, General Track* (pp. 61-75).
- [76] Lipovetzky, N., & Geffner, H. (2011, March). Searching for Plans with Carefully Designed Probes. In *ICAPS* (pp. 154-161).
- [77] Slaney, J., & Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, 125(1), 119-153.

APPENDIX A
TABLE OF ICAPS SURVEY

Title	Year	Category
An Online Replanning Approach for Crop Fields Mapping with Autonomous UAVs. [29]	2015	Dependent
Ground and Onboard Decision-Making on Satellite Data Downloads. [30]	2015	Dependent
Multi-UAV Monitoring with Priorities and Limited Energy Resources. [31]	2015	Dependent
Planning-Based Reasoning for Automated Large-Scale Data Analysis. [32]	2015	Dependent
Heuristic Search and Receding-Horizon Planning in Complex Spacecraft Orbit Domains. [33]	2015	Dependent
Complete Decentralized Method for On-Line Multi-Robot Trajectory Planning in Well-formed Infrastructures. [34]	2015	Dependent
Satellite Data Download Management with Uncertainty about the Generated Volumes. [35]	2014	Dependent
Integrated Motion Planning and Coordination for Industrial Vehicles. [36]	2014	Dependent
A Tree-Based Algorithm for Construction Robots. [37]	2014	Dependent
Concurrent Plan Recognition and Execution for Human-Robot Teams. [38]	2014	Dependent
Planning for Mining Operations with Time and Resource Constraints. [39]	2014	Hybrid
Planning the Behaviour of Low-Cost Quadcopters for Surveillance Missions. [40]	2014	Hybrid
Finding Ways to Get the Job Done: An Affordance-Based Approach. [41]	2014	Hybrid
In-silico Behavior Discovery System: An Application of Planning in Ethology. [42]	2015	Independent
ROSPan: Planning in the Robot Operating System. [43]	2015	Independent
Plan, Repair, Execute, Explain - How Planning Helps to Assemble your Home Theater. [44]	2014	Independent
Near-Optimal Nonmyopic Contact Center Planning Using Dual Decomposition. [45]	2014	Independent
Automated Planning for Multi-Objective Machine Tool Calibration: Optimising Makespan and Measurement Uncertainty. [46]	2014	Independent
Planning meets Data Cleansing. [47]	2014	Independent
A Robotic Execution Framework for Online Probabilistic (Re)Planning. [48]	2014	Independent
Planning in Action Language BC while Learning Action Costs for Mobile Robots. [49]	2014	Independent
Path Planning for Dexterous Mobility. [50]	2014	Irrelevant
Making A* Run Faster than D*-Lite for Path-Planning in Partially Known Terrain. [51]	2014	Irrelevant
The Route Not Taken: Driver-Centric Estimation of Electric Vehicle Range. [52]	2014	Irrelevant
Mixed-Initiative Planning and Execution for Multiple Drones in Search and Rescue Missions. [53]	2015	Unknown

