We can only see a short distance ahead, but we can see plenty there that needs to be done.

– Alan Turing.

University of Alberta

Refinement Implemented As Planning Operators

by

Bruce Allen Matichuk ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2008

Your file Votre référence
ISBN: 978-0-494-47302-3
Our file Notre référence
ISBN: 978-0-494-47302-3

Dedication

To my wife and my family,
Thank you for letting me pursue my dreams.

Abstract

Programming is a difficult task that very few humans can perform, and yet it is arguably the most important and broadly influential technology in society today. The challenge of Software Engineering is to make programmers more effective. Promising advances in AI Planning and program comprehension provide tools to address the challenge. This research demonstrates the use of Program Refinement laws to construct planning operators in a code planning domain. The purpose of this domain is to automate code generation (also referred to as code synthesis.) Using dynamically generated plans as programs, these methods can assist with the task of Software Engineering by providing tools for model checking and automated programming. The goal of the research is to provide a framework that will enable programmers and system designers to build larger and more complex systems than possible with the current approaches to program construction. The proposed technique can apply to low-level code or it can also apply to high-level code constructs used in the areas of workflow validation and automated service assembly. The dynamic nature of this method also enables the creation of a new type of software that makes decisions about how to behave at run time without human intervention. The research demonstrates the viability of the technique for small programs and programs without looping. One of the significant barriers to this technology's adoption is the difficulty in translating specifications into planning statements.

Acknowledgements

I would like to thank my friend Julian Fogel for teaching me about logic and for encouraging me to think clearly.

Thank you Dr. Hoover for taking me on as a project, my ambitions are not always practical and you have helped me stay grounded. And thank you to Dr. Eleni Stroulia and Dr. Paul Sorenson for agreeing to work with me on the CELLEST lab from which this journey began.

I owe a debt of gratitude to my wife, Allison, who has always been my rock; to my sons Nathan, Daniel, Isaac and Paul, for their love; and to my large and wonderful family for understanding. And thank you to my parents who raised me to believe I can do anything.

Table of Contents

1 Introduction **1**
1.1 Why Building Software is Hard 2
1.2 The Application of AI to Software Engineering 3
1.3 Related Work 3
1.3.1 Recording Knowledge: Programming by Example 4
1.3.2 The Programmer's Apprentice 5

2 Planning Overview **6**
2.1 Historical Overview of Propositional Planning 7
2.2 Alternatives to AI Planning 8
2.3 Plan Domain Description Language (PDDL) 9
2.3.1 History of PDDL 9
2.3.2 PDDL Basics 10
2.4 Graphplan 13
2.4.1 Graphplan Algorithm 14
2.5 Rocket Planning Example 15

3 Refinement Calculus **17**
3.1 Proving Code Correctness Relative to Starting and Ending States 17
3.2 The Code-Proof Isomorphism 18
3.3 Morgan's Refinement System 18
3.3.1 Morgan's Refinement Laws 20

4 Using AI Planning for Automated Refinement **22**
4.1 Description of Refinement Operations 22
4.2 Recursive Predicates 23
4.3 Description of Predicates 24
4.4 PDDL Example: Sequential Composition 26
4.5 Manipulation of the Syntax Tree 27
4.6 Swapping Variables Program Specification 35
4.7 Looping and Complex Algorithms 40

5 Experimental Results **47**
5.1 Sequential Composition Test 47
5.2 Swapping Variable Test 48
5.3 Selecting the Minimum from a list 51

6 Analysis **54**
6.1 Sort Induction 54
6.2 Application to Workflows and Service Assembly 56
6.2.1 Service Assembly 57
6.3 Agent Architecture with Planning 58

7 Summary and Future Directions **60**
7.1 Debugging Plans 60
7.2 Round Trip Modeling and the Complete Program Representation 60
7.3 Future Research 61

8 Conclusions **63**

Bibliography **64**

A Appendix A - Glossary of Terms **69**

B Appendix B - A Direct Encoding of Morgan's Laws. **72**

C Appendix C - Minimum of List Domain **83**

Chapter 1

Introduction

Software Engineering is the science and discipline of building, testing and maintaining software systems. According to the IEEE glossary of terms, "The discipline of software engineering encompasses knowledge, tools, and methods for defining software requirements, and performing software design, computer programming, user interface design, software testing, and software maintenance tasks". We live in the information age and computers dominate all areas of our life. From entertainment, to commerce, to mundane every day office tasks, everyone is affected by computers and the software that runs them. Despite the ubiquity and importance of software, programming remains a slow and time consuming human effort performed by unique individuals with specific talents and training. One of the goals of the discipline of Software Engineering is to increase the efficiency and effectiveness of this rare resource. However, the general method of programming has not changed much since Ada Lovelace developed the first program for Babbage's Analytical Engine. Programmers still painstakingly craft each line of code. This research project advances the field of Software Engineering by applying AI to the task of programming.

Large commercial software projects are notoriously complex to plan and to schedule. Most projects fail or are abandoned prior to completion generally resulting in significant financial costs to project sponsors. Faulty software can also be very expensive because many high value business processes rely on flawless execution. Increasing the reliability of the software engineering process both in terms of predicting time and effort and in the quality of the product requires a new approach that makes use of the ability of computer systems to assist in the reasoning process. As has been shown in other fields such as game playing, optimization, and machine learning, currently available hardware and software is sufficient to perform tasks that only a few years ago were exclusively in the domain of human reasoning. Software Engineering is a new frontier for AI and poses many challenges that will test our ability to build intelligent systems. The core methods of code refinement (Caroll Morgan) and program proofs have been well researched in Software Engineering as part of the field of software verification.

I will show that AI Planning can, within limits, be used to implement Morgan's Refinement rules to automate code synthesis. This is accomplished by encoding the state formulas of Morgan's

Refinement language as predicates within the Plan Domain Description Language (PDDL), the language of AI Planning. Each refinement rule is encoded as a planning operator with pre-conditions and post-conditions representing Morgan's refinement rule conditions and transformation effects. The method is applied to the problem of swapping two variables and planning is demonstrated to work using a planner that supports PDDL. The method is limited because some of Morgan's rules require processing beyond the sequencing analysis available with just AI Planning. One of the biggest challenges in code generation is handling iteration. The refinement method can be used for generating iteration code by assuming that universal quantification requires looping over a set a values and that existential quantification requires a variable declaration. Given these assumptions I am able to show that it is possible to implement refinement for some simple looping problems including finding the minimum of a list and sorting. In this document I introduce the methods of refinement and planning followed by a detailed specification of automated refinement using AI Planning.

1.1 Why Building Software is Hard

Software is difficult because it involves the assembly of large numbers of interacting components that form a complex system. Even small applications often involve thousands or millions of individual artifacts that must work together in perfect synchrony. Any imperfection is likely to lead to complete failure. Modern software engineering tools are built to address these issues by assisting in the construction of code, the organization of system components, and in alleviating repetitive manual tasks like building and testing software.

Although major improvements have been made to language design, editors, debuggers, interface design tools, testing tools etc, limited progress has been made in the development of methods that can systematically evaluate code or assist in the actual coding process by ordering code statements. The limitation is due to the computational limits of modern cpu's and memory. Code evaluation methods that are based on full state coverage must evaluate impossibly large state spaces rendering such techniques useless except for extremely small worlds. Best practice model checking therefore relies on state minimization where a user trims away parts of the state space that need not be considered until all that is left is a small tractable problem. These tools and techniques are complex and applicable to a narrow range of computing problems, so they are not practical in general.

application of AI

1.2 The Application of AI to Software Engineering

Artificial Intelligence (AI) is the study of automating intelligent behavior i.e. behavior that is normally associated with human beings. Surprisingly, there is very little research that incorporates emerging trends in AI into the software engineering discipline.

The field of AI has many contributing topics including natural language understanding, machine learning, automated reasoning, planning, pattern recognition, robotics, game playing, theorem proving, etc. Despite the variety of AI topics there are, nevertheless, two general categories of reasoning that encompass all these topics: thinking about what is in the world (noun reasoning), and deciding how to act (verb or action reasoning). One method of reasoning about actions is known as "AI Planning" which is applicable to software engineering because software is essentially a collection of statements, each of which determines the "action" of a program at runtime. Automated reasoning about program actions, based on design specifications, can be used to verify that a program contains correctly sequenced actions or even to generate a program that meets the specification requirements.

Code generation using AI Planning as the statement sequencing algorithm requires encoding specifications in a planning language like Plan Domain Description Language (PDDL). Unfortunately, PDDL is not very expressive which makes building complex specifications difficult. This thesis explores the capabilities and limitations of PDDL for encoding specifications and the use of AI planning to generate code. The first section provides an overview of planning and provides a detailed look at the Graphplan algorithm, one of the most successful planning methods developed so far and foundational to other later techniques. An overview of the Plan Domain Description Language is provided with a few simple examples. An overview of one method of reasoning about software known as "Refinement Calculus" is provided followed by a detailed review of some of Carroll Morgan's Refinement Laws [60]. The next section shows how refinement calculus can be compared to AI Planning and made executable. This section shows how some of the key laws Morgan developed can be translated into PDDL and demonstrates the application with a fully automated derivation of a variable swapping program using the domain with MetricFF, an award winning PDDL planner that is in the public domain. The following section details the application of this technique in the area of business workflow. The final sections summarize the results, discuss future research and provide final conclusions.

1.3 Related Work

In his seminal work titled "Assigning Meaning to Programs" [11], Robert Floyd describes a method for reasoning about programs. His techniques were later developed into what has become the Floyd-Hoare [14] axiomatic method for code analysis. Although these methods are precise, they address individual code statements and are incomplete for reasoning about architectural issues. An extensive body of work relates to software model checking. Approaches range from tools such as the eclipse plug-in outlined by Beyer et al. in [1] to precise logical methods such as the constraint technique outlined by Flanagan in [2]. In all of these approaches, individual statements within code are mapped to rules about program states. The problem with this approach is that the declarative representation of the program becomes too complex for people to use as pointed out by Beyer et al. in [1].

UML already allows for representing processes with pre and post conditions associated with process steps. Object Constraint Language (OCL) has also been added to UML [10] to enable the specification of "guards" and "verification tests". But in both these cases, the semantics of the language do not allow for automated "assembly" of sequences of actions into a program. In Mellor et al. [10] a proposal is offered for extending UML with action semantics. The research in this paper suggests an approach for defining programs that can be assembled which cannot be captured within the current UML specification or within any UML extension.

More closely related to the suggested approach in this paper is the work by Wu et al. [3] in which the authors describe a system for automatically generating service descriptions using the JSHOP2[8][9] planner. There are direct commercial benefits to developing systems that reason over service descriptions for searching and for derivation. A new initiative referred to as the Executable Semantic Web has been launched by OASIS [4] covering industry standards for defining and building commercially viable intelligent agents that define their behavior at runtime using automated reasoning. The knowledge engineering standards suggested as an initial starting point are based on the Semantic Web [21] emerging from research in Description Logic [20] and Ontology research [5]. Most of the work in this area has concentrated on knowledge representation required for agent invocation, which is a critical component of intelligent systems architecture but does not address the issue of agent execution.

1.3.1 Recording Knowledge: Programming by Example

Conventional workflow engineering involves time consuming and difficult analysis of many activities to determine the exact nature of how required tasks fit together; however, it is possible to observe and use task behavior to infer task pre-conditions and effects. This technique has been fully developed and applied to task recording within the Legacy System Terminal Data Stream environment [45], [48], [35]. In this research a recorder was constructed that can observe the activities of a user working on a Legacy System. The task information, along with the instructions required to perform the task, are stored in a knowledge repository. Task pre-conditions and effects are derived from the observed expert system user's ordering of tasks. Propositions are generated in such a way that the observed ordering can be regenerated.

The use of trace mining to derive specifications can be applied to the problem of programming by example. In programming by example, a user demonstrates examples of what they would like to write a program to do. For example, instead of writing a sort routine, a user could demonstrate manual sort processes. By mapping the demonstrated operations to individual components of code, its possible to generate a model of how the program should work. The model can then be used to generate code.

There are many applications for a model based approach with automated reasoning applied to software engineering. Advantage include better code management and maintenance, the ability to use intelligent systems to help derive difficult code, and the ability to learn models of system behavior to use in the code generation process. This thesis investigates model driven development based on AI Planning and program refinement methods.

1.3.2 The Programmer's Apprentice

The main objective of building an automated synthesis tool is to assist a user implementing code to some specfification. Ideally, a code synthesis system would be able to examine code up to some point within a system and make suggestions about next possible coding steps. In the mid 1980's, Charles Rich and Richard Waters at MIT developed the Programmers Workbench[67] as a programmers help tool with the objective of understanding how "expert programmers analyze, synthesize, modify, explain, specify, verify, and document" programs. According to the authors their research spans both artificial intelligence and software engineering. A series of tools were developed incorporating automated reasoning and knowledge management to enable limited code analysis and code synthesis. A central component of the architecture is their "Plan Calculus" which is a formal framework for representing programs. The basic functions of the tool include information gathering, design and implementation of code.

One of the most difficult aspect of code analysis and programming is handling loops. In their concluding section (p. 28), Rich and Waters state that, "loops are generally acknowledged as the hardest parts of a program to understand". To generate code with loops they employ what they refer to as "Synchronizable Series Expressions". These are similar to the logic "macros" developed in this research for generating code with refinement.

The Programmer's Apprentice research examines some of the key issues central to building tools that enable AI to be used for assisting with Software Engineering. Unfortunately, the knowledge representation methods, reasoning methods, CPU speed and memory available at the time the research was completed severely limited the application of AI techniques. With progress on all of these fronts a newer version of the Programmer's Apprentice would likely achieve better results.

Chapter 2

Planning Overview

Artificial Intelligence (AI) Planning and Scheduling is a subject area within computing science concerned with reasoning about activities and sequencing them [55]. Action sequences constitute plans that, if executed, represent a means of achieving a goal. The focus of planning research is on the representation of actions, storage and retrieval of actions, time conflict, resources, priorities, action reasoning, time and space complexity, proving correctness, proving solvability (within a reasonable amount of time), knowledge representation, problem complexity and scale, real-world applications, uncertainty, model coverage and plan domain learning. Scheduling is a sub-field of planning that deals specifically with planning resource allocations over time to achieve specific goals within minimal time frames and with minimal resource allocation.

Plan systems are steadily improving to the point where practical real-world problems can be expressed as plan domains and quickly solved with available planning systems. In 1994, the classic "Blocks World" problem was difficult - solutions for problems with 10 blocks would require hours of solution time. At AIPS 2000 [19], most planners were able to handle more than 100 blocks, and the winning planner of the plan competition was able to solve planning problems with 300 blocks in under a few seconds. AI planning has proven to be superior for action ordering over other forms of automated reasoning such as logic programming, SAT solvers or general constraint solvers. While constraint solvers have achieved great commercial success, it is still difficult to encode planning problems as constraint problems [30].

Planning methods have become popular for describing and solving a broad range of problems including: Intelligent Robotic Systems, Problem Solvers (Theorem Proving, Games), Automated Program Generators, Natural Language Processors, Expert Systems, Service Assembly, Computer Integrated Manufacturing, Distributed Agents, Scheduling and much more. Planning, however, has not yet become a mainstream Software Engineering method for solving problems. Planning uses a declarative representation of program behavior, which is not taught in standard programming classes. There are no commercially successful AI Planning tools, and, in the past, planners had a very narrow range of application. However, recent advancements in planning algorithms and in CPU and space availability have prompted interest in planning architectures. There have been several recent papers published addressing the use of AI Planning for service assembly in popular engineering journals indicating interest from other fields outside of the AI Planning community.

There are two distinct types of planning systems: Propositional [42] and Decision Theoretic [22]. Propositional planning deals with the world in terms of actions that are composed of propositional statements about action pre-condition and effects. The "Rocket World" problem described later in this thesis is an example of a propositional planning problem. Decision Theory combines Probability Theory with Utility Theory, and treats the world as a set of states over which an action policy determines likely transitions between states. Decision Theoretic planning involves the calculations of probabilities, and does not provide mechanisms for demand-time goal directed behavior. Rather, Decision Theoretic systems attempt to pre-compute the likelihood that some action in response to some observation will lead to a situation that will ultimately allow a goal to be achieved. The goal and the policy are related so that changing a goal requires re-computing a policy. This can be prohibitively expensive computationally and so this technique is not practical for situations where goals constantly change. While Propositional Planning determines a path to a goal by chaining together logical propositions about how the world works, Decision Theoretic planning determines a path to a goal by calculating probabilities of expected outcomes and choosing the most likely course of action which will lead to a specific goal. There have been some attempts to merge these two fields, but generally they represent two completely distinct areas of research. The focus on this research is on Propositional Planning. But the general problem of code synthesis has not been solved, so further progress in this area may require decision theoretic planning approaches.

2.1 Historical Overview of Propositional Planning

AI planning is closely related to state-space search, problem reduction and means-end analysis used in problem solving, and to other areas of AI such as theorem proving and situation calculus. The first major planning system was called STRIPS (Stanford Research Institute Planning and Scheduling) [28], and was the planning component of the Shakey robot project at SRI. A series of planners have emerged since then, addressing many important problems in AI planning.

The initial STRIPS implementation used simplistic greedy search algorithms to find plans; however, the simple approach proved ineffective. Using a simple approach, non-interleaved plans cannot find goals where sub-goals must be protected halfway through a plan. The Sussman Anomaly is a classical planning problem that illustrates the nuances of interleaving (see HACKER [51]. WARPLAN [54] and INTERPLAN [52] introduced solutions to the interleaving problem.)

Task networks allow plans to be partially ordered, i.e. not all sequences of actions need to be ordered in a valid plan. NOAH [47] and NONLIN [53] investigate this idea. Chapman provides some of the early analysis in planning using TWEAK [26]. Chapman's work included detailed analysis, proofs of completeness and intractability of various formulations of the planning problem. SIPE [56] introduces the notion of Hierarchical Task Networks, which allow for scaling plans by dividing plans into many subplans that are solved individually. Russel and Norvig [46], describe POP, which is based on SNLP [50].

More recently, several advances in AI planning have made planners more commercially viable. Early planners suffered from memory use and run-time problems. Graphplan [23] introduces an algorithm that guarantees a solution (if one exists) within polynomial time and space (see Glossary for polynomial time), and is an early example of a disjunctive planner. A disjunctive planner retains the current planset without splitting its components into different search branches. SATPLAN [33] and BLACKBOX [34] employ this idea to generate constraints, which are solved using a constraint solver. TLPlan [18] is one of the most successful planners in terms of memory efficiency and speed. TLPlan employs heuristic search, which has been found to work well in the area of problem solving. Other planners like TALPlanner [36], Fast-Forward [31] and SHOP [41] also use heuristic search.

Heuristic search techniques have been shown to be quite fast, and are the basis for many interesting problem solvers such as game systems. Recent advances in both heuristic search and in planning have lead to breakthroughs in AI Planning. Systems such as FF from Jeorg Hoffman and later derivatives of FF use a relaxed graph to quickly generate a plan length which is an admissible heuristic for A* search.

Recently planners have seen steady improvements towards commercial viability. The introduction of planning and scheduling competitions citeMCD1 has had a positive impact on the quality and reliability of planning systems. Current planning research is focused on time and space concerns, and knowledge representation issues have been of limited interest [56]. STRIPS introduced a simple notation that was useful for describing plan domains and planning problems, but allowed description of very limited worlds. Later, the Action Description Logic (ADL) [43] introduced by Pednault in 1986 was the first attempt to extend the very limited semantics of STRIPS. PEDESTAL [40] was the first partial implementation of ADL, while a complete implementation was found in UCPOP [44]. To facilitate planning and scheduling competitions, a new language was introduced that incorporated the success of ADL with other advances in planning. A committee from AIPS chaired by Drew McDermott developed Plan Domain Description Language (PDDL). Most modern planners use PDDL as a front-end input and testing language, greatly facilitating inter-operability. A standard test set of plans is maintained on various conference based web sites hosted by AIPS and ICAPS and is available on the Internet.

2.2 Alternatives to AI Planning

The only other viable alternatives to AI Planning to address the problem of refinement are constraint programming and logic programming. While logic programming using languages such as Prolog is a well developed field, the problem with logic programming is that there is not enough knowledge about action reasoning built into the solver to enable rapid reasoning for action oriented problems. Constraint programming is another viable approach to refinement; however, constraint programming languages are much more difficult to use than AI Planning [30]. In both planning and constraint programming, the solution to a problem is determined automatically by declaring the description of a knowledge domain and some problem to solve within that domain. In constraint programming, the world is defined using a set of variables and a set of constraints between those variables. Constraint solvers are specialized programs that are able to examine the constraints defined within a system, and then automatically derive a set of variable assignments that match the constraints. While powerful, constraint systems on their own do not have any specific knowledge of action reasoning problems; however, knowledge of action reasoning issues such as mutually exclusive conditions can be added to a constraint domain and thus provide guidance for action reasoning using a constraint reasoning system.

2.3 Plan Domain Description Language (PDDL)

2.3.1 History of PDDL

The Plan Domain Description Language was initially developed by Drew McDermot to compare planners. Prior to the development of PDDL, every planner would use its own domain and problem representation. This made comparison difficult because of the effort involved with translating between representations. In the late 90's when AIPS was formed, researchers decided to compare planners using a tournament format thus requiring a common language. By encoding all the tournament problems in PDDL, it was a simple process to run each planner on the presented problem and compare the run-time results.

In order to achieve its initial goal of planner interoperability, the design of PDDL did not accommodate all of the features of all available planning systems. Instead, a small subset of popular features was encoded within the language. Although this decision resulted in early criticism of the language, it was nevertheless effective in allowing the release of a functional unifying representative language. Shortly after the creation of PDDL 1.0 other releases followed. The most recent release of PDDL is version 3.1 (`http://ipc.informatik.uni-freiburg.de/PddlExtension`) and it includes many advanced planning features.

PDDL syntax and semantics is largely based on STRIPS. STRIPS has conditions and operators and planning using initial conditions and goals. Preconditions are represented as truth assignments to named condition variables, and post-conditions are represented as an add list and a delete list. In STRIPS planning, the effects of an operator are to add or delete predicate formulas from the planning world. PDDL dispenses with the add/delete terminology and instead simply refers to the truth value of a predicate. So stating a predicate in the effect is the same as "add" in STRIPS. Stating the negation of a predicate is the same as a "delete" in STRIPS.

2.3.2 PDDL Basics

PDDL planning requires the creation of a plan domain file and the creation of a problem description file. The problem description encodes the state of all initial variables, the objects available to the planner, and the goals that must be achieved. The domain file contains the types, the axioms and the operators.

PDDL is designed to allow a planner to decide whether or not it can work with a given problem. In planning there are various types of planning features required to work with a given domain. These features must be listed at the top of the description. The domain can then include types, predicates, functions, operators and axioms.

The first line is the name of the domain.

```
(define (domain logistics-adl)
```

The next line lists the required features of the domain. Not every planner can work with every planning feature. So if a plan requires ADL extensions and a particular planner does not allow them, then the planner will need to abort when it sees the requirements definition. The next two lines show the declaration for a domain that requires ADL extensions and domain axioms.

```
(:requirements :adl :domain-axioms)
```

The next line can list constants. These are objects that can be globally referenced without declaration on the parameter line of a parameter. Constants are useful because they reduce the complexity of operators. Not all planners allow constants though. The FF planner, for example, does not. So all referenced objects must be on the parameter line.

```
(:constants left right - gripper)
```

Types are simply names and can be declared in a hierarchy. In order to use types, a domain must declare "types" as a features of the domain. The use of types is important because they constrain the search for applicable operator. For example, if there are a hundred objects in a planning problem but only two of them are "vehicle" objects, then an operator that has this type in its parameter list only needs to consider two alternatives instead of a hundred. The time savings become much more significant when you have planning operators with several parameters. The search space grows exponentially with each additional parameter.

```
(:types physobj - object
    obj vehicle - phssobj
    truck airplane - vehicle
    location city - object
    airport - location)
```

Predicates include a name followed by a list of required types.

```
(:predicates
    (at ?x - physobj ?l - location)
    (in ?x - obj ?t - vehicle)
    (in-city ?l - location ?c - city)
)
```

A domain can also use functions. These are parameterized numeric variables that can have values that vary throughout a plan. In order to use functions, a domain must declare "fluents" as a feature of the domain. Functions are similar to predicates except that they have a numeric value instead of a boolean value.

```
(:functions
        (fuel ?car - vehicle)
        (distance ?car - vehicle)
)
```

Some planners allow the declaration of metrics. Metrics allow planners to maximize or minimize certain values. A metric expression is composed of a set of "fluents". For example:

```
(:metric minimize (+ (* 5 (fuel car)) (distance package)))
```

Planning operators describe the operations that the planner can use to achieve goals. An operation consists of a name, parameters, preconditions, and effects. All expressions are bracketed. A predicate may be a pre-defined predicate or a declared predicate. To chain together a list of predicates, the list must be preceded with the "and" operation. The "not" operator can be applied to negate a predicate. Bracketed expressions are also permitted as well as an "or" operator for simplicity.

While not essential for planning, PDDL adds various syntactic extensions to planning such as conditional effects and quantified effects. While it is possible to "flatten" these expressions into sets of operators, these extension simplify working with large numbers of objects and operators. A conditional effect is an effect that is included with the operator application under a specific condition. In the drive-truck operator below there is a conditional effect that states "when an object x is in a truck then the effect of the drive-truck operation is that object x is not at loc-from and it's not at loc-to." In order to use conditional effects, the "conditional-effects" feature must be listed as a requirement for the domain. A quantified effect is one that iterates over all the objects of a given type. In the example below, the conditional effect (described above) of the drive-truck operator is over all the "obj" objects are types.

```
(:action drive-truck
    :parameters (
        ?truck - truck
        ?loc-from
        ?loc-to - location
        ?city - city)
:precondition
    (and
        (at ?truck ?loc-from)
        (in-city ?loc-from ?city)
        (in-city ?loc-to ?city))
:effect
    (and
        (at ?truck ?loc-to)
        (not (at ?truck ?loc-from))
        (forall (?x - obj)
        (when (and (in ?x ?truck))
              (and (not (at ?x ?loc-from))
                   (at ?x ?loc-to))))))
```

Once a planning domain is created, a set of one or more problem statement files can be created. A problem statement declares the objects of each type that are available for planning, the initial states of predicates and functions and the goal state of predicates and functions.

The first line establishes the problem name.

```
(define (problem strips-gripper2)
```

The next line indicates which domain the problem statement is for.

```
(:domain gripper-strips)
```

Now list the objects of the world.

```
(:objects rooma roomb ball1 ball2 left right)
```

The init line indicates the initial value of any listed functions.

```
(:init
(room rooma)
(room roomb)
(ball ball1)
(ball ball2)
(gripper left)
(gripper right)
(at-robby rooma)
(free left)
(free right)
(at ball1 rooma)
(at ball2 rooma))
```

The goal statement indicates the predicate and function values that must be achieved to reach a plan.

```
(:goal (at ball1 roomb)))
```

Some planners also allow the specification of axioms. These are global rules that describe relationships between propositions. To use axioms, the requirement of ":domain-axioms" must be specified in the domain description.

```
(:axiom
:vars (?x ?y   physob)
:context (on ?x ?y)
:implies (above ?x ?y)))
```

Its possible to use optimization with the planner by adding a minimization metric. On some tests a counter, (*steps*), was inserted into the operations to plan for a minimum number of steps. However, the objects that were chosen in the problem resulted in a minimal plan. To test the effectiveness of the counter there would need to be more objects specified in the problem statement than necessary. This test was not run. To use the metric, the following statement must be added to the problem statement.

```
(:metric minimize (steps)))
```

Other advanced features have been added to recent versions of the PDDL Language (PDDL 3.x) providing tools to handle temporal constraints derived predicates and much more. The language is evolving into more than just a planning language and may one day rival prolog as a general problem solving language for the development of intelligent systems. However, the planning features that were used in this research included only the basic features of PDDL 2.1.

PDDL is not a very expressive language relative to standard programming languages. Many features found in standard object oriented programming languages such as inheritance or polymorphism are missing from the language. PDDL syntax for conditions does not allow the description of standard programming structures such as trees, or lists, which makes planning difficult to work with because of the large effort involved with translating programming problems into planning problems. Several researchers have suggested various extensions to PDDL to address these limitations and this is an active area of research [58]. Despite these limitations, planning addresses the difficult problem of ordering actions that have pre and post conditions. For code synthesis, this is critical. Simplistic approaches such as logic programming or optimization that do not account for the specific problem of action ordering are ineffective. For this reason, planning is an ideal algorithm to use for code synthesis despite the limitation of PDDL as a planning language.

2.4 Graphplan

Graphplan is a popular planning algorithm that is the basis for several successful planners. When Graphplan was first introduced in 1995, it generated a lot of interest because of its ability to solve plans much faster than previous algorithms. Since Graphplan was introduced, several other algorithms have been developed that are equally competitive. Most new algorithms, however, continue to use the graphing concept of Graphplan to describe the relationship between propositions and actions. So Graphplan is a good starting point for understanding how planning works.

A planning graph consists of a set of levels. The first level is a set of facts (grounded predicates) that correspond to the initial conditions. The second level is a set of actions (applied operators) that can be performed given the previous level of facts. The subsequent level is a set of facts that would be true if the previous level actions were to be applied.

The Graphplan algorithm begins by taking the initial state facts and attempts to apply all possible actions with preconditions that match existing facts. For every fact, a No-Operation action is also applied. This results in a set of facts at the next level corresponding to all possible conditions in that level for any combination of applicable operations. The subsequent levels repeat this process until a set of facts are found that match the goal state, or until no new actions may be applied that impact the fact list.

The graph is used to record mutually exclusive constraints between actions and between facts.

Each pair of mutually exclusive actions or facts is called a mutex pair.

A mutex action pair occurs under three conditions: Inconsistent Effects, Interference and Competing Needs. - Inconsistent Effects occur when the effect of one action is the negation of another action's effects. - Interference occurs if either action deletes a precondition the other. - Competing Needs occurs if any preconditions for a pair of actions are marked as mutex.

A pair of propositions is marked as mutex if all ways of creating one proposition are exclusive of creating the other, or if one is the negation of the other.

An example of mutex actions in the rocket domain includes "Load A" and "Fly to the Moon". Clearly, one cannot fly to the moon at the same time that one is loading the rocket. "Fly to the Moon" deletes the precondition of "Load A", which is that A must be in the same location as the rocket. The propositions that "Rocket on the Moon" and "Rocket on the Earth" are clearly mutually exclusive and cannot exist at the same level in a plan.

Once a planning graph is constructed, a backward chaining algorithm is used to determine the best possible path backwards through the graph. In the paper titled "Fast Planning Through Planning Graph Analysis", Blum and Furst were able to prove that Graphplan will find a plan if one exists in polynomial time, or will detect that no solution exists within polynomial time. This result validated the applicability of planning for real-world applications because it shows that planning times will scale slowly with respect to domain sizes i.e. its not possible for planners to run forever without stopping.

2.4.1 Graphplan Algorithm

The following section describes the Graphplan algorithm in pseudo-code format.

```
Add initial conditions
Find out which ones are mutex
Repeat until done
    Check to see which actions can apply (no mutex preconditions)
    If no actions can be applied then a plan cannot be found
    Add all actions that apply (no mutex preconditions)
        given current conditions
    Find out which actions are mutex
    Discover new conditions given application of
        all possible actions
    Find out which conditions are mutex
    If current conditions meet goal conditions then
        Check to see if a plan can be found by backchaining to
            initial conditions
        Mutex actions cannot occur in the same time frame
            of a plan
    If a plan can be found then Graphplan is done
End repeat
End Graphplan
```

2.5 Rocket Planning Example

The term classical planning is often used to describe simple worlds with initial states, goal states and a set of deterministic actions. Although the general planning problem is much more complicated, understanding classical planning is an important conceptual step towards understanding general planning. The Rocket domain is a classical planning example that will help to illustrate the key points. The Rocket domain addresses the problem of transporting packages from earth to the moon using a single rocket [32]. In the example of this domain provided below there are three classic planning operations: Load, Unload, and Fly. There are two objects, A and B, and three locations, Earth, Moon and Rocket. The Load operation applied to a particular container has a pre-condition that the container is not loaded. The effect of the Load operation is that the container becomes Loaded. This problem has been a commonly used example since Veloso [57], and so it is reviewed here.

Note: 'A' refers to Package A and Load A means Load Package A. 'B' refers to Package B and Load B means Load Package B. 'R' refers to the rocket.

In planning language of PDDL, the rocket problem would be defined in the following way. (NOTE: this is not pseudo-code but an actual plan domain used for the planning problem described in Figure 1).

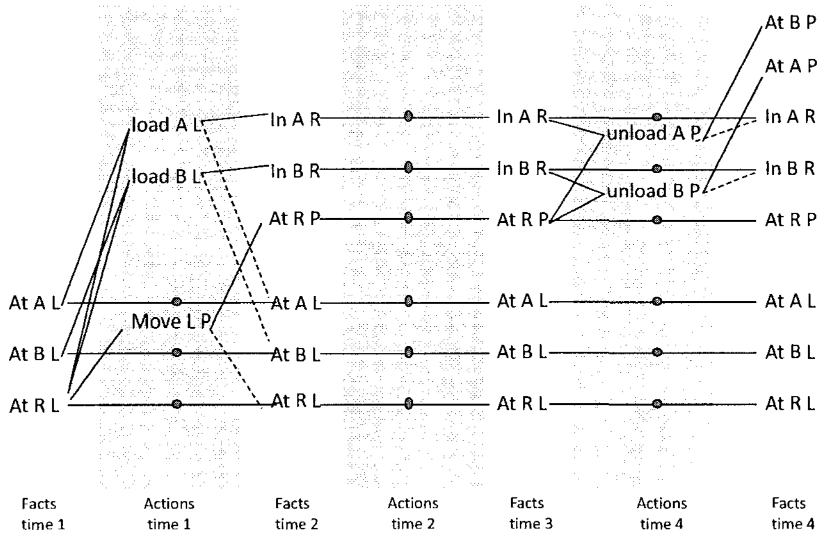
```
(define (domain rocket)
(:predicates
(at ?r - rocket ?from - place)
(in ?c- cargo ?r - rocket))
(:action move
:parameters ( ?r - rocket ?from ?to - place)
:precondition (and (at ?r ?from))
:effect (and (at ?r ?to) (not (at ?r ?from)))

(:action unload
:parameters (?r - rocket ?p - place ?c - cargo)
:precondition (and (at ?r ?p) (in ?c ?r))
:effect (and (not (in ?c ?r) (at ?c ?p)))

(:action load
:parameters (?r - rocket ?p - place ?c - cargo)
:precondition (and (at ?r ?p) (at ?c ?p) )
:effect (and (not (at ?c ?p)) (in ?c ?r) ))
```

Use the following syntax to make a request of the planner to find a plan:

```
(define (problem rocketProblem)
(:domain rocket)
(:objects A B - cargo E M - places R - rocket )
(:init (at A E) (at B E) (at R E))
(:goal (and (at A M) (at B M) )
```

In the graphics below the fact layers are interleaved with the operation layers. A solid line indicates an operation setting a predicate to true in its effect. A dotted line illustrates the negation

Figure 2.1: The Rocket Planning Example



of a predicate. A small circle is a "NOOP" which means that no change to the predicate allows the value of the predicate to persist in the next time layer. The Figure 2.1 graphic is a variation on Blum and Furst's [23] original Graphplan depiction of the problem.