APPENDIX B
DATA CENTER PDDL DOMAIN FILE

This example PDDL domain file was generated using our T4 template. It is parameterized using 5 servers that equally contribute to each other with a temporal contribution curve of a square wave with the domain [-1 0]. It has 5 jobs, though job deadlines aren't built in and the heat per job isn't a function of the server.

```
(define (domain transient-thermal-model)
  (:requirements :typing :equality :universal-preconditions :fluents :constraints)
  (:types server job time)
  (:predicates
    (updatedInletTemperatures ?t - time)
    (updatedOutletTemperatures ?t - time)
    (jobCompleted ?j - job)
  )
  (:functions
    (currentTime)
    (time ?t -time)

    (w ?s1 ?s2 - server)

    (cCurveDt0 ?s1 ?s2 - server)
    (cCurveDt1 ?s1 ?s2 - server)

    (temperatureIn ?s - server ?t - time)

    (temperatureOut ?s - server ?t - time)

    (temperatureIncrease ?s - server)

    (jobHeat ?j - job)
    (jobDuration ?j - job)
    (jobStartTime ?j - job)
  )
  (:action updateInletTemperatures
    :parameters (?server0 ?server1 ?server2 ?server3 ?server4 - server ?dt0 ?dt1 - time)
    :precondition
    (and
      (updatedOutletTemperatures ?dt1)
      (not (updatedInletTemperatures ?dt0))
      (= (time ?dt0) (currentTime))
      (= (time ?dt0) (+ (time ?dt1) 1))
    )
    :effect
    (and
      (assign (temperatureIn ?server0 ?dt0)
        (+
          (* (w ?server0 ?server0)
            (+
              (* (cCurveDt1 ?server0 ?server0)
                (temperatureOut ?server0 ?dt1))
              (* (cCurveDt0 ?server0 ?server0)
                (temperatureOut ?server0 ?dt0))
            )
          )
        )
    )
  )
```

```

)
(* (w ?server1 ?server0)
(+
    (* (cCurveDt1 ?server1 ?server0)
    (* (cCurveDt0 ?server1 ?server0)
(temperatureOut ?server1 ?dt1))
(temperatureOut ?server1 ?dt0))
)
)
(* (w ?server2 ?server0)
(+
    (* (cCurveDt1 ?server2 ?server0)
    (* (cCurveDt0 ?server2 ?server0)
(temperatureOut ?server2 ?dt1))
(temperatureOut ?server2 ?dt0))
)
)
(* (w ?server3 ?server0)
(+
    (* (cCurveDt1 ?server3 ?server0)
    (* (cCurveDt0 ?server3 ?server0)
(temperatureOut ?server3 ?dt1))
(temperatureOut ?server3 ?dt0))
)
)
(* (w ?server4 ?server0)
(+
    (* (cCurveDt1 ?server4 ?server0)
    (* (cCurveDt0 ?server4 ?server0)
(temperatureOut ?server4 ?dt1))
(temperatureOut ?server4 ?dt0))
)
)
)
)
(assign (temperatureIn ?server1 ?dt0)
(+
    (* (w ?server0 ?server1)
    (+
        (* (cCurveDt1 ?server0 ?server1)
        (* (cCurveDt0 ?server0 ?server1)
(temperatureOut ?server0 ?dt1))
(temperatureOut ?server0 ?dt0))
)
)
)
(* (w ?server1 ?server1)
(+
    (* (cCurveDt1 ?server1 ?server1)
    (* (cCurveDt0 ?server1 ?server1)
(temperatureOut ?server1 ?dt1))
(temperatureOut ?server1 ?dt0))
)
)
)
(* (w ?server2 ?server1)
(+

```

```

(* (cCurveDt1 ?server2 ?server1)
(temperatureOut ?server2 ?dt1))
(* (cCurveDt0 ?server2 ?server1)
(temperatureOut ?server2 ?dt0))
)
)
(* (w ?server3 ?server1)
(+
(* (cCurveDt1 ?server3 ?server1)
(temperatureOut ?server3 ?dt1))
(* (cCurveDt0 ?server3 ?server1)
(temperatureOut ?server3 ?dt0))
)
)
(* (w ?server4 ?server1)
(+
(* (cCurveDt1 ?server4 ?server1)
(temperatureOut ?server4 ?dt1))
(* (cCurveDt0 ?server4 ?server1)
(temperatureOut ?server4 ?dt0))
)
)
)
)
(assign (temperatureIn ?server2 ?dt0)
(+
(* (w ?server0 ?server2)
(temperatureOut ?server0 ?dt1))
(* (cCurveDt1 ?server0 ?server2)
(temperatureOut ?server0 ?dt0))
)
)
(* (w ?server1 ?server2)
(+
(* (cCurveDt1 ?server1 ?server2)
(temperatureOut ?server1 ?dt1))
(* (cCurveDt0 ?server1 ?server2)
(temperatureOut ?server1 ?dt0))
)
)
(* (w ?server2 ?server2)
(+
(* (cCurveDt1 ?server2 ?server2)
(temperatureOut ?server2 ?dt1))
(* (cCurveDt0 ?server2 ?server2)
(temperatureOut ?server2 ?dt0))
)
)
)
(* (w ?server3 ?server2)
(+
(* (cCurveDt1 ?server3 ?server2)
(temperatureOut ?server3 ?dt1))