In summary, AI planning is a specific method for reasoning about sequences of actions that are composed of logical pre-conditions and post-conditions. PDDL is a rich language that allows the specifications of the physics of a domain in a generic way consistent with multiple planning systems. The solution to a planning problem is plan which is a sequence of applicable operations proving that the initial states can be converted to goal states using a limited set of pre-defined operations.

The following chapter on Refinement Calculus similarly describes programs as sequences of operations that convert initial states to goal states. Subsequently, automated refinement as discussed in this paper adapts refinement calculus laws, converting them into planning operators allowing the application of AI Planning to the problem of generating code.

Chapter 3

Refinement Calculus

3.1 Proving Code Correctness Relative to Starting and Ending States

In Hoare's Oxford Distinguished Lecture speech marking the 25th anniversary of Hoare logic, Tony Hoare indicated that he felt that program comprehension is a "grand challenge" problem for computing science. Hoare was the first to develop a formal method of program comprehension using a logic which is now referred to as Hoare Logic. Program comprehension is difficult because fully understanding a program requires knowledge of the goals of the program and the reasoning capability to achieve those goals. Code synthesis is the reverse process of program comprehension in which the knowledge and reasoning ability required for program comprehension are used to generate code.

Program refinement, based on Hoare's Logic, is a promising technique for implementing code systhesis. The first study of program refinement was made by Ralph-Johan Back[65] in 1978. His Ph.D. thesis was entitled "On the Correctness of Refinement Steps in Program Development" in which he provided the basic development for sequencing, variable declaration and iteration. In a book entitled "Programming from Specification", originally published in 1990, Carroll Morgan compiles refinement techniques from various authors, including Back, King, Morris and Dijkstra, and develops the most useful treatment of the subject available by providing a set of laws with sufficient detail to allow for investigation into automated methods. A more formal evaluation of refinement can be found in Back's book[64] "Refinement Calculus: A Systematic Introduction".

Morgan's refinement methods consist of a set of "recipes" for demonstrating that a given program can be derived from a specification comprised of the initial states of a program and the final expected states. These "recipes" consist of a series of laws and rules for their application which can be converted into fully automated methods for code synthesis as demonstrated in the next chapter on "Using AI Planning for Automated Refinement".

3.2 The Code-Proof Isomorphism

The Curry-Howard isomorphism equates code with a proof by showing that logical type theory corresponds to computational type theory. This equivalence provides insight into methods for code synthesis because it shows that code comprehension is related to proof comprehension. A proof is a sequence of transformations of truth statements - truth is itself established by the rules of the logic in which the proof is provided. So to say that a statement is true, one means that it is either defined to be true or that it can be discovered to be true by tracing through a logic's transformation rule set i.e. the semantics of the logic. But these traces are simply equivalent to what code does when it is executing. Code performs a series of transformations on memory according to the semantic rules of of a programming language. A proof that a variable of a type can be converted to another will directly correspond to statements of a program that can accomplish this task [59].

Like the Curry-Howard Isomorphism, Program Refinment relates a proof to code. Morgan's refinement laws show that a proof demonstrating that a particular starting state will be converted to a certain output state by a particular set of actions, which is a program, will emit the code of exactly that program. AI planning is a kind of automated theorem proving technique that can be used to sequence state transformations and thus generate code based on Morgan's refinement laws.

3.3 Morgan's Refinement System

The universe inhabited by programs can be described using states. The states of a program are changed by applying functions. A state can be described as consisting of attributes that are represented as variables. Each variable has its own set of values. Changes to state occur as assignment operations which modify variable values. In the context of refinement, programs are often referred to as agents and state changes referred to as actions. Predefined sequences of actions are written as $S1; S2; S3$ etc. Choice among actions is written $S1 or S2$. So there is a simple language of states that can be written as

$S := f | S; S | S or S$

Both sequences and alternatives are associative, i.e. $(S1; S2); S3 = S1; (S2; S3)$ and $(S1 or S2) or S3 = S1 or (S2 or S3)$.

A program is a set of actions comprised of sequences and alternatives. A program specification is a statement about the initial conditions of a program and the end conditions. A program is considered "correct" relative to its specification if an application of the program to the initial conditions guarantees the end conditions. Initial conditions are sometimes referred to as pre-conditions and end conditions are sometimes referred to as post-conditions.

The concept of Refinement is based on Dijkstra's observation that the weakest precondition of a program, written as $wp(Program, Result)$ is the set of states from which the program 'Program' is guaranteed to deliver the set of states identified as 'Result'. If $pre \rightarrow wp(P, post)$ and pre is true, then running P must establish $post$. The operational semantics (i.e. the specific action set) of P are not important, any program will do. But whatever P is, it must guarantee $post$ in the context of pre. So any choice of action set is a refinement of a specification because it eliminates possible action set interpretations of the specification. Refinement is expressed as follows:

$$[pre, post] \sqsubseteq P \tag{3.1}$$

It should be noted that any refinement implicitly carries with it the pre and post conditions of the specification being refined. Also, a specification itself may be refined. There is a correspondence between specifications and programs but they should not be equated since a specification may be refined to many different programs, and a program may be a refinement of many different specifications.

Morgan further expands the syntax of refinement in [66] with the definitions below.

Definition 1:

$$P \sqsubseteq Q \iff \forall post\ wp(P, post) \Rightarrow wp(Q, post) \tag{3.2}$$

This defines the $\sqsubseteq$ symbol as the 'Refinement' operator. The refinement operator declares that P is refined by Q iff for all post-conditions $post$ the specific states for which P results in $post$ are also states for which Q results in $post$. The program P can be replaced by the program Q.

Definition 2:

$$wp([pre, post], R) = pre \wedge (\forall \vec{v}.post \Rightarrow R) \tag{3.3}$$

This defines a specification of a program as $[pre, post]$ with precondition pre and post-condition $post$. The vector $\vec{v}$ are all the variables of the program.

Definition 3:

$$wp(\vec{w} : [pre, post], R) = pre \wedge (\forall \vec{w}.post \Rightarrow R) \tag{3.4}$$

This defines the frame variables $\vec{w}$ of a specification as only those that will change within the program defined by the specification.

For example: if $\vec{v}$ is "x,y" then

$wp(x : [true, x = y], R)$

$= true \wedge (\forall x.x = y \Rightarrow R)$

$= R[x\backslash y]$.

Note that the weakest precondition for assignment is similar: $wp(x := y, R) = R[x\backslash y]$. This shows that logical equation as a post condition of a specification where only x can change is the same as a program that assigns a value to x i.e. "x:[true,x=y]" has the same meaning as "x:=y".

Definition 4:

$$wp(\vec{w} : [pre, post], R) = pre \wedge (\forall \vec{w}.post \Rightarrow R[\vec{v_0}\backslash\vec{v}]) \tag{3.5}$$

This defines the 0-subscripted variables as the initial values of the same variables without the subscript. Note that if *post* does not refer to initial variables then definition 4 is the same as definition 3.

Definition 5:

$$\vec{w} : [post] \text{ abbreviates } w : [(\exists \vec{w}.post)[\vec{v_0} \backslash \vec{v}], post] \tag{3.6}$$

This simply means that if $\vec{w} : [post]$ then "it is possible to establish the post-condition".

Definition 6:

$$x : \lhd E \text{ abbreviates } x : [x \lhd E[\vec{x} \backslash \vec{x_0}]] \tag{3.7}$$

This definition extends the notion of syntactic replacement to any binary relation. Remember that "$x := y$" means that "$x : [x = y]$". In this definition, the "$=$" is generalized to "$\lhd$" which is any binary relation. For example:

$x :< x$ means that x is decreased

$m :\in s$ means that m is chosen from the set s.

3.3.1 Morgan's Refinement Laws

Based on the above definitions, Morgan defines a set of laws that can be used systematically in developing refinement over specifications.

Law 1.1: **Strengthen postcondition** If $post' \Rightarrow post$ then

$$w : [pre, post] \sqsubseteq w : [pre, post']. \tag{3.8}$$

So, for example, suppose a specification is given that says, "build a program for getting the variable x to a number that is greater than 2." The post condition of the program will be x¿2. The post condition can be strengthened to x¿4 because x¿4 implies that x¿2.

Law 1.2: **Weaken preconition** If $pre \Rightarrow pre'$ then

$$w : [pre, post] \sqsubseteq w : [pre' post]. \tag{3.9}$$

Law 1.3: **Assignment** If $pre \Rightarrow post[w \backslash E]$, then

$$w, x : [pre, post] \sqsubseteq w := E. \tag{3.10}$$

The "\" symbol refers to syntactic replacement. In the formula $post[w \backslash E]$ all occurances of w are replaced by E.

Law 3.2: **Introduce local block** If w and x are disjoint, then

$$w : [pre, post] \sqsubseteq |[\textbf{var } x : T; \textbf{and } inv \bullet w, x : [pre, post]]| \tag{3.11}$$

This law states that a variable can be added to the frame if it is declared.

Law 4.1: **Skip command** If $pre \Rrightarrow post$, then

$$w : [pre, post] \sqsubseteq skip. \tag{3.12}$$

Law 4.2: **Sequential composition** For any formula *mid*,

$$w : [pre, post] \sqsubseteq w : [pre, mid]; w : [mid, post]. \tag{3.13}$$

This states that any formula can be inserted between two formulas defining a specification; the result is two specifications where one is in sequence with the other.

Law 4.3: **Skip composition** For any program prog,

$$prog; skip = skip; prog = prog. \tag{3.14}$$

This states that the statement **skip** can simply be removed from a program.

Law 4.4: **Following assignment** For any term E,

$$w, x : [pre, post] \sqsubseteq w, x : [pre, post[x \backslash E]]; w := E. \tag{3.15}$$

This law combines **assignment** with **sequential composition**. It is used in the "Swapping Variables" example in a later section.

Law 5.1: **Alternation** if $pre \Rrightarrow (\bigvee i \bullet G_i)$, then

$$w : [pre, post] \sqsubseteq \textbf{if}([]i \bullet G_i \rightarrow w : [pre \wedge G_i, post])\textbf{fi} \tag{3.16}$$

This law states that if a pre condition implies a set of guard expressions, then the program refinement is a conditional expression that includes each possible guard such that if the guard is true then a code is run represented by a specification consisting of the original specification but with the guard added to the precondition.

For example: $m : [true, m = maximum(a, b)]$ would resovle to if $a \geq b \rightarrow m : [a \geq b, m = maximum(a, b]$ $[]b \geq a \rightarrow m : [b \geq a, m = maximum(a, b)]$ fi.

Law 7.1: **Iteration** Let *inv*, the invariant, be any formula. Let V, the variant, be an integer valued expression.

$$w : [inv, inv \wedge \neg(\bigvee i \bullet G_i] \sqsubseteq \textbf{do}([]i \bullet G_i \rightarrow w : [inv \wedge G_i, inv \wedge (0 \leq V < V[w \backslash w_0)])\textbf{od}. \tag{3.17}$$

This law states that a set of negated guards in the post condition can be iterated over. Each iteration test one of the guards, if the guard is true then a program is run specified by the original program specification with the guard added to the precondition and the post condition simplified to include only the invariant and an expression that tests an integer valued expression.

Note that this law does not provide much information about exactly how the iteration should work.

In Morgan's Refinement book he describes more detail for the refinement method including procedure calls, bags, recursion blocks and more. Only the basic laws that were investigated are discussed here.

Chapter 4

Using AI Planning for Automated Refinement

4.1 Description of Refinement Operations

To convert refinement calculus laws into PDDL operators, the input and output of each law must be mapped to predicates. Each precondition and post-condition of each refinement operation must be composed of predicates that establish a syntax tree that can be manipulated with planning operators. In order to "execute" a refinement process, a planning problem must be devised that sets up the syntax trees as the inital conditions and the goal must be to place all specification objects into the "completed" state. The following set of examples demonstrates the set up a simple planning problem.

The first part of setting up the planning problem is the allocation of objects to be used for planning. The example below allocates four specification objects, four formula objects, six variables, one precondition, one post-condition, one variable set, and four lines of code.

```
(define (problem seqcompproblem) (:domain refinement)
(:objects

s1 s2 s3 s4 - program
f1 f2 f3 f4 - formula
x1o x2o x1 y1 x2 y2 t - variable
pre1 post1 - specformula
vs1 - variableset
c1 c2 c3 c4 - codeexpression
```

Operations used within the formulas must also be specified as objects. For example, suppose assignment, and, or, and skip are being used as operators. The following declaration could be used to specify this.

```
Assigntype Andtype Addtype Ortype Skip - atype
```

Consider a specification $x1 : [pre, post]$. To specify this using a set of predicates the following is required.

```
(inframe s1 x1)
(getpre s1 pre1)
(getpost s1 post1)
(used s1)
```

The $inframe$ predicate specifies that $x1$ can be altered by the program described by the specification. The $getpre$ predicate specifies that $pre1$ is a precondition expression for the specification identified by $s1$. The $getpost$ predicate specifies that $post1$ is a post-condition expression for the specification identified by $s1$. The $used$ predicate specifies that the specification identified by $s1$ has been assigned to a particular set of frame variables, precondition and post-condition.

To plan using the refinement operators the problem statement must construct the syntax tree. The initial states of a planning problem represent the initial state description of a program. As discussed earlier, each syntactical structure required in the plan must be constructed from layers of predicates. For example, consider the arithmetic expression $x1 + x2 = t$. Based on the earlier defined objects, the following propositions would be required to construct the expression.

```
(left f1 f2)
(right f1 t)
(ctype f1 Assigntype)
(left f2 x1)
(right f2 x2)
(ctype f2 Addtype)
```

4.2 Recursive Predicates

In some places in the refinement calculus, recursive operations are required to evaluate replacements of variables. For example, in Law 1.3 "**Assignment**" If $pre \Rightarrow post[w\backslash E]$, then

$$w, x : [pre, post] \sqsubseteq w := E. \tag{4.1}$$

So there must be an implication check on $pre \Rightarrow post[w\backslash E]$. Before the check can be made a replacement must be done on the post expression - all occurrences of w must be replaced with the term E. This is a recursive replacement. Planning does not have a direct mechanism for recursion. However, recursive operations can be planned for by setting up the operators in a very specific way. First, the formula $post$ must be marked as needing a replacement. A replacement statement must be obtained from the general pool of objects allocated in the problem statement that are not assigned to the predicate $used$. A different operator is set up to wait for a formula that needs to be checked. The operator's precondition checks the left and the right components of the formula. If a component is atomic, then no more checks need to be done. If a component is not atomic, then both its left component and right component must be checked. In this way, a replacement parse tree is constructed one operation at a time. Another operator is constructed that 'waits' for these lower level structures to complete. A series of operators are applied until the final operator is applied to the top-level formula that began the process. This is not a particularly efficient way to represent the replacement or to execute it. The lack of semantics within PDDL makes it much more difficult to work with than other declarative systems that have more complete formula manipulation features.

4.3 Description of Predicates

The following types are used in predicates.

```
(:types atype formula program variableset codeexpression cnt)
(:types variable specformula - formula)
```

The following predicates are declared for use by the operators in the refinement domain. They are used to indicate the status of different components of the syntax tree for a formula. Operations use these predicates to manipulate formulas during the refinement process. The ultimate purpose of the refinement process is to emit code using the *code* $-$ xxx predicates. Once a plan is complete, some procedural code must be used to navigate the remaining predicate structure to extract the code.

```
; Formula is in the process of having some part of it replaced.
(replaceuse ?e - formula)

; Formula replacement is complete.
(replacedone ?v - variable)

; A variable is in the frame of the spec.
(inframe ?vs - variableset ?f - formula)

; A formula is actually a constant.
(constant ?f - formula)

; A flag used to check if assignment is in progress.
(testassign)

; A specification is being used.
(used ?spec - program)

; A code object is is being used.
(usedcode ?c - codeexpression)

; A formula object is being used.
(usedformula ?f - formula)

; Gets or assigns the precondition for a specification.
(getpre ?spec - program ?pre - specformula)

; Gets or assigns the post-condition for a specification.
(getpost ?spec - program ?pre - specformula)

; Gets or assigns parameters for a specification.
(getparms ?spec - program ?parmlist - variableset)
```

```
; Gets or assigns variables to a set.
(inset ?varset - variableset ?var - variable)

; Indicates that a specification is completed.
(completed ?spec - program)

; Attaches a line of code to a specification.
(coded-spec ?spec - program ?c - codeexpression)

; Attaches the left branch of a code expression.
(code-left ?code - codeexpression ?w -formula)

; Attaches the right branch of a code expression.
(code-right ?code - codeexpression ?e - formula)

; Attaches the type of a code expression.
(code-type ?code - codeexpression ?type - atype)

; Sets the order of two specifications.
(follows ?spec1 ?spec2 - program)

; Indicates that a variable must be replaced within a formula.
(replacevar ?stat - formula ?w - variable)

; Indicates that a term must be replaced within a formula.
(replaceterm ?stat - formula ?e - formula)

; Indicates that a check for replacement is required on the left
branch of a formula.
(need_replace_check_left ?post ?top - formula)

; Indicates that a check for replacement is required on the right
branch of a formula.
(need_replace_check_right ?post ?top - formula)

; Indicates that a particular formula is the topmost formula
for a specification.
(toplevel ?specformula - formula)

; Assigns the right branch of a formula.
(right ?newstat ?newstat_right - formula)

; Assigns the left branch of a formula.
(left ?newstat ?newstat_right - formula)

; Assigns the type of a formula.
(ctype ?stat - formula ?t - atype)

; Indicates that a formula is complex and not atomic.
(complex ?stat - formula)

; Indicates that checking is complete on the left branch of
a formula.
```

```
(done_replace_check_left ?stat - formula)

; Indicates that checking is complete on the right branch of
a formula.
(done_replace_check_right ?stat - formula)

; Indicates that a particular formula is ready to start the
testing process.
(ready_to_test ?stat - formula)

; Indicates that a specification is in the process of being
checked for the assignment operation.
(check-assignment ?spec - program)

; Indicates that an alteration has been made to the left
branch of a formula.
(makenewleft ?stat ?leftstat - formula)

; Indicates that a formula is to have its left branch replaced.
(replace_left ?f - formula)

; Indicates that a formula is to have its right branch replaced.
(replace_right ?f - formula)

; Used to indicate that a particular specification must be
cleaned up after a series of operations have taken place on it.
(cleanup ?spec1 ?spec2 - program)

; Indicates that two formulas have the equivalent left branches.
(same-left ?f1 ?f2 - formula)

; Indicates that two formulas have the same right branches.
(same-right ?f1 ?f2 - formula)

; Indicates that two formulas are the same.
(same ?f1 ?f2 - formula)

; Marks a formula for use by some operation.
(mark ?f - formula)

; Indicates that one formula is inside another formula.
(informula ?f1 ?f2 - formula)
```

4.4 PDDL Example: Sequential Composition

A simple example of a refinement law converted to a PDDL operator is the Sequential Composition law. Sequential composition is really specification decomposition in which one specification is split into two. Consider the specification $w : [pre, post]$. The resulting program can be specified as two programs where the result of the first feeds into the second with some intermediate state. If the intermediate state is called *mid* then the first split specification is $w : [pre, mid]$ and the second split specification is $w : [mid, post]$. To implement this as a planning operator, the following is required.

```
(:action seqcomp
:parameters (?mid - specformula
        ?spec - program
        ?spec1 - program
        ?pre ?post - specformula
        ?parms - variableset)
:precondition
    (and
        (used ?spec)
        (not (completed ?spec))
        (not(used ?spec1))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (not (= ?mid ?pre))
        (not (= ?mid ?post))
)

:effect(and
        (used ?spec1)
        (not (getpost ?spec ?post))
        (getpost ?spec ?mid)
        (getpre ?spec1 ?mid)
        (getpost ?spec1 ?post)
        (getparms ?spec1 ?parms)
        (follows ?spec ?spec1)
    )
)
```

In this example the preconditions test that the specification is used and not completed and that the second specification is not used. The precondition formula and post-condition formula are then obtained. The final step tests that the precondition and post-condition do match the middle expression to be inserted between the precondition and post-condition. The results of splitting the specification are that a new specification is set to "*used*". The post-condition for the original specification is removed and added as the post-condition to the new specification. The middle condition is added as the post-condition of the original specification and as the pre-condition for the new specification. Finally, the two specifications are ordered using the $follows$ predicate.

4.5 Manipulation of the Syntax Tree

Automated Refinement uses planning operators to manipulate the pre and $post$ conditions of a specification. Changes are made to alter the structure of the syntax tree representing the formulas that the conditions represent. For example, consider the following specification:

$$x, y[x = X \wedge y = Y, y = Y \wedge x = X] \qquad (4.2)$$

From inspection, it should be clear that pre = $post$ and so this specification should refine to the program consisting of the single operation "skip". However, the planner must use reasoning to infer that these two expressions are the same. Since AI Planning lacks any direct unification processes as found in Prolog or other languages, a plan must be derived to establish the equality. This can be done by manipulating the expressions and by setting certain predicates that apply to these expressions.

The first manipulation must be to swap the formulas around so that the left expression of the precondition lines up with the left expression of the post-condition. This is done with a planning operator named "FOL swap". The operation works by assigning predicates representing the syntax tree components of the conditions. The original state of the above expression would look something like the following if it were specified as an initial condition.

```
(left pre f1)
(left post f2)
(left f1 f3)
(left f2 f4)
(right f1 f5)
(right f2 f6)
(ctype f1 Andtype)
(ctype f2 Andtype)
(left f3 x)
(left f4 y)
(right f3 X)
(right f4 Y)
(ctype f3 Assigntype)
(ctype f4 Assigntype)
(left f5 y)
(left f6 x)
(right f5 Y)
(right f6 X)
(ctype f5 Assigntype)
(ctype f6 Assigntype)
(getpre s1 pre)
(getpost s1 post)
(inframe s1 x)
(inframe s1 y)
```

Once the swap is complete, a grounded version of the predicates would appear as the following:

```
(left pre f1)
(left post f2)
(left f1 f3)
(left f2 f4)
(right f1 f5)
(right f2 f6)
(ctype f1 Andtype)
(ctype f2 Andtype)
(left f3 x)
(left f4 y)
(right f3 X)
(right f4 Y)
(ctype f3 Assigntype)
(ctype f4 Assigntype)
(left f5 x)
```

```
(left f6 y)
(right f5 X)
(right f6 Y)
(ctype f5 Assigntype)
(ctype f6 Assigntype)
(getpre s1 pre)
(getpost s1 post)
(inframe s1 x)
(inframe s1 y)
```

The "FOL swap" operator is declared as follows:

```
(:action FOL_swap
:parameters (
    ?top - specformula
    ?f1 ?leftstat ?rightstat - formula
    ?t - atype)
:precondition
    (and
        (informula ?top ?f1)
        (not (toplevel ?top))
        (ctype ?f1 ?t)
        (left ?f1 ?leftstat)
        (right ?f1 ?rightstat)
        (not (= ?leftstat ?rightstat))
        (or
            (= ?t Assigntype)
            (= ?t Ortype)
            (= ?t Andtype)
        )
    )
:effect
    (and
        (not (left ?f1 ?leftstat))
        (not (right ?f1 ?rightstat))
        (left ?f1 ?rightstat)
        (right ?f1 ?leftstat)
        (forall (?stest - formula)
            (and
            (not (same-left ?f1 ?stest))
            (not (same-left ?stest ?f1))
            (not (same-right ?f1 ?stest))
            (not (same-right ?stest ?f1))
            )
        )
    )
)
```

Most of the "FOL swap" description should be clear from the code except for the *forall* statement. Planning effects in PDDL can contain universal quantifiers. These allow effects to be propagated over all objects that meet certain criteria. In the above example, because formula $f1$ has been modified, it is not known if it is the same as any other formula. So by default, its left and right branch must be set to be not equal to any other formula. If this turns out to be false for some formula, some other operation will set it back true.

Ultimately a specification is "complete" when it is not possible to emit any more code through transformation operations. A special operator "skipcheck" tests to see if this is true.

```
(:action skipcheck
:parameters (
    ?spec - program,
    ?pre ?post - specformula
    ?c - codeexpression)
:precondition
    (and
        (> (steps) 12)
        (used ?spec)
        (not (usedcode ?c))
        (not (completed ?spec))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (not (mark ?post))
        (or
            (= ?pre ?post)
            (same ?pre ?post)
        )
    )
:effect
    (and
        (completed ?spec)
        (code-type ?c Skip)
        (code-spec ?spec ?c)
    )
)
```

The operation for "skipcheck" is not actually complete because it only tests for equality between the precondition and post-condition. To make this complete, additional operations would have to added to check for implication. This could make the planning operations much more complex though because checking for implication would require quite a few proof steps, particularly in complicated operations.

The "skipcheck" operation checks to see if the precondition is the same as the post-condition by checking the "same" predicate. The operation that sets this predicate is "make-same".

```
(:action make_same
:parameters (?f1 ?f2 - formula ?op1 ?op2 - atype)
:precondition
    (and
        (complex ?f1)
        (complex ?f2)
        (not (toplevel ?f1))
        (not (toplevel ?f2))
        (not (= ?f1 ?f2))
        (atype ?f1 ?op1)
        (atype ?f2 ?op2)
        (= ?op1 ?op2)
        (same-left ?f1 ?f2)
        (same-left ?f2 ?f1)
        (same-right ?f1 ?f2)
        (same-right ?f2 ?f1)
        (not (and
            (same ?f1 ?f2)
            (same ?f2 ?f1)
        ))
    )
:effect
    (and
        (same ?f1 ?f2)
        (same ?f2 ?f1)
        (not (mark ?f1))
        (not (mark ?f2))
    )
)
```

This operator simply checks the left and right sides of two formulas and checks the operation. If the operation is the same and the branches are the same, then the formulas must be the same. The "mark" predicate is used by the "assignment" operation for optimization purposes. If "assignment" is being checked at some level in the plan, then the expression used in the assignment must be marked so that similarity operation is not applied. Although this can be discovered through the mutual exclusion planning process, it would take many steps to prove it. So the "mark" operation is an explicit indicator of this fact.

Another test that does not seem obvious is the one added in the precondition to ensure that if two expression are already the same. This is used to ensure that if two formulas are already the same, that they are not made the same again. This could happen in the case where the "mark" is removed for some reason. There is no way to directly tell the planner that this detail is really a side-effect of the operation and should not be considered as part of the goal. If there was a way to indicate this, then the "same" check at the end of the precondition could be removed.

```
(:action make_same_left
:parameters (?f1 ?f2 ?leftf1 ?leftf2 - formula)
:precondition
    (and
        (> (steps) 12)
        (not (testassign))
        (complex ?f1)
        (complex ?f2)
        (not (= ?f1 ?f2))
        (not (toplevel ?f1))
        (not (toplevel ?f2))
        (left ?f1 ?leftf1)
        (left ?f2 ?leftf2)
        (or
            (= ?leftf1 ?leftf2)
```

```
                (and
                (same ?leftf1 ?leftf2)
                (same ?leftf2 ?leftf1)
                )
            )
            (not (and
                (same-left ?f1 ?f2)
                (same-left ?f2 ?f1)
            ))

        )
:effect
        (and
            (increase (steps) 1)
            (same-left ?f1 ?f2)
            (same-left ?f2 ?f1)
        )
)


(:action make_same_right
:parameters (?f1 ?f2 ?rightf1 ?rightf2 - formula)
:precondition
        (and
            (complex ?f1)
            (complex ?f2)
            (not (= ?f1 ?f2))
            (not (toplevel ?f1))
            (not (toplevel ?f2))
            (right ?f1 ?rightf1)
            (right ?f2 ?rightf2)
            (or
                (= ?rightf1 ?rightf2)
                (and
                (same ?rightf1 ?rightf2)
                (same ?rightf2 ?rightf1)
                )
            )
            (not (and
                (same-right ?f1 ?f2)
                (same-right ?f2 ?f1)
            ))
        )
:effect
        (and
            (increase (steps) 1)
            (same-right ?f1 ?f2)
            (same-right ?f2 ?f1)
        )
)
```

In some contexts the precondition or the post-condition can be reduced to a simpler expression. If the post-condition can be reduced to "true" then if the precondition can be reduced to true, then the specification is complete. The following operations check to see if this is possible.

```
(:action make_formula_left_true
:parameters (
    ?f ?f1 ?leftf1 ?rightf1 - formula
    ?t - atype)
:precondition
    (and
        (not (testassign))
        (not (toplevel ?f))
        (left ?f ?f1)
        (ctype ?f1 Assigntype)
        (left ?f1 ?leftf1)
        (right ?f1 ?rightf1)
        (or
            (= ?leftf1 ?rightf1)
            (same ?leftf1 ?rightf1)
        )
    )
:effect
    (and
        (increase (steps) 1)
        (left ?f Trueformula)
        (not (left ?f ?f1))
        (not (ctype ?f1 Assigntype))
        (not (left ?f1 ?leftf1))
        (not (right ?f1 ?rightf1))
        (when (mark ?f) (not (mark ?f)))
    )
)

(:action make_formula_right_true
:parameters (
    ?f ?f1 ?leftf1 ?rightf1 - formula
    ?t - atype)
:precondition
    (and
        (not (testassign))
        (not (toplevel ?f))
        (right ?f ?f1)
        (ctype ?f1 Assigntype)
        (left ?f1 ?leftf1)
        (right ?f1 ?rightf1)
        (or
            (= ?leftf1 ?rightf1)
            (same ?leftf1 ?rightf1)
        )
    )
:effect
    (and
        (increase (steps) 1)
        (right ?f Trueformula)
        (not (right ?f ?f1))
        (not (ctype ?f1 Assigntype))
        (not (left ?f1 ?leftf1))
        (not (right ?f1 ?rightf1))
        (when (mark ?f) (not (mark ?f)))
    )

)
```

One other check that can be made to test for completion of a specification is a true post-condition. The following operations check for this situation and make the assignment. By simplifying the conditions, this allows the "skipcheck" operation to succeed and finalize the specification.

```
(:action make-true-post
:parameters (
    ?spec - program
    ?pre ?post - specformula
    ?leftside ?rightside - formula
    ?ftype - atype)
:precondition
    (and
        (used ?spec)
        (not (completed ?spec))
        (not (toplevel ?post))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (complex ?post)
        (left ?post ?leftside)
        (right ?post ?rightside)
        (ctype ?post Assigntype)
        (not (mark ?post))
        (or
            (= ?leftside ?rightside)
            (same ?leftside ?rightside)
        )
    )
:effect
    (and
        (not (getpre ?spec ?pre))
        (not (getpost ?spec ?post))
        (getpre ?spec True)
        (getpost ?spec True)
    )
)

(:action make-true2-post
:parameters (
    ?spec - program
    ?pre ?post - specformula
    ?leftside ?rightside - formula
    ?ftype - atype)
:precondition
    (and
        (used ?spec)
        (not (completed ?spec))
        (not (toplevel ?post))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (complex ?post)
        (left ?post ?leftside)
        (right ?post ?rightside)
        (ctype ?post Andtype)
        (not (mark ?post))
        (= ?leftside ?rightside)
        (= ?leftside Trueformula)
    )
:effect
    (and
        (not (getpre ?spec ?pre))
        (not (getpost ?spec ?post))
        (getpre ?spec True)
        (getpost ?spec True)
    )
)
```

4.6 Swapping Variables Program Specification

Deriving the code for swapping variables is a basic programming task and is a good challenge problem for automated refinement. The specification for this problem is as follows:

$$x, y[x = X \wedge y = Y, x = Y \wedge y = X] \tag{4.3}$$

In order to solve the problem, the automated refinement system must introduce a local variable and use it as intermediate storage during the swap process. The value of one of the frame variables must be assigned to the local variable. The value of the second variable must be assigned to the first. Then the value of the local variable must be assigned to the second variable. During the course of these operations, the post-condition is modified at each step as code is emitted by the refinement operations. At some point in the process, the precondition will be the same as the post-condition i.e. $pre \rightarrow post$. This specification resolves to **skip** and the refinement is complete.

The complete development is as follows:

$x, y[x = X \wedge y = Y, x = Y \wedge y = X]$

$\sqsubseteq$ **var** $t \bullet x, y, t[x = X \wedge y = Y, x = Y \wedge y = X]$

$\sqsubseteq$ "following assignment"

$x, y, t[x = X \wedge y = Y, t = Y \wedge y = X]; x := t$

$\sqsubseteq$ "following assignment"

$x, y, t[x = X \wedge y = Y, t = Y \wedge x = X]; y := x$

$\sqsubseteq$ "Assignment"

$x, y, t[x = X \wedge y = Y, y = Y \wedge x = X]; t := y$

$= x, y, t[x = X \wedge y = Y, x = X \wedge y = Y]$

$\sqsubseteq$ "Skip"

skip

The rules for "assignment" and "following assignment" do not translate directly into operations. To emit assignment code, an operation is required to see if the syntactic replacement is possible as defined by the "assignment" and "following assignment" laws. It would very inefficient for replacement checks to be occurring all the time for every syntactical component. So there is one operation that triggers the assignment check. This operation cascades downward to the right and left branches of the syntax tree of an expression. Once an assignment check has been completed on a condition formula, the assignment must be cleaned up and code must be emitted. This is done by the "finish assignment" operator.