```

```

(* (cCurveDt0 ?server3 ?server2)
(temperatureOut ?server3 ?dt0))
)
)
(* (w ?server4 ?server2)
(+
(* (cCurveDt1 ?server4 ?server2)
(* (cCurveDt0 ?server4 ?server2)
(temperatureOut ?server4 ?dt1))
(temperatureOut ?server4 ?dt0))
)
)
)
(assign (temperatureIn ?server3 ?dt0)
(+
(* (w ?server0 ?server3)
(+
(* (cCurveDt1 ?server0 ?server3)
(* (cCurveDt0 ?server0 ?server3)
(temperatureOut ?server0 ?dt1))
(temperatureOut ?server0 ?dt0))
)
)
(* (w ?server1 ?server3)
(+
(* (cCurveDt1 ?server1 ?server3)
(* (cCurveDt0 ?server1 ?server3)
(temperatureOut ?server1 ?dt1))
(temperatureOut ?server1 ?dt0))
)
)
(* (w ?server2 ?server3)
(+
(* (cCurveDt1 ?server2 ?server3)
(* (cCurveDt0 ?server2 ?server3)
(temperatureOut ?server2 ?dt1))
(temperatureOut ?server2 ?dt0))
)
)
(* (w ?server3 ?server3)
(+
(* (cCurveDt1 ?server3 ?server3)
(* (cCurveDt0 ?server3 ?server3)
(temperatureOut ?server3 ?dt1))
(temperatureOut ?server3 ?dt0))
)
)
(* (w ?server4 ?server3)
(+
(* (cCurveDt1 ?server4 ?server3)
(* (cCurveDt0 ?server4 ?server3)
(temperatureOut ?server4 ?dt1))
(temperatureOut ?server4 ?dt0))
)
)

```

```

    )
  )
  (assign (temperatureIn ?server4 ?dt0)
    (+
      (* (w ?server0 ?server4)
        (+
          (* (cCurveDt1 ?server0 ?server4)
            (temperatureOut ?server0 ?dt1))
          (* (cCurveDt0 ?server0 ?server4)
            (temperatureOut ?server0 ?dt0))
        )
      )
      (* (w ?server1 ?server4)
        (+
          (* (cCurveDt1 ?server1 ?server4)
            (temperatureOut ?server1 ?dt1))
          (* (cCurveDt0 ?server1 ?server4)
            (temperatureOut ?server1 ?dt0))
        )
      )
      (* (w ?server2 ?server4)
        (+
          (* (cCurveDt1 ?server2 ?server4)
            (temperatureOut ?server2 ?dt1))
          (* (cCurveDt0 ?server2 ?server4)
            (temperatureOut ?server2 ?dt0))
        )
      )
      (* (w ?server3 ?server4)
        (+
          (* (cCurveDt1 ?server3 ?server4)
            (temperatureOut ?server3 ?dt1))
          (* (cCurveDt0 ?server3 ?server4)
            (temperatureOut ?server3 ?dt0))
        )
      )
      (* (w ?server4 ?server4)
        (+
          (* (cCurveDt1 ?server4 ?server4)
            (temperatureOut ?server4 ?dt1))
          (* (cCurveDt0 ?server4 ?server4)
            (temperatureOut ?server4 ?dt0))
        )
      )
    )
  )
  (updatedInletTemperature ?currentTime)
)
)
(action updateOutletTemperatures
  :parameters (?t - time)
  :precondition
  (and

```

```

        (updatedInletTemperatures ?t)
        (not (updatedOutletTemperatures ?t))
        (= (time ?t) (currentTime))
    )
    :effect
    (and
        (assign (temperatureOut ?server0 ?t) (+ (temperatureIn ?server0 ?t)
        (temperatureIncrease ?server0)))
        (assign (temperatureOut ?server1 ?t) (+ (temperatureIn ?server1 ?t)
        (temperatureIncrease ?server1)))
        (assign (temperatureOut ?server2 ?t) (+ (temperatureIn ?server2 ?t)
        (temperatureIncrease ?server2)))
        (assign (temperatureOut ?server3 ?t) (+ (temperatureIn ?server3 ?t)
        (temperatureIncrease ?server3)))
        (assign (temperatureOut ?server4 ?t) (+ (temperatureIn ?server4 ?t)
        (temperatureIncrease ?server4)))
        (updatedOutletTemperatures ?t)
        (increase (currentTime) 1)
    )
)

(action startJob
  :parameters (?t - time ?j - job ?s - server)
  :precondition
  (and
    (not (jobStartTime ?j))
    (= (time ?t) (currentTime))
    (not (updatedOutletTemperatures ?t))
  )
  :effect
  (and
    (assign (jobStartTime ?j) (currentTime))
    (increase (temperatureIncrease ?s) (jobHeat ?j))
  )
)

(action endJob
  :parameters (?t - time ?j - job ?s - server)
  :precondition
  (and
    (= (currentTime) (+ (jobStartTime ?j) (jobDuration ?j)))
    (= (time ?t) (currentTime))
    (updatedOutletTemperatures ?t)
  )
  :effect
  (and
    (jobCompleted ?j)
    (decrease (temperatureIncrease ?s) (jobHeat ?j))
  )
)
)