Below are the operators required for automating the refinement of the swapping variable program specification.

```
(:action finish_assignment
:parameters (
    ?spec ?spec2 - program
    ?post - specformula
    ?w - variable
    ?e - formula
    ?c - codeexpression)
:precondition
    (and
        (not (used ?spec2))
        (check-assignment ?spec)
        (getpost ?spec ?post)
        (toplevel ?post)
        (or
            (replace_right ?post)
            (replace_left ?post)
        )

        (not (usedcode ?c))
        (replacevar ?post ?w)
        (replaceterm ?post ?e)
        (not (= ?w ?e))

    )
:effect
    (and
        (not (replacevar ?post ?w))
        (not (replaceterm ?post ?e))
        (not (testassign))
        (not (toplevel ?post))
        (follows ?spec ?spec2)
        (not (check-assignment ?spec))
        (cleanup ?spec ?spec2)
        (forall (?f - formula)
            (when (and (informula ?post ?f)(complex ?f))
                (mark ?f)
            )
        )

        (forall (?f - formula)
            (and
                (not (need_replace_check_left ?f ?post))
                (not (need_replace_check_right ?f ?post))
            )
        )

        (when
            (replace_right ?post)
            (not (replace_right ?post))
        )

        (when
            (replace_left ?post)
            (not (replace_left ?post))
        )
        (usedcode ?c)
        (used ?spec2)
        (completed ?spec2)
        (code-spec ?spec2 ?c)
        (code-left ?c ?w)
        (code-right ?c ?e)
        (code-type ?c Assigntype)
        (replaceuse ?e)
    )
)
```

```
(:action assignment_check
:parameters (
    ?spec - program,
    ?e - formula,
    ?w - variable,
    ?post - specformula
    ?vs - variableset)
:precondition
    (and
        (used ?spec)
        (not (completed ?spec))
        (not (check-assignment ?spec))
        (getparms ?spec ?vs)
        (inset ?vs ?w)
        (getpost ?spec ?post)
        (complex ?post)
        (not (complex ?e))
        (not (= ?w ?e))
        (not (replaceuse ?e))
        (not (replacedone ?w))
        (or
            (inframe ?vs ?e)
            (constant ?e)
            (informula ?post ?e)
        )
;       (not (usedformula ?e))
    )
:effect
    (and
        (check-assignment ?spec)
        (need_replace_check_left ?post ?post)
        (need_replace_check_right ?post ?post)
        (toplevel ?post)
        (replacevar ?post ?w)
        (replaceterm ?post ?e)
    )
)
```

```
(:action replace_check_left
:parameters (
    ?top - specformula
    ?stat ?leftstat - formula
    ?w - variable
    ?e - formula)
:precondition
    (and
        (usedformula ?stat)
        (usedformula ?leftstat)
        (toplevel ?top)
        (need_replace_check_left ?stat ?top)
        (left ?stat ?leftstat)
        (informula ?stat ?w)
        (replacevar ?top ?w)
        (replaceterm ?top ?e)
        (not (= ?w ?e))
    )

:effect
    (and
        (not (need_replace_check_left ?stat ?top))
        (when (not (complex ?leftstat))
            (done_replace_check_left ?stat)
        )
        (when (and (not (complex ?leftstat))(= ?leftstat ?w))
            (and
                (not (left ?stat ?leftstat))
                (left ?stat ?e)
                (replace_left ?top)
                (usedformula ?e)
                (replacedone ?w)
                (not (informula ?stat ?w))
                (not (informula ?top ?w))
                (informula ?stat ?e)
                (informula ?top ?e)
            )
        )
        (when (complex ?leftstat)
            (and
            (need_replace_check_left ?leftstat ?top)
            (need_replace_check_right ?leftstat ?top)
            )
        )
    )
)
```

```
(:action replace_check_right
:parameters (
    ?top - specformula
    ?stat ?rightstat - formula
    ?w - variable
    ?e - formula)
:precondition
    (and
        (usedformula ?stat)
        (usedformula ?rightstat)
        (toplevel ?top)
        (need_replace_check_right ?stat ?top)
        (right ?stat ?rightstat)
        (informula ?stat ?w)
        (replacevar ?top ?w)
        (replaceterm ?top ?e)
        (not (= ?w ?e))
    )

:effect
    (and
        (not (need_replace_check_right ?stat ?top))
        (when (not (complex ?rightstat))
            (done_replace_check_right ?stat)
        )
        (when (and (not (complex ?rightstat))(= ?rightstat ?w))
            (and
                (not (right ?stat ?rightstat))
                (right ?stat ?e)
                (replace_right ?top)
                (usedformula ?e)
                (replacedone ?w)
                (not (informula ?stat ?w))
                (informula ?stat ?e)
                (not (informula ?top ?w))
                (informula ?top ?e)
            )
        )
        (when (complex ?rightstat)
            (and
            (need_replace_check_left ?rightstat ?top)
            (need_replace_check_right ?rightstat ?top)
            )
        )
    )
)
```

4.7 Looping and Complex Algorithms

In Morgan's refinement laws there is a law for iteration. The iteration law is the following:

Law 7.1: **Iteration** Let *inv*, the invariant, be any formulat. Let V, the variant, be an integer valued expression.

$$w : [inv, inv \wedge \neg(\bigvee i \bullet G_i] \sqsubseteq \mathbf{do}([]i \bullet G_i \rightarrow w : [inv \wedge G_i, inv \wedge (0 \leq V < V[w \backslash w_0)])\mathbf{od}. \quad (4.4)$$

The law states that if the post-condition consists of a set of conditions that must be true that can be indexed in some way, then the refinement is a program that iterates over all the conditions until all the conditions have been met. This, however, is an extremely weak law in that it is very difficult to discover this exact situation in a specification. Also, there is no guarantee that iterating repeatedly over a set of conditions will make those conditions true, even if each condition is set one at a time. For example, its possible that setting one condition will interfere with another condition. So the law is incomplete in that it does not consider interference. There is also no refinement law to find the correct sequence of steps to make the iteration work in all cases. Iteration requires induction and reasoning about induction is a very difficult task for which there is no general solution.

Morgan's iteration law can be used simple situations, but in general much more intelligent processing is required. Deeper analysis is possible by reasoning about the inductive argument required to infer how the iteration should work. The section on "Further Research" describes some possible automation techniques that could be investigated to solve this problem. However, without this inductive step reasoner, an alternative way of handling this is to modify the post-condition language to include more information in the specification. If, for example, a particular post-condition includes universal quantification of a variable with a particular type, then this information could be used directly by the code generator to infer iteration over all type values. Helper expressions of this sort are like specification "macros" that get automatically expanded into particular code such as iteration.

Refinement macros can also be used to make the refinement process more efficient. Each operator can be associated with a block of code to execute and a pre and post condition that indicate the context in which the block can be used. With Morgan's refinement laws, each line of code must be derived from application of one refinement law. However, if it is known in advance that some section of code has a known pre-condition and a known post-condition, any specification that matches these conditions can immediately be refined to the entire block of code.

Consider the example of building a sort program. Sorting requires iteration. The input can be described as a collection. The output is a sorted list, a structure that has all of its elements arranged according to certain rules. The condition expression must be modified to include the quantifiers and other pre-defined procedures that allow the input collection to be converted into a sorted list.

There are two quantifiers that can be added: all and exists. Each must allow a variable and a set that the variable belongs to. The quantifier must also have scope which is the formula for which the quantifier applies. These are the same as the $\forall$ and $\exists$ quantifiers in FOL.

```
(all (member var set) spec)
(exist (member var set) spec)
```

A few set operators are required to simplify some of the operations involving arrays. Although this could also be reduced to FOL, there is no need to since arrays are commonly worked with in programming. One operation required is "up to" which can be symbolized by "|". This takes all the elements of an array up to some index. For example: given the list of ordered pairs $< a, 1 >< b, 2 >< c, 3 >$ then $M|2$ would include $< a, 1 >< b, 2 >$.

Another operation is exclude. This takes all elements from one list excluding elements from another list and can be symbolized by "/". The exclude operator can have the added function of ignoring second items of ordered pairs. Consider the lists $< a, 2 >< b, 4 >< c, 1 >= M$ and $< a, 3 >< b, 1 >= N$, then $M/N \Rightarrow< c, 1 >$.

Finally, the predicate "assigned" is taken to mean that all items positioned in M up to x are assigned. An order list of indices is $1, 2..n = Z$.

```
(and (exists (member M L)
    (all (member x Z) (exist (member v  L)
    (and (member <v,x> M)
    (and (all (member y L/M|x) (v<=y))) )
```

This statement can be translated into code in the following way:

```
for x in Z
        v = undef
        for y in L/M|x
                If v == undef or !(v<=y) then v=y
        end
        Add <v,x> to M
end
```

The operations below represent a suggested plan domain which may work with the expression modifications outlined above to handle the generation of a program that sorts a list.

```
(:action existcheck
:parameters (
    ?spec - program
    ?listvar - variable
    ?post - specformula
    ?funcall - formula
    ?c ?c2 - codeexpression)
:precondition
    (and
        (used ?spec)
            (not (used ?spec2))
            (not (usedcode ?c2))
            (getpost ?spec ?post)
            (left ?post ?funcall)
            (ctype ?funcall Exist)
            (left ?funcall ?listvar)
        )
:effect
    (and
        (usedcode ?c2)
        (code-type ?c2 Declare)
        (code-left ?c2 ?listvar)
        (used ?spec2)
        (completed ?spec2)
        (code-spec ?spec2 ?c2)
        (not (ctype ?funcall Exist))
        (ctype ?funcall Apply)
        (follows ?spec2 ?spec)
        (when (firstspec ?spec)
            (and
                (not (firstspec ?spec))
                (firstspec ?spec2)
        )
))
)
```

```
(:action allcheck
:parameters (
    ?spec ?spec2 ?spec3 - program
    ?newvar ?listvar - variable
    ?post - specformula
    ?funcall - formula
    ?c2 ?c3 - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (used ?spec2))
        (not (used ?spec3))
        (not (usedcode ?c2))
        (not (usedcode ?c3))
        (not (usedformula ?newvar))
        (getpost ?spec ?post)
        (left ?post ?funcall)
        (ctype ?funcall All)
        (left ?funcall ?listvar)
        ;(right ?funcall ?listtype)
)
:effect
    (and
        (usedcode ?c2)
        (usedcode ?c3)
        (code-type ?c2 Loop)
        (code-left ?c2 ?listvar)
        (code-right ?c2 ?newvar)
        (code-spec ?spec2 ?c2)

        (completed ?spec2)
        (completed ?spec3)
        (used ?spec2)
        (used ?spec3)
        (code-type ?c3 Endloop)
        (code-spec ?spec3 ?c3)
        (follows ?spec2 ?spec)
        (follows ?spec ?spec3)
        (usedformula ?newvar)
        (not (ctype ?funcall All))
        (ctype ?funcall Apply)
        (when (firstspec ?spec)
            (and
                (not (firstspec ?spec))
                (firstspec ?spec2)
        )
))
)
```

```
(:action apply
:parameters (
    ?spec - program
    ?post - specformula
    ?f ?rightf - formula)
:precondition
    (and
        (used ?spec)
        (getpost ?spec ?post)
        (left ?post ?f)
        (ctype ?f Apply))
:effect
    (and
        (not (ctype ?f Apply))
        (not (left ?post ?f))
        (when (right ?f ?rightf)
            (left ?post ?rightf)
        )
        (when (not (right ?f ?rightf))
        (left ?post True)
))
)
```

```
(:action shiftover
:parameters (
    ?spec - program
    ?post - specformula
    ?f ?newleft
    ?newright - formula)
:precondition
    (and
        (used ?spec)
        (getpost ?spec ?post)
        (left ?post True)
        (right ?post ?f)
        (left ?f ?newleft)
        (right ?f ?newright)
        (ctype ?post And)
        (ctype ?rightf ?t))
:effect
    (and
        (not (ctype ?post And))
        (ctype ?post ?t)
        (not (left ?post True))
        (left ?post ?newleft)
        (right ?post ?newright)
        )
)
```

```
(:action makecomplete
:parameters (
    ?spec - program
    ?post - specformula
    ?f ?newleft
    ?newright - formula)
:precondition
    (and
        (used ?spec)
        (not (completed ?spec))
        (getpost ?spec ?post)
        (= ?post True))
:effect
    (and
        (completed ?spec)
))
```

```
(:action nextmin
:parameters (
    ?spec - program
    ?var - variable
    ?post - specformula
    ?comp ?comp2 - formula
    ?rightf - formula
    ?c ?c2 - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (usedcode ?c))
        (getpost ?spec ?post)
        (left ?post ?f)
        (left ?f ?var)
        (ctype ?post NEXTMIN)
        (right ?post ?rightf)
        (not (usedformula ?comp))
        (not (= ?comp ?comp2))
        (not (usedformula ?comp2)))
:effect
    (and
        (code-type ?c2 If)
        (code-left ?c2 ?comp)
        (left ?comp ?var)
        (right ?comp ?rightf)
        (code-right ?c2 ?comp2)
        (left ?comp2 ?var)
        (right ?comp2 ?rightf)
        (ctype ?comp2 Assigntype)
        (usedcode ?c2)
        (when (code-spec ?spec ?c)
            (and
                (not (code-spec ?spec ?c))
                (code-spec ?spec ?c2)
                (next ?c2 ?c)))
        (when (not (code-spec ?spec ?c))
            (code-spec ?spec ?c2))
))
```

```
(:action Member
:parameters (
    ?spec ?spec2 - program
    ?post - specformula
    ?f ?leftf ?rightf - formula
    ?c ?c2 - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (usedcode ?c))
        (not (used ?spec2))
        (getpost ?spec ?post)
        (left ?post ?f)
        (ctype ?f Member)
        (left ?f ?leftf)
        (right ?f ?rightf))
:effect
    (and
        (code-type ?c2 Member)
        (code-left ?c2 ?leftf)
        (code-right ?c2 ?rightf)
        (when (code-spec ?spec ?c)
            (and
                (not (code-spec ?spec ?c))
                (code-spec ?spec ?c2)
                (next ?c2 ?c)))
        (when (not (code-spec ?spec ?c))
            (code-spec ?spec ?c2))
        (when (ctype ?leftf Pair)
            (and
                (not (ctype ?f Member))
                (ctype ?f Assigntype)
                (code-left ?c2 ?newvar)
                (not (left ?f ?leftf))
                (left ?f ?newvar)
                (not (right ?f ?rightf))
                (right ?f ?leftf)))
        (when (not (ctype ?leftf Pair))
            (and
                (code-left ?c2 ?leftf)
                (not (ctype ?f Member))
                (ctype ?f Apply))))
))
```

```
(:action Pair
:parameters (
    ?spec - program
    ?post - specformula
    ?newvar ?f ?leftf ?rightf - formula
    ?c ?2 ?c3- codeexpression)
:precondition
    (and
        (used ?spec)
        (not (usedcode ?c2))
        (not (usedcode ?c3))
        (not (usedformula ?newvar))
        (getpost ?spec ?post)
        (left ?post ?f)
        (ctype ?f Pair)
        (left ?f ?leftf)
        (right ?f ?rightf))
:effect
    (and
        (not (ctype ?f Pair))
        (ctype ?f Apply)
        (code-type ?c2 Pair)
        (code-left ?c2 ?leftf)
        (code-right ?c2 ?rightf)
        (code-type ?c3 Assigntype)
        (code-left ?c3 ?newvar)
        (code-right ?c3 ?c2)
        (usedcode ?c2)
        (usedcode ?c3)
        (when (code-spec ?spec ?c)
            (and
                (not (code-spec ?spec ?c))
                (code-spec ?spec ?c3)
                (next ?c3 ?c)))
        (when (not (code-spec ?spec ?c))
            (code-spec ?spec ?c3))
))
```

Chapter 5

Experimental Results

A few different problems were attempted using the developed refinement system, Sequential Composition, Assignment, Swapping Values, Minimum and Sort. All of the operations for Sort were developed but the final planning work was not completed as of the time of the completion of this document.

5.1 Sequential Composition Test

The sequential composition example was described in section 4.5. For this example, a simple problem was provided, simply to attempt the decomposition.

```
(define (problem seqcompproblem) (:domain refinement)
(:objects
s1 s2 s3 s4 - program
f1 f2 f3 f4 Trueformula - formula
True Test - specformula
v1 v2 v3 v4 v5 - variable
post1 - specformula
vs1 - variableset
c1 c2 - codeexpression
code1 code2 - code
cnt1 - cnt
Assigntype Andtype Plustype Minustype Timestype
Dividetype Ortype Openbrackettype Closebrackettype Skip - atype
    )
(:init
    (= (steps) 0)
    (used s1)
    (getpre s1 True)
    (getpost s1 post1)
    (left post1 v1)
    (right post1 v2)
    (ctype post1 Assigntype)
    (getparms s1 vs1)
    (inset vs1 v1)
    (complex post1)
)
```

```
(:goal (and
(used s2)
))
```

The following result was recorded. Note that the unassigned object *Test* is what was used in the replacement. No other object was available so this was the only choice possible so planning was fast. However, this object is meaningless in any context unless it is actually assigned some formula. So the sequential composition operation can be used by a person using a manual step, because the replacement for the "mid" expression would be decided by a person. Unfortunately, deciding which formula to choose as the intermediate expression is not trivial and so it is not easily automated. In order to make use of the operator, there would have to be other operators that attempt to derive some intermediate expressions. While this is ideally correct, it would severely impact planning by introducing many meaningless operations into the planning process.

```
ff: found legal plan as follows

step    0: SEQCOMP TEST S1 S2 TRUE POST1 VS1


time spent:     0.78 seconds instantiating 66150 easy,
                     14840 hard action templates
                0.09 seconds reachability analysis, yielding
                     144 facts and 182 actions
                0.00 seconds creating final representation with
                     84 relevant facts, 0 relevant fluents
                0.01 seconds computing LNF
                0.00 seconds building connectivity graph
                0.00 seconds searching, evaluating 2 states,
                     to a max depth of 1
                0.88 seconds total time
```

5.2 Swapping Variable Test

The Swapping Values experiment was the most meaningful one of the test problem set. This run demonstrated full planning and full use of refinement laws. It also demonstrates the use of planning for proving logical properties of expressions which is vital for the refinement process. Total planning time was 1 minute which is obviously not reasonable in a real-world situation. Also, the object set size in the planning problem was extremely small; there were just enough objects to verify that the algorithm works.

```
(define (problem seqcompproblem) (:domain refinement)
(:objects
s1 s2 s3 s4 - program
f1 f2 f3 f4 Trueformula - formula
True - specformula
x1o x2o x1 y1 x2 y2 t - variable
```

```
pre1 post1 - specformula
vs1 - variableset
c1 c2 c3 c4 - codeexpression
cnt1 - cnt
Assigntype Andtype Ortype Skip - atype
    )
(:init
    (used s1)
    (getpre s1 pre1)
    (getpost s1 post1)
    (left post1 f1)
    (right post1 f2)
    (ctype post1 Andtype)

    (left pre1 f3)
    (right pre1 f4)
    (ctype pre1 Andtype)



    (left f3 x1o)
    (right f3 y1)
    (ctype f3 Assigntype)
    (left f4 x2o)
    (right f4 y2)
    (ctype f4 Assigntype)



    (left f1 x2)
    (right f1 y1)
    (ctype f1 Assigntype)
    (left f2 x1)
    (right f2 y2)
    (ctype f2 Assigntype)

    (getparms s1 vs1)

    (same x1o x1)
    (same x2o x2)
    (same x1 x1o)
    (same x2 x2o)


    (inset vs1 x1)
    (inset vs1 x2)
    (inset vs1 t)
    (complex post1)
    (complex f1)
    (complex f2)

    (complex pre1)
    (complex f3)
    (complex f4)
    (informula pre1 f3)
    (informula pre1 f4)
    (informula pre1 pre1)

    (informula post1 f1)
    (informula post1 f2)
    (informula post1 post1)

    (informula post1 x1)
    (informula post1 x2)

    (informula f2 x1)
    (informula f1 x2)

    (usedformula pre1)
    (usedformula post1)
    (usedformula f1)
    (usedformula f2)
    (usedformula Trueformula)
    (usedformula x1)
    (usedformula x2)
    (usedformula t)
    (inframe vs1 x1)
    (inframe vs1 x2)
    (inframe vs1 t)
    (= (steps) 0)
)

(:goal (and
(completed s1)
(completed s2)
(completed s3)
(completed s4)
)
```

Based on the above problem statement

```
ff: found legal plan as follows

step    0: ASSIGNMENT_CHECK S1 X1 X2 POST1 VS1
        1: REPLACE_CHECK_LEFT POST1 POST1 F1 X2 X1
        2: REPLACE_CHECK_RIGHT POST1 POST1 F2 X2 X1
        3: REPLACE_CHECK_LEFT POST1 F1 X2 X2 X1
        4: FINISH_ASSIGNMENT S1 S2 POST1 X2 X1 C1
        5: ASSIGNMENT_CHECK S1 T X1 POST1 VS1
        6: REPLACE_CHECK_RIGHT POST1 POST1 F2 X1 T
        7: REPLACE_CHECK_LEFT POST1 F2 X1 X1 T
        8: FINISH_ASSIGNMENT S1 S3 POST1 X1 T C2
        9: ASSIGNMENT_CHECK S1 X2 T POST1 VS1
       10: REPLACE_CHECK_RIGHT POST1 POST1 F2 T X2
       11: REPLACE_CHECK_LEFT POST1 F2 T T X2
       12: FINISH_ASSIGNMENT S1 S4 POST1 T X2 C3
```

```
        13: MAKE_SAME_RIGHT F3 F1 Y1 Y1
        14: MAKE_SAME_LEFT F3 F1 X1O X1
        15: MAKE_SAME F3 F1
        16: MAKE_SAME_RIGHT F4 F2 Y2 Y2
        17: MAKE_SAME_LEFT F4 F2 X2O X2
        18: MAKE_SAME F4 F2
        19: MAKE_SAME_LEFT POST1 PRE1 F1 F3
        20: MAKE_SAME_RIGHT POST1 PRE1 F2 F4
        21: MAKE_SAME POST1 PRE1


time spent:     7.55 seconds instantiating 132396 easy, 121500
                     hard action templates
                0.33 seconds reachability analysis, yielding
                     973 facts and 792 actions
                0.03 seconds creating final representation with
                     299 relevant facts, 2 relevant fluents
                0.07 seconds computing LNF
                0.02 seconds building connectivity graph
               60.76 seconds searching, evaluating 33823 states,
                     to a max depth of 2
               68.76 seconds total time
```

5.3 Selecting the Minimum from a list

The problem description below is based on the definition of the *lte* operator. This is a kind of macro operator that checks if a value is less than or equal to another value. The post-condition that we would like to establish is that all tests of values in L compared to v will be less than or equal to v. The *lte* operator will then make the assignment based on the condition being true. The post condition expression for this is the following:

$(all(ltev)L)$

The code that will be generated will loop over L and apply lte v to each element.

Here is how the problem must be described:

```
(define (problem seqcompproblem) (:domain refinement)
(:objects
s1 s2 s3 s4 - program
f1 f2 f3 f4 Trueformula - formula
True - specformula
x1o x2o x1 y1 x2 y2 t - variable
pre1 post1 - specformula
vs1 - variableset
c1 c2 c3 c4 - codeexpression
cnt1 - cnt
Assigntype Andtype Ortype Skip All - atype
Move LTE If Loop Apply Endloop - atype
    )
(:init
    (used s1)
    (getpre s1 pre1)
    (getpost s1 post1)
    (left post1 f1)
    (right post1 y1)
    (ctype post1 All)
    (ctype f1 LTE)
    (left f1 x1)


    (getparms s1 vs1)

    (same x1o x1)
    (same x2o x2)
    (same x1 x1o)
    (same x2 x2o)


    (inset vs1 x1)
    (inset vs1 x2)
    (inset vs1 t)
    (complex post1)
    (complex f1)
    (complex f2)

    (complex pre1)
    (complex f3)
    (complex f4)

    (informula pre1 f3)
    (informula pre1 f4)
    (informula pre1 pre1)

    (informula post1 f1)
    (informula post1 f2)
    (informula post1 post1)

    (informula post1 x1)
    (informula post1 x2)

    (informula f2 x1)
    (informula f1 x2)

    (usedformula pre1)
    (usedformula post1)
    (usedformula f1)
    (usedformula f2)
    (usedformula Trueformula)
    (usedformula x1)
    (usedformula x2)
    (usedformula t)
;   (constant y1)
;   (constant y2)
    (inframe vs1 x1)
    (inframe vs1 x2)
    (inframe vs1 t)
    (= (steps) 0)
)

(:goal (and
(completed s1)

))
```

The operators that were used with this experiment are listed in Appendix C.

The following shows the results of running the plan in FF:

```
ff: found legal plan as follows

step    0: ALLCHECK S1 S4 Y2 Y1 POST1 F1 C4 C4
        1: APPLY S1 POST1 F1 X1 LTE
        2: LTE S1 X1 POST1 Y1 X20 Y2 C3


time spent:   13.07 seconds instantiating 1785144 easy,
                    246924 hard action templates
               2.17 seconds reachability analysis,
                    yielding 1093 facts and 10547 actions
               0.06 seconds creating final representation
                    with 378 relevant facts, 2 relevant fluents
               0.32 seconds computing LNF
               0.09 seconds building connectivity graph
               0.05 seconds searching, evaluating 4 states,
                    to a max depth of 1
              15.76 seconds total time
```

Chapter 6

Analysis

6.1 Sort Induction

Induction in proofs is very difficult. Using induction in a proof generally requires a great deal of thought and effort, and sometimes creative genius. When an automated proof system must prove something using induction, this generally requires human intervention. This is a problem for code synthesis methods such as Morgan's Refinement where induction is required to handle complex reasoning. Without a guide to the process, the state explosion problem will limit these methods to only simple proofs and therefore limit code generation to simple programs.

The problem faced by proof systems is that the inductive step requires some kind of intuition about the problem that can be exploited. The solution is to provide this intuition using by considering examples of the problem at hand and using planning to order operations on the example to generate the desired target representation. This method is based on the way humans typically solve problems - we experiment with samples until a repeatable sequence of steps can be found to solve the given problem.

Experimentation involves considering samples of a problem and potential solution methods to find a sequence of operations that correspond to the sample inputs and outputs. We apply transformations to these representations until a repeatable method is developed. A transformation ordering is a sequence of transformations that remain constant regardless of input and that consistently generate the desired output from the sample input. This does not guarantee that a solution is found, rather, it points to the intuition about the problem that can be exploited to attempt a proof. If a proof cannot be found using the "intuition based transformation sequence" then the process can be repeated.

If a programmer has a good understanding of a problem, he should be able to generate examples of input and examples of output. For example, consider the idea of sorting something. If a programmer has an understanding of this problem they would be able to write the following:

Sample 1: A C D B E $\Rightarrow$ A B C D E

Sample 2: E B C D A $\Rightarrow$ A B C D E

Or more generally, an algorithm can be devised that generates random input along with random output where the output represents the input transformed as if the target code transformed it. In the case of a sort routine, consider a randomizing system that generates an ordered list and then scrambles it. The ordered list is the output of a sort routine and the scrambled list is the input.

Now consider what induction is. Induction is a method by which a proof relating to a single item of a set can apply to complete sets in a way that proves some property of the set. All looping procedures require some kind of induction since the loop repeatedly applies to some state of an application in such a way that a desired end state is ultimately achieved. But the specifics of a proof with induction are extremely complex and therefore difficult for automated systems. For example, there are many different kinds of sorting methods. All of these methods involve some kind of looping over the states of a list. When proving that some sorting technique can be derived, which induction techniques should be used? Bubble sort, insertion sort, quick sort, merge sort, etc. are all valid but unique methods of sorting a list. Given the FOL description of sort, the challenge is to induce the nested loops required to implement the desired code.

An "algorithm" is a repeatable sequence of steps that achieves some desired goal given some known input. During the process of thinking of how to construct an algorithm, a person will typically "play" with the input and output data attempting to apply sequences of operations until some repeatable sequence is generated. For a list, the kinds of operations that might be used include replicate, swap, move, compare, and iterate. When thinking about the sort routine, a person would rearrange variations of these operations until some sequence sorted the list. Then a person would try and detect a pattern in the sequence of operations. If no pattern can be found, the process can repeat itself.

In the case of the sort example 1, consider the following:

Sample 1: A C D B E $\Rightarrow$ A B C D E

Iterate to B, Swap b d, swap b c

For example 2 we might have

Sample 2: E B C D A $\Rightarrow$ A B C D E

Iterate to a, swap a e,

But **Sample 3:** B D E C A $\Rightarrow$ A B C D E

Iterate to a, swap a b, iterate to b, swap b d, iterate to c, swap c e, iterate to d, swap d e The insight should be that each time that the *iterate* operation is chosen, it stops on an item that is greater than the item assessed before. Swap is observed to always apply to successive positions which are not filled with ordered values.

These sequences can be used to "learn" the code. Consider a program as containing some sequence of decision making states or lines of code (LOC). The decision making LOC emits an operation and then goes to one of the other LOCs. Unfortunately, the next LOC is not known, but the next emitted operation is. We hypothesize a sequence of connected LOCs. No other interconnection is assumed. To attach LOCs, the method of refinement can be used. This method considers the pre-conditions of a line of code, and generates only the possible next LOCs. Then, in-turn, each LOC is selected, and all possible next LOCs are selected. This continues until no LOCs can be evaluated. A pre-determined LOC count is chosen so as to limit the search.

The end product of the refinement method for LOC attachment results in a graph. This graph represents many possible algorithms - the goal is to refine the graph down to one algorithm by eliminating attachments between LOCs. This can be done by observing planned activity and using a weight system to determine the most likely attachment for each LOC pair. A planner is used to manipulate the input using the provided transformations until the output is obtained. This action sequence is chosen as a trace candidate. When building the trace candidate, only next LOCs are chosen that conform to possible next LOCs as determined by the refinement method. This eliminates the choice of actions which may result in reasonable plans, but which do not conform to any desirable algorithm. More than one trace candidate can be generated for each generated example.

After examining each trace candidate, an assessment can be made with regards to which states can belong in sequence. This can then inform the refinement method. The problem with the refinement method is that there are too many possible refinements and there must be some means by which choices can be reduced. The planning technique that uses examples provides a hint about which refinements are reasonable because certain choices must be made in a plan in order to achieve a goal. This allows another layer of refinements to be made which further limits the possible traces that can be used. This process of eliminating traces and eliminating refinements continues until a complete refinement sequence is possible.

In a paper titled "Automated AI Planning and Code Pattern Based Code Synthesis", the authors outline a Graphplan based strategy for assembling code [63]. In their research they focus on the looping problem and use Graphplan to assist in finding the looping parameters to automatic construct a loop.

6.2 Application to Workflows and Service Assembly

A workflow (also referred to as a business process) describes a series of steps that must be taken in some business context. The term "workflow" usually refers to steps taken by people, whereas the term "process" can mean something entirely automated. However the terms "workflow" and "business process" are generally interchangeable. The term "process automation" refers to technology used to eliminate people from the process by automating all the individual steps of the business process. The goal of "Service Assembly" is to automate the process of putting together a sequence of tasks such as business process or a workflow. In the context of programming API's, the goal automated service assembly is to dynamically sequence the programmatic access to a set of services into a single controlling service. In a recent paper by Rao et. el. [61], the authors review the concept of using planning and the PDDL language for service assembly. Pathek et al. [62] use XSP Tabled Logic and goals to assemble services.

The basic components of a workflow include tasks and business objects. Business objects can be thought of as all the real world entities that must be manipulated within the workflow. Tasks are the individual activities that comprise the workflow. Tasks have pre-conditions and effects. But in workflow, tasks can also include other data such as resource usage, time requirements etc. A workflow diagram will include a starting point, tasks, or-splits and joins, and-splits and joins, and an ending point. A workflow planner must be able to fully assemble a workflow including start points, tasks and the end point.

Workflow specification has a long history beginning with Taylor and Gantt in 1919 with their time and motion studies to assess and increase worker productivity. Since that time workflow has become a popular management tool to assist in the creation of effective operations. Workflow standards for computer systems began to emerge in the 90s when companies such as IBM began to produce modeling tools. The first version of the language "Flowmark" by IBM emerged in 1994 and allowed the complete specification of a workflow using a formal description. With the advent of process automation and data integration in the late 90's, many process languages emerged with a variety of interesting features. With the development of XML, several standards emerged including BPEL (Business Process Execution Language) from Microsoft, and ebXML from OASIS a consortium focused on language standards. In order to fully automate complex business systems, modern workflow languages bind a diverse array of technologies including payments, transactions, security, error processing, fail-over, logging, and much more. In many respects, these are like domain specific programming languages designed for commerce.

Workflow modeling is largely considered a management function rather than a Software Engineering function; however, the complexity of modern workflow systems is such that management can only be involved in the early stages of workflow design. Modern workflow systems incorporate many of the elements of a complete programming language and so it's possible to build workflow's that operate like bad software. So the need for better tools is apparent.

Like programming each step within a process must be carefully arranged into a set of interlocking tasks. But these tasks can be thought of as planning operators. Each task represents some function that must be performed and must be assembled so that the pre-conditions of one task match the effects of another. So it seems like workflow or process design can benefit from AI Planning in the same way that software systems benefit. AI Planning can be used to reason about the workflow to assist in its design and its creation. AI Planning can also be used to validate that a workflow meets a certain specification.

6.2.1 Service Assembly

Consider the process of handling a purchase order. A typical purchase order handler might employ the following tasks: Receive PO, Credit Check, No Credit Message, Inventory Check, Send Confirmation, Order Supplies and Send Back Order Notice. If the tasks are stored in a task database, the purchase order processing plan can automatically be derived by chaining together each task based on its pre-condition and effect. For example, the only task that makes sense as the first task is the Receive PO task, since nothing else can be done until a PO is received. The database would therefore contain the Receive PO task with a pre-condition that indicates a PO arrival event. The actions within the Receive PO task would load the PO information into data structures easily accessed by the other tasks. The effect of the PO task would be that the PO is ready for processing. In this particular scenario, the only task that could apply is the Credit Check task. That is because other tasks in the database have pre-conditions other than those provided as effects from the Receive PO task. Since the result of a credit check cannot be predetermined, there are two alternate paths that handle the result. Similarly, an inventory check result cannot be pre-determined and thus there are two alternate paths. So with a properly configured database of tasks, it is possible to automatically derive a complex process by providing a planner the starting point and the ending point, as well as all of the tasks that can be used to achieve a goal.