```

APPENDIX C
DATA CENTER PDDL FACT FILE


```

(define (problem datacenter1)
  (domain transient-thermal-model)
  (:requirements :typing :equality :universal-preconditions :fluents :constraints)
  (:objects t1n t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 t19 t20 t21 t22 t23 t24
t25 t26 t27 t28 t29 t30 t31 t32 t33 t34 t35 t36 t37 t38 t39 t40 t41 t42 t43 t44 t45 t46 t47 t48 t49 t50 - time
s0 s1 s2 s3 s4 - server
j0 j1 j2 j3 j4 - job

  (:init
    (= (currentTime) 0)
    (= (time t1n) -1)
    (= (time t0) 0)
    (= (time t1) 1)
    (= (time t2) 2)
    (= (time t3) 3)
    (= (time t4) 4)
    (= (time t5) 5)
    (= (time t6) 6)
    (= (time t7) 7)
    (= (time t8) 8)
    (= (time t9) 9)
    (= (time t10) 10)
    (= (time t11) 11)
    (= (time t12) 12)
    (= (time t13) 13)
    (= (time t14) 14)
    (= (time t15) 15)
    (= (time t16) 16)
    (= (time t17) 17)
    (= (time t18) 18)
    (= (time t19) 19)
    (= (time t20) 20)
    (= (time t21) 21)
    (= (time t22) 22)
    (= (time t23) 23)
    (= (time t24) 24)
    (= (time t25) 25)
    (= (time t26) 26)
    (= (time t27) 27)
    (= (time t28) 28)
    (= (time t29) 29)
    (= (time t30) 30)
    (= (time t31) 31)
    (= (time t32) 32)
    (= (time t33) 33)
    (= (time t34) 34)
    (= (time t35) 35)
    (= (time t36) 36)
    (= (time t37) 37)
    (= (time t38) 38)
    (= (time t39) 39)
    (= (time t40) 40)
    (= (time t41) 41)
    (= (time t42) 42)
    (= (time t43) 43)
  )

```

(= (time t44) 44)
(= (time t45) 45)
(= (time t46) 46)
(= (time t47) 47)
(= (time t48) 48)
(= (time t49) 49)
(= (time t50) 50)

(= (w s0 s0) 0.2)
(= (w s0 s1) 0.2)
(= (w s0 s2) 0.2)
(= (w s0 s3) 0.2)
(= (w s0 s4) 0.2)
(= (w s1 s0) 0.2)
(= (w s1 s1) 0.2)
(= (w s1 s2) 0.2)
(= (w s1 s3) 0.2)
(= (w s1 s4) 0.2)
(= (w s2 s0) 0.2)
(= (w s2 s1) 0.2)
(= (w s2 s2) 0.2)
(= (w s2 s3) 0.2)
(= (w s2 s4) 0.2)
(= (w s3 s0) 0.2)
(= (w s3 s1) 0.2)
(= (w s3 s2) 0.2)
(= (w s3 s3) 0.2)
(= (w s3 s4) 0.2)
(= (w s4 s0) 0.2)
(= (w s4 s1) 0.2)
(= (w s4 s2) 0.2)
(= (w s4 s3) 0.2)
(= (w s4 s4) 0.2)

(= (cCurveDt1 s0 s0) 1)
(= (cCurveDt1 s0 s1) 1)
(= (cCurveDt1 s0 s2) 1)
(= (cCurveDt1 s0 s3) 1)
(= (cCurveDt1 s0 s4) 1)
(= (cCurveDt1 s1 s0) 1)
(= (cCurveDt1 s1 s1) 1)
(= (cCurveDt1 s1 s2) 1)
(= (cCurveDt1 s1 s3) 1)
(= (cCurveDt1 s1 s4) 1)
(= (cCurveDt1 s2 s0) 1)
(= (cCurveDt1 s2 s1) 1)
(= (cCurveDt1 s2 s2) 1)
(= (cCurveDt1 s2 s3) 1)
(= (cCurveDt1 s2 s4) 1)
(= (cCurveDt1 s3 s0) 1)
(= (cCurveDt1 s3 s1) 1)
(= (cCurveDt1 s3 s2) 1)
(= (cCurveDt1 s3 s3) 1)
(= (cCurveDt1 s3 s4) 1)
(= (cCurveDt1 s4 s0) 1)