6.3 Agent Architecture with Planning

Within Software Engineering, an important research objective is to construct systems that perform well, are easy to build, and are easy to maintain. Intelligent software systems are those that automatically seek optimal performance, are self constructing through machine learning, and that adapt to new circumstances over time. While one can argue that machines cannot be intelligent because they simply carry out instructions, for the purposes of categorizing software architecture, it is necessary to distinguish between intelligent systems and all other types of systems. For example, intelligence distinguishes between plants and animals, humans and other creatures, and between programs that perform certain kinds of complex tasks. Tasks performed by enzymatic processes in a plant are extremely complex, but merely complex processes must be distinguished from processes performed by humans like solving problems or having a conversation. From a software architecture point of view, the goal in understanding intelligent systems is to develop a software architecture that is most suitable to the construction of systems that behave intelligently (See [13] for a thorough discussion of agent oriented architecture).

The most obvious characteristic of intelligent behavior is goal seeking. A system or organism that we think of as behaving intelligently is, in most cases, performing a series of activities that allow some goal to be achieved. These activities can be divided into three essential processes; sensing the environment in which actions are to occur, planning out a series of actions in the environment that will achieve the goal, and acting on the plan. Goal directed behavior involves repeating these processes continually, and in some cases simultaneously, until a goal is achieved.

If goal seeking behavior is an essential feature of intelligent behavior, we can further refine our understanding of intelligent systems by considering how planning can be achieved within a system. In order to plan, an agent (a system or organism that is able to act) must have some knowledge of the world in which the plan will be carried out. This knowledge is often referred to as "world" knowledge or more commonly as "Plan Domain" knowledge. The most essential components of plan domain knowledge include objects, agents, and activities. In a general sense, we can compare planning to Object Oriented Programming (OOP) architecture. Planning objects and agents are similar to class instantiation and planning activities are similar to methods.

Another important characteristic of intelligent behavior is adaptation or learning. As with goal seeking behavior, we think of organisms as behaving intelligently when they learn from their environment and modify their behavior to improve the likelihood that goals can be achieved. There are two ways that agents can be adaptive. One kind of adaptation is achieved through knowledge updating based on observed correspondence between activities and results. Another kind of adaptive behavior occurs when knowledge is shared between agents by training, or by direct communication. In either case, the result of learning in an intelligent agent is domain knowledge creation or modification.

One of the problems with using knowledge representation and reasoning for software engineering is the size of the search space required to reason automatically about any kind of problem. For example, the search space for the 15 puzzle is 1013. This limits most declarative representations to small domains that can be reasoned about with currently available methods and hardware. To enable practical applications of domain knowledge techniques, some restrictions on the kind of knowledge that can be expressed must be made.

Unfortunately, too much restriction to a modeling language results in a language that is not very expressive and therefore not ideal for building intelligent systems. One example of a restricted modeling language is Universal Modeling Language (UML) [6] designed for expressing architectural facts about programs. While UML has proven an extremely useful language in the software engineering community, it is still far too limited to directly express the knowledge required for intelligent systems [12] . For example, UML does not directly represent goal directed reasoning requirements, a necessary intelligent systems component.

Chapter 7

Summary and Future Directions

7.1 Debugging Plans

Getting a plan to work using FF is quite challenging due to the limited editing and debugging facilities for PDDL. All PDDL files must be edited in a text editor. There is no way to trace through a plan. Generally if there is anything wrong with a plan, the FF planner just reports an error. If there is some issue with the physics of the domain, then the end result will just simply be no plan. However, with some careful placement of predicates, it is possible to do debugging. One of the key techniques deployed for debugging plans is to use a counter ($steps$) within operations. If it is known which operations should occur at certain points in planning time, then these can simply be planned for as goals. If the goals cannot be achieved, then it is clear that something is not working in the logic.

7.2 Round Trip Modeling and the Complete Program Representation

Building a high level specification of a program is hard to do. In most cases, this is harder than programming. Most programmers find it difficult to think declaratively and so they simply write programs to meet certain architectural and test specifications. However, given a procedural code description for a routine it is possible to trace through the code automatically and derive pre and post conditions. This is already done in many contexts including program analysis, code optimization, etc. These same methods can be used to derive plan domain knowledge from hand crafted code.

A fully specified system is one where specifications, code and tests are all in agreement. Each representation places constraints on other representations. An architecture that incorporates all representations into a single knowledge repository is more adaptable to changing situations than conventional systems because automated reasoning techniques can be used to infer change requirements across representations. Automated reasoning also enforces consistent representation of knowledge. If, for example, code changes imply domain knowledge updates that conflict with code representation in other areas of a system, an adaptive system can directly make the changes or inform a coder that the changes are not allowed.

One of the grand challenges of Software Engineering and an often sought goal of tool vendors, is full round-trip modeling. To enable full round-trip editing, a model must be able to generate code and code must be able to generate a model. This is difficult to achieve since current models and development methods can generate a large number of programs for any given model and there are a large number of possible models for any given program. The modeling formalism and technique must work together to limit the selection of related programs and models. The development process should therefore consist of a user directed search through a matching model and program space to find the correct model and corresponding program that achieves the designer's goals.

7.3 Future Research

One of the limitations in AI Planning is the representation of knowledge. Inheritance and polymorphism, for example, are absent from PDDL. Planning systems also lack the ability to handle inter-relating predicates like those describing structured objects. In the refinement domain, for example, there is a strong relationship between the first predicate that matches and the later predicates within the precondition. This is because these predicates are all part of the syntax tree that defines a single expression. Since the planner does not know that these predicates are all interrelated, it must try all possible object variations prior to landing on the specific set required. Planners could be significantly enhanced with better object choice evaluators that consider the interrelationship of the predicates. A valuable next step in planning research would be to enhance the structure of the objects that are manipulated in plans. This can be done by building formal structures as part of the PDDL syntax. Description Logic could be used for representing and reasoning about complex structures in planning.

Planning systems lack the ability to directly handle recursive objects such as trees. It is still possible to build plans that involve recursive structures but the planning process is very inefficient. For example, consider a series of predicates linked together to form a syntax tree of an expression. To plan a replacement requires a series of operators that recursively walk through the structure. The end sequence of planning layers will involve successive operations on the tree; however, the planning process must check and recheck all of the predicates in the tree at every single layer. An alternative would be to include recursive predicates in pre and post conditions. These issues could also be dealt with using Description Logic for knowledge representation and reasoning.

Because conventional systems rely on pre-determined procedural logic, dynamic run-time systems are not easy to build. Real-time action sequencing is a requirement in areas such as game play, real-time business process handling and information system integration. The ideal method of handling dynamic situations is to allow the system to reason about the best code to generate and execute. This may be possible if planning could be integrated with object oriented programming tools and methods.

Machine Learning techniques can be used to automate the construction of plan domains using specialized task recorders. If a plan domain can be generated, code can automatically be synthesized. This represents a form of programming by example. This requires research into advanced recorders considering all aspects of system development, including: mathematical operations, string manipulations [BBMA], database interactions, external procedural calls, looping behavior and more. With these types of extensions, programming by example could become a standard feature of future Software Engineering environments.

Code and specifications are two different representations of a program. Other representations include comments, graphics, tests, state diagrams etc. If all representations could be designed to share a common knowledge model, this would enable an entirely new class of Software Engineering tools with a broad range of methods for developers and designers to specify their systems.

Another method of guiding the refinement process is to use *"meta-program"* statements i.e. statements that describe the program structure in some way. For example, run-time characteristics of a program can easily be inferred by counting loops. So, one kind of metric is the program run-time estimate. Other metrics include the loop count, variable count, conditional count or any assessment of code structure. Since planning allows minimization of metrics, these can be used to restrict the set of possible generated programs to those meeting these meta-program characteristics.

Chapter 8

Conclusions

Due to the changing nature of how businesses operate, new software is always required with significant business functionality, in short time frames. To meet this challenge new approaches in Software Engineering are required that can help programmers deal with complex requirements. AI Planning is a promising area of research that can be applied to the problem of automated reasoning with code. Program Refinement based on Caroll Morgan's research may be used to implement models based on AI Planning that generate code. While the present applications are simplistic the results seem promising and warrant further investigation.

If code synthesis is possible, there are many applications within the broader context of software architecture. Intelligent systems are a good example of an emerging technology that requires the dynamic programming capability of planning based systems including goal seeking and adaptive behavior, hallmarks of intelligent behavior. Merging planning with conventional object oriented programming provides new directions in software architecture that integrates emerging AI techniques such as machine learning and constraint processing.

Program refinement is a very challenging problem for current planning tools and for the planning paradigm in general. The PDDL language does not allow complex syntactical structures for direct manipulation within planning operators. In order to provide these structural elements, the nodes of a syntax tree for expressions must be described in PDDL using predicates. This limited use of structure hinders planning because the planner cannot make use of the inter-related information of predicates that make up structures. Extensions to planning that allow for these structures will both enhance the planning paradigm as well as provide more efficient methods of program refinement.

Bibliography

Bibliography

[1] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, An Eclipse Plug-in for Model Checking, IWPC 2004: 251-255, 2004.

[2] Cormac Flanagan, Automatic software model checking via constraint logic, Sci. Comput. Program, 50(1-3): 253-270, 2004.

[3] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, Automating DAML-S web services composition using SHOP2, Proceedings of the Second International Semantic Web Conference (ISWC2003), 2003.

[4] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=semantic-ex.`

[5] `http://www.w3.org/2001/sw/.`

[6] `http://www.uml.org/.`

[7] J. B. Warmer and A. G. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998.

[8] D. Nau, Y. Cao, A. Lotem, and H. Muoz-Avila, SHOP: Simple Hierarchical Ordered Planner, IJCAI-99, pp. 968-973, 1999

[9] D. Nau, H. Muoz-Avila, Y. Cao, A. Lotem, and S. Mitchell, Total-Order Planning with Partially Ordered Subtasks, IJCAI-2001. Seattle, August, 2001.

[10] Stephen J. Mellor, Stephen R. Tockey, Rodolphe Arthaud, Philippe Leblanc, An Action Language for UML: Proposal for a Precise Execution Semantics, UML 1998: 307-318, 1998.

[11] R. W. Floyd, Assigning meaning to programs, In Symp. in Applied Mathematics, volume 19, 1967.

[12] `http://ecpe.ece.iastate.edu/kothari/papers/UMLActionSemanticsIntro.pdf.`

[13] Leite, J. Omicini, A. Torroni, P. Yolum, P. (Eds.), Declarative Agent Languages and Technologies II, Second International Workshop, DALT 2004, New York, NY, USA, July 2004.

[14] T. Hoare, An axiomatic basis of computer programming, CACM, 12, 1969.

[15] Fahiem Bacchus, The AIPS '00 Planning Competition, AI Magazine, 22(3): 47-56, 2001.

[16] Jrg Hoffmann, Bernhard Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, J. Artif. Intell. Res. (JAIR), 14: 253-302, 2001.

[17] Hector J. Levesque, Planning with Loops, IJCAI, 509, 2005

[18] F. Bacchus and M. Ady, *Planning with Resources and Concurrency: A Forward Chaining Approach.* Proc IJCAI-2001, 2001.

[19] F. Bacchus, AIPS2000 Planning Competition Results in Powerpoint, F. Bacchus Presented at AIPS2000, 2000.

[20] F. Baader and U. Sattler, An overview of tableau algorithms for description logics, Studia Logica, 69:5-40, 2001.

[21] Tim Berners-Lee, James Hendler & Ora Lassila, The Semantic Web, Scientific American, 284(5):34-43, May 2001.

[22] Jim Blythe, Decision-Theoretic Planning, AI Magazine, 20(2): 37-54, 1999.

[23] A. Blum and M. Furst, Fast Planning Through Planning Graph Analysis, Artificial Intelligence, 90:281–300, 1997.

[24] Blum, A. and Furst, M., Fast Planning through Planning Graph Analysis, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, 1636-1642. Menlo Park, Calif.: International Joint Conferences Artificial Intelligence, 1995.

[25] Blum, A. L., and Langford, J. C., Probabilistic Planning in the GRAPHPLAN Framework, Proceedings of the AIPS98 Workshop on Planning as Combinatorial Search, 8-12. Pittsburgh, Penn.: Carnegie Mellon University, 1998.

[26] D. Chapman, Planning for conjunctive goals, Artificial Intelligence, 32:333-377 1990.

[27] M. d'Inverno and M. Luck, Understanding Agent Systems, Springer-Verlag, 191, 2001.

[28] Richard Fikes, Nils J. Nilsson, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, IJCAI 1971, 608-620, 1971.

[29] K. Golden, Planning and Knowledge Representation for Softbots, PHD Thesis, University of Washington, 1997.

[30] Helzerman, R.A and Harper, M.P., MUSE CSP: An Extension to the Constraint Satisfaction Problem, JAIR, Volume 5:pages 239-288, 1996.

[31] J. Hoffmann, B. Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, Journal of Artificial Intelligence Research, Volume 14, 2001.

[32] Subbarao Kambhampati, Refinement Planning as a Unifying Framework for Plan Synthesis, AI Magazine 18(2): 67-97, 1997.

[33] Henry Kautz, Bart Selman, Unifying SAT-based and Graph-based Planning, Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14-16, 1999.

[34] Henry Kautz and Bart Selman, BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving, Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98, Pittsburgh, PA, 1998.

[35] Kong, E. Stroulia, and B. Matichuk, Legacy Interface Migration: A Task-Centered Approach, Proceedings of the 8th International Conference on Human-Computer Interaction 22-27, Munich, Germany, Lawrence Erlbaum Associates, 1167-1171, August 1999.

[36] Jonas Kvarnstrm, Patrick Doherty, TALplanner 2000: A temporal logic based forward chaining planner, Annals of Mathematics and Artificial Intelligence, 30(1-4): 119-169, 2000.

[37] A. Lotem and S. Dana Nau, New advances in GraphHTN: Identifying independent subproblems in large HTN domains, In AIPS, pages 206–215, 2000.

[38] B. Matichuk, Three Methods of EAI in a Unix Environments, Online publication The HP Chronicle, `http://www.serverworldmagazine.com/hpchronicle/2000/06/eai_unix.shtml`, June 2000.

[39] Drew McDermott, PDDL: The Planning Domain Definition Language, AIPS-98 Competition Committee, draft 1.6 edition, June 1998.

[40] D. McDermott, Regression Planning, Intl. Jour. Intelligent Systems, 6:357-416, 1991.

[41] D. Nau, Y. Cao, A. Lotem, and H. Muoz-Avila, SHOP: Simple Hierarchical Ordered Planner, IJCAI-99, pp. 968-973, 1999.

[42] Bernhard Nebel, On the Compilability and Expressive Power of Propositional Planning Formalisms, JAIR, 12: 271-315, 2000.

[43] Edwin P. D. Pednault, ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus, KR 1989, 324-332, 1989.

[44] Penberthy, J., and Weld, D., UCPOP: A sound, complete, partial order planner for ADL, Proceedings of the Third International Conference on Knowledge Representation and Reasoning, (KR'92), Boston, MA., 108-114, 1992.

[45] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, B. Matichuk, Modeling the System-User Dialog Using Interaction Traces, Proceedings of the 8th Working Conference on Reverse Engineering, 2-5 October 2001, Stuttgart, Germany, 208-217, IEEE Press, October 2001.

[46] Stuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, PrenticeHall, Englewood Cliffs, NJ, 1995.

[47] Sacerdoti, E., A structure for plans and behavior, American Elsevier, New York, 1977.

[48] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, B. Matichuk, Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach, Proceedings of the 6th Working Conference on Reverse Engineering, 6-8 October 1999, Atlanta, Georgia USA, 292-302, IEEE Computer Society, October 1999.

[49] N. Sadeh-Koniecpol, D. Hildum, T.J. Laliberty, S. Smith, J. McA'Nulty, and D. Kjenstad, An Integrated Process-Planning/Production-Scheduling Shell for Agile Manufacturing, tech. report CMU-RI-TR-96-10, Robotics Institute, Carnegie Mellon University, May 1996.

[50] S. Soderland, T. Barrett, and D. Weld, The SNLP planner implementation, Contact bugsnlp@cs.washington.edu, 1990.

[51] Sussman, G. J., A computer model of skill acquisition, American Elsevier, New York, 1975.

[52] Tate, A., Interplan: A plan generation system which can deal with interactions between goals, Memo MIP-R-109, Univ. of Edinburgh, Machine Intelligence Research Unit, 1974.

[53] Tate, A., Generating project networks, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, pp. 888-893 Cambridge, MA. IJCAI, 1977.

[54] Warren, D., Generating conditional plans and programs, Proceedings of the Summer Conference on Artificial Intelligence and the Simulation of Behaviour, pp. 344-354 Edinburgh. AISB, 1976.

[55] D. S. Weld, Recent advances in AI planning, AI Magazine, 20(2):93-123, 1999.

[56] David E. Wilkins and Marie desJardins, A call for knowledge-based planning, AI Magazine, 22(1):99–115, 2001.

[57] Manuela M. Veloso, Nonlinear problem solving using intelligent casual-commitment, Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[58] A. Gerevini and D. Long, Preferences and Soft Constraints in PDDL3, Proceedings of ICAPS workshop on Planning with Preferences and Soft Constraints, 46-53, 2006.