(= (cCurveDt1 s4 s1) 1)
(= (cCurveDt1 s4 s2) 1)
(= (cCurveDt1 s4 s3) 1)
(= (cCurveDt1 s4 s4) 1)
(= (cCurveDt0 s0 s0) 1)
(= (cCurveDt0 s0 s1) 1)
(= (cCurveDt0 s0 s2) 1)
(= (cCurveDt0 s0 s3) 1)
(= (cCurveDt0 s0 s4) 1)
(= (cCurveDt0 s1 s0) 1)
(= (cCurveDt0 s1 s1) 1)
(= (cCurveDt0 s1 s2) 1)
(= (cCurveDt0 s1 s3) 1)
(= (cCurveDt0 s1 s4) 1)
(= (cCurveDt0 s2 s0) 1)
(= (cCurveDt0 s2 s1) 1)
(= (cCurveDt0 s2 s2) 1)
(= (cCurveDt0 s2 s3) 1)
(= (cCurveDt0 s2 s4) 1)
(= (cCurveDt0 s3 s0) 1)
(= (cCurveDt0 s3 s1) 1)
(= (cCurveDt0 s3 s2) 1)
(= (cCurveDt0 s3 s3) 1)
(= (cCurveDt0 s3 s4) 1)
(= (cCurveDt0 s4 s0) 1)
(= (cCurveDt0 s4 s1) 1)
(= (cCurveDt0 s4 s2) 1)
(= (cCurveDt0 s4 s3) 1)
(= (cCurveDt0 s4 s4) 1)

(= (temperatureIn s0 t1n) 0)
(= (temperatureIn s1 t1n) 0)
(= (temperatureIn s2 t1n) 0)
(= (temperatureIn s3 t1n) 0)
(= (temperatureIn s4 t1n) 0)

(= (temperatureOut s0 t1n) 0)
(= (temperatureOut s1 t1n) 0)
(= (temperatureOut s2 t1n) 0)
(= (temperatureOut s3 t1n) 0)
(= (temperatureOut s4 t1n) 0)

(= (temperatureIncrease s0) 0)
(= (temperatureIncrease s1) 0)
(= (temperatureIncrease s2) 0)
(= (temperatureIncrease s3) 0)
(= (temperatureIncrease s4) 0)

(= (maxTemperature s0) 1)
(= (maxTemperature s1) 1)
(= (maxTemperature s2) 1)
(= (maxTemperature s3) 1)
(= (maxTemperature s4) 1)

(= (jobHeat j0) 1)

```

(= (jobHeat j1) 1)
(= (jobHeat j2) 1)
(= (jobHeat j3) 1)
(= (jobHeat j4) 1)

(= (jobDuration j0) 10)
(= (jobDuration j1) 10)
(= (jobDuration j2) 10)
(= (jobDuration j3) 10)
(= (jobDuration j4) 10)

(updatedOutletTemperatures t1n)
)
(:constraints
  (forall (?s - server ?t - time)
    (< (temperatureIn ?s ?t) (maxTemperature ?s))
  )
)
(:goal
  (and
    (forall (?j - job)
      (jobCompleted ?j)
    )
    (= (currentTime) 50)
  )
)
)

```

APPENDIX D

BLOCKS WORLD PDDL DOMAIN FILE

The PDDL domain for BlocksWorld used in the planning competitions is shown below.

```
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
               (ontable ?x)
               (clear ?x)
               (handempty)
               (holding ?x)
               )

  (:action pick-up
    :parameters (?x)
    :precondition (and (clear ?x) (ontable ?x) (handempty))
    :effect
    (and (not (ontable ?x))
          (not (clear ?x))
          (not (handempty))
          (holding ?x)))

  (:action put-down
    :parameters (?x)
    :precondition (holding ?x)
    :effect
    (and (not (holding ?x))
          (clear ?x)
          (handempty)
          (ontable ?x)))

  (:action stack
    :parameters (?x ?y)
    :precondition (and (holding ?x) (clear ?y))
    :effect
    (and (not (holding ?x))
          (not (clear ?y))
          (clear ?x)
          (handempty)
          (on ?x ?y)))

  (:action unstack
    :parameters (?x ?y)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect
    (and (holding ?x)
          (clear ?y)
          (not (clear ?x))
          (not (handempty))
          (not (on ?x ?y))))))
```