[59] Poernomo I.H., Crossley J., Wirsing M., Adapting proofs-as-programs : The Curry-Howard protocol, (Monographs in computer science), Springer, 07-2005.

[60] Carroll Morgan, Programming from specifications, Prentice Hall International Series In Computer Science archive, 1990.

[61] Dongning Rao, Zhihua Jiang, Yunfei Jinag Fault Tolerant Web Services Composition as Planning, ISKE-2007 Proceedings, Advances in Intelligent Systems Research, 2007.

[62] Jyotishman Pathak, Samik Basu, Robyn Lutz, Vasant Honavar, Selecting and Composing Web Services through Iterative Reformulation of Functional Specifications, 18th IEEE International Conference on Volume Tools with Artificial Intelligence, 445 - 454, Nov - 2006.

[63] Jicheng Fu, Farokh B. Bastani, I-Ling Yen, Automated AI Planning and Code Pattern Based Code Synthesis, 18th IEEE International Conference on Tools with Artificial Intelligence, 540 - 546, Nov - 2006.

[64] Back, Ralph-Johan and von Wright, Joakim, Refinement Calculus: A Systematic Introduction (Back1998) Springer-Verlag, 1998.

[65] Back, Ralph-Johan, Ph.D. Thesis: On the Correctness of Refinement Steps in Program Development, bo Akademi, Department of Computer Science, 1978.

[66] C.C. Morgan and K.A. Robinson, Specification statements and refinement, IBM. Journal of Research and Development, 31(5):546-555, September 1987.

[67] Charles Rich and Richard C. Waters, The Programmer's Apprentice Project: A Research Overview, A.I. Memo 1004 (MIT AI Lab), November 1987.

Appendix A

Appendix A - Glossary of Terms

1. Action Reasoning - Algorithms that can string together sequences of actions according to the preconditions and effects of each individual action in a sequence.
2. Knowledge Representation - A method for describing knowledge in a precise way. Knowledge is a kind of data that describes some world and that can be used to make inferences about the described world.
3. Plan Domain Learning - A sub-field of Machine Learning that deals with learning knowledge database structures corresponding to plan domain formats. Observations are made of an agent's behavior. The observations are mined to infer Plan Domain structures.
4. Propositional Planning - The rules that determine an action's pre-conditions and effects are propositions about the state of the world.
5. Decision Theoretic - Refers to applications of Decision Theory, such as Decision Theoretic planners. Decision Theory is Probability Theory added to Utility Theory.
6. Probability Theory - The theory of probabilities. Used by AI researchers and statisticians to predict future results from current data.
7. Utility Theory - Assesses the likely usefulness of a particular test or action.
8. Mutex - An abbreviation for the term "Mutual Exclusion", referring to two propositions or two actions that interfere with each other. It is impossible, for example, for a rocket to be on the earth and on the moon at the same time. These propositions regarding the state of the rocket are therefore mutually exclusive, i.e. mutex.
9. Backward Chaining Algorithm - A method of problem solving whereby a solution is determined by reasoning backwards from the goal. Also known as Means-End Analysis.
10. The Blocks World Problem - A classic planning problem where a set of blocks on a table must be rearranged into a target configuration.

11. W3C - World Wide Web Consortium: An organization devoted the development and promotion of standards for the Web.

12. Description Logic - A specific kind of reasoning system that deals with the descriptions of things. Description logic is concerned with issues such as inheritance and set membership, and is emerging as a valuable method to address semantic mapping for data integration problems.

13. Semantic Web - A new initiative promoted by the W3C that integrates Description Logic, the worldwide web and XML protocols. The goal is to achieve autonomous web agents that can speak to each other using a common language, with a common reasoning mechanism, and a common method to define information.

14. Constraint Programming - An emerging programming technique whereby a solution to a problem is declared as a set of constraints on variable values within a system. A special program called a conform to the constraints.

15. Polynomial Time - If the number of objects to be considered in a problem is N, Polynomial time problems can be solved in approximately K steps where K ¿ Log N and less than NM and where M is some constant. Since M could be a big number, not all polynomial time problems are easy. The field of Complexity Theory addresses issues regarding the time and space required to solve problems.

16. Interleaved Plan - A plan requiring steps that move away from the goal prior to steps that achieve the goal.

17. Hierarchical Task Networks - A set of tasks, where some tasks can be broken down into subtasks.

18. Disjunctive Planner - Allows sets of conflicting actions to live in a particular time step.

19. Heuristic Search - A heuristic is an algorithm producing a particular result that is not easily proved mathematically, but that can be verified empirically. Problem solving in AI often involves search through a large space of possible answers to a complex question such as "What is the best next move in a chess game?" Heuristic search applies heuristics at each point in the search to determine the best next path to follow. In chess problems, for example, a useful heuristic is the piece value count at any point in the game.

20. Machine Learning - A field in AI dealing with the problem of automating the development of systems. Machine Learners are programs that can construct knowledge structures or programs that can be used to achieve certain desired results. Voice print recognition is an example of Machine Learning. After a sequence of training events, a database of recognition rules can be devised to automatically detect an individual's voice.

21. Backtracking - When an algorithm has to undo a series of steps in order to restore an earlier state, this is known as backtracking. Search often involves some form of backtracking, where various solution paths are attempted, but are undone if success cannot be achieved.

22. Branching Factor - When searching through a space of possible solutions, the number of choices at any given point is called the branching factor. In chess, for example, the branching factor is the number of next possible moves.

23. Intelligent Agent - Software with dynamically generated run-time behavior planned using a model.

Appendix B

Appendix B - A Direct Encoding of Morgan's Laws.

The following set of operators represent a comprehensive attempt at encoding all of the Morgan's Laws into operators. These encoded laws do not actually allow planning because they are incomplete. Some operations, like iteration or alternation, require much more support of related operations. I've included them here because they represent a necessary starting point for this encoding. Getting a comprehensive set of operations for Morgan's laws working as planning operators would take quite a long time. One of the significant limiting factors is the lack of recursive structure specification for constructs like syntax trees.

```
(:action followassig
:paramaters (
    ?e - term,
    ?spec - program,
    ?v - var,
    ?statement - code)
:precondition
    (and
        (getspec ?spec ?parms ?pre ?post)
        (unused ?codeline)
        (isparm ?parms,?v)
        (formulavar ?post ?v)
        (slot ?post ?v ?s))
:effect
    (and
        (not (formulavar ?post ?v))
        (forall(?evar - term)
        (formulavar ?post ?evar))
        (assign (?s (slot ?post ?e))
        (increase (lines ?code))
        (used ?statement)
        (assign (pos ?statement)  (lines ?code))
        (ctype ?statement assignment)
        (left ?statement ?v)
        (right ?statement ?e))
)
```

```
(:action make_alternation
:paramaters (
    ?stat - statement,
    ?spec - program)
:precondition
    (and
        (> ?need_alt_spec 0)
        (not (used ?spec))
        (makealtpre_goal need_alt_spec ?g)
        (makealtpre_pre need_alt_spec ?pre)
        (makealtpre_post need_alt_spec ?pre))
:effect
    (and
        (descrease ?need_alt_spec 1)
        (ctype ?stat andtype)
        (left ?stat ?g)
        (right ?statement ?pre)
))

(:action alternation
:paramaters (
    ?spec - program,
    ?goals - formulalist)
:precondition
    (and
        (= ?need_alt_spec 0)
        (used ?spec)
        (getpre ?spec ?pre)
        (implication ?pre ?goals)
        (guardlist ?goals))
:effect
    (and
    (forall(?g - goaltype)
        (when (inlist ?goals ?g)
            (and
                (increase need_alt_spec 1)
                (makealtpre_goal needspec ?g)
                (makealtpre_pre needspec ?pre)
                (makealtpre_post needspec ?post)))
)))


(:action equal_check_verify
:parameters (
    ?a ?b - stat)
:precondition
    (and
        (verifyequal ?a ?b)
        (left ?a ?lefta)
        (left ?b ?leftb)
        (right ?a ?righta)
        (right ?b ?rightb)
        (donecheck ?lefta ?leftb)
        (donecheck ?righta ?rightb))
:effect
    (and
    (not (verifyequal ?a ?b))
    (when
        (and
            (isequal ?lefta ?leftb)(isequal ?righta ?rightb))
            (isequal ?a ?b)
            (when (not (and (isequal ?lefta ?leftb)
                            (isequal ?righta ?rightb)))
                (not (isequal ?a ?b))
                (donecheck ?a ?b)))
))

(:action equal_check
:parameters (
    ?a ?b - stat)
:precondition
    (and
        (needcheck ?a ?b)
        (left ?a ?lefta)
        (left ?b ?leftb)
        (right ?a ?righta)
        (right ?b ?rightb)
        (ctype ?a ?ctypea)
        (ctype ?b ?ctypeb))
:postcondition
    (and
        (not (needcheck ?a ?b))
        (when (and (atomic ?a)(atomic ?b))
            (and
                (donecheck ?a ?b)
                (when (= ?a ?b)
                    (isequal ?a ?b))
                (when (not (= ?a ?b))
                    (not (isequal ?a ?b)))
            )
        )
        (when (and (atomic ?a) (not (atomic ?b)))
            (and
                (not (isqual ?a ?b))
                (donecheck ?a ?b)
            )
        (when (and (not (atomic ?a)) (atomic ?b))
            (and
                (not (isqual ?a ?b))
                (donecheck ?a ?b)
            )
        (when (and (not (atomic ?a))(not (atomic ?b)))
            (when (= ?ctypea ?ctypeb)
                (and
                    (needcheck ?lefta ?leftb)
                    (needcheck ?righta ?rightb)
                    (verifyequal ?a ?b)
```

```
                 )
            (when (not (= ?ctypea ?ctypeb))
                (and
                    (not (isqual ?a ?b))
                    (donecheck ?a ?b)
)))))))))

(:action finditer
:parameters (
    ?spec - program,
    ?line1 ?line2 - program)
:precondition
    (and
        (_finditer ?spec)
        (unused ?line1)
        (unused ?line2)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (not (atomic ?post))
        (left ?post ?postleft)
        (right ?post ?postright)
        (donecheck ?pre ?postleft)
        (donecheck ?pre ?postright))
:effect
    (and
    (not (_finditer ?spec))
    (when (isequal ?pre ?postleft))
        (and
            (codetype ?line1 do_codetype)
            (follow ?line1 ?spec)
            (before ?spec ?line1)
            (codetype ?line2 od_codetype)
            (follow ?spec ?line2)
            (before ?line2 ?spec)
            (guard ?spec ?postright)
            (changespec ?spec)
            (appendpre ?spec ?postright))
))

(:action iteraction_laststep
:parameters (
    ?spec - program,
    ?statpre ?statpost ?rangespec ?initspec - statement)
:precondition
    (and
        (unused ?specpre)
        (unused ?specpost)
        (unused ?rangespec)
        (unused ?initspec)
        (changespec ?spec)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (appendpre ?spec ?postright))
:effect
```

```
    (and
        (not (changespec ?spec))
        (used ?specpre)
        (used ?specpost)
        (used ?rangespec)
        (used ?initspec)
        (not (getpre ?spec ?pre))
        (not (getpost ?spec ?post))
        (getpre ?spec ?specpre)
        (getpre ?spec ?specpost)
        (left ?specpost ?pre)
        (ctype ?specpost andtype)
        (right ?specpost ?rangespec)
        (rangemin ?rangespec zero)
        (rangemid ?rangespec ?w)
        (rangemax ?rangespec ?initspec)
        (ctype ?initspec initial)
        (variable ?initspec ?w))
)
(:action iteration
:paramaters (?spec - program)
:precondition
    (and (= ?need_iter_spec 0)
        (used ?spec)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (not (atomic ?post))
        (left ?post ?postleft)
        (right ?post ?postright)
        (ctype ?post ?postctype)
:effect
    (when (= ?postctype andtype))
        (and
            (_finditer ?spec)
            (needcheck ?pre ?postleft)
            (needcheck ?pre ?postright)))
)

(:action initialvar
:parameters (
    ?x ?newname - var,
    ?spec - program,
    ?con - conprop,
    ?stat - statement)
:precond
    (and
        (usedspec ?spec)
        (getparms ?spec ?parms)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (getvartype ?x ?t)
        (free ?newname)
        (not (used ?con))
        (not (used ?stat)))
```

```
)
:effect
    (and (not (free ?x))
        (used ?con)
        (getcon ?spec ?con)
        (getconname ?con ?newname)
        (gettype ?newname ?t)
        (used ?stat)
        (ctype ?stat assignmenttype)
        (left ?stat ?x)
        (right ?stat ?newname)
        (used ?newpre)
        (getstattype ?newpre andtype)
        (left ?newpre ?pre)
        (right ?newpre ?stat)
        (not (getpre ?spec ?pre))
        (getpre ?spec ?newpre)
        (forall (?s - statement)
            (when (and (instatement ?post ?s) (left ?s ?x))
                (and (not (left ?s ?x))
                    (left ?s ?newname))))
            (forall (?s - statement)
            (when (and (instatement ?post ?s) (right ?s ?x))
                (and (not (right ?s ?x))
                    (right ?s ?newname))))
)))


; if c occurs nowhere in program spec
; then remove logical constant c from spec

(:action remove_logical_constant
:parameters (
    ?spec - program
    ?c - variable)
:precondition
    (and
        (spec_constant ?spec ?c)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (not (contains ?pre ?c))
        (not (contains ?post ?c))
    )
:effect (not (spec_constant ?spec ?c))
)

(:action fix_initial_value
:parameters (
    ?spec - program
    ?c - variable
    ?stat ?stat2 ?stat3 - statement
    ?ttype - termtype )
:precondition
    (and
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (implication_member ?pre ?stat ?ttype)
        (unused ?c)
        (unused ?stat2)
        (unused ?stat3)
    )
:effect
    (and
        (used ?c)
        (used ?stat2)
        (used ?stat3)
        (ctype ?stat2 andtype)
        (left ?stat2 ?pre)
        (right ?stat2 ?stat3)
        (ctype ?stat3 equaltype)
        (left ?stat3 ?c)
        (right ?stat3 ?stat)
        (not (getpre ?spec ?pre))
        (getpre ?spec ?stat2)
    )
)

(:action introduce_logical_constant
:parameters (
    ?spec - program
    ?t - termtype
    ?pre2 - statement)
:precondition
    (and
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (getparsm ?spec ?parms)
        (implication_exist ?pre ?c ?t ?pre2)
        (not (contains ?pre ?c))
        (not (contains ?post ?c))
        (not (contains ?parms ?c))
    )
:effect
    (and
        (spec_constant ?spec ?c)
        (constant_type ?c ?t)
        (not (getpre ?spec ?pre))
        (getpre ?spec ?pre2)
    )
)

(:action checkstrengthen
:paramaters (
    ?spec - program
    ?stat - statement)
:precondition
    (and
        (check_strengthen ?spec)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (right ?stat ?postprime)
:effect
    (and
        (not (check_strengthen ?spec))
        (when (implication ?stat ?post)
            (and
                (not (getpost ?spec ?post))
                (getpost ?spec ?postprime)
            )
        )
    )
)


(:action strengthen_postcondition
:parameters (
    ?spec - program
    ?postprime - statement
    ?stat - statement)
:precondition
    (and
        (not (check_strengthen ?spec))
        (unused ?stat)
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
    )
:effect
    (and
        (used ?stat)
        (left ?stat ?pre)
        (ctype ?stat andtype)
        (right ?stat ?postprime)
        (check_strengthen ?spec)
        (strengthenspec ?spec ?stat)
    )
)

(:action done_replace_check
:precondition
    (and
        (need_replace_check ?stat)
        (done_replace_check_left ?stat)
        (done_replace_check_right ?stat)
    )
:effect
    (not (need_replace_check ?stat))
    (ready_to_test ?stat)
)


(:action replace_check
:parameters (
    ?stat ?newstat_left ?newstat_right - statement)
:precondtion
    (and
        (need_replace_check ?stat)
        (replacevar ?stat ?w)
        (replaceterm ?stat ?e)
        (left ?stat ?leftstat)
        (right ?stat ?rightstat)
        (ctype ?stat ?stattype)
        (getreplacement ?stat ?newstat)
        (unused ?newstat_left)
        (unused ?newstat_right)
    )

:effect
    (and
        (used ?newstat)
        (ctype ?newstat ?stattype)
        (when (atomic ?leftstat)
            (and
            (done_replace_check_left ?stat)
            (when (= ?leftstat ?w)
                (and
                    (left ?stat ?e)
                )
            )
            )
        )
        (when (atomic ?rightstat)
            (and
            (done_replace_check_right ?stat)
            (when (= ?rightstat ?w)
                (and
                    (right ?stat ?e)
                )
            )
            )
        )

        (when (not (atomic ?leftstat))
            (and
            (need_replace_check ?leftstat)
            (replacevar ?leftstat ?w)
            (replaceterm ?leftstat ?e)
            (left ?newstat ?newstat_left)
            (getreplacement ?leftstat ?newstat_left)
            )
        )
        (when (not (atomic ?rightstat))
            (and
            (need_replace_check ?rightstat)
            (replacevar ?rightstat ?w)
            (replaceterm ?rightstat ?e)
            (right ?newstat ?newstat_right)
```

```
                (getreplacement ?rightstat ?newstat_right)
                )
            )
        )
)


(:action assignment_check
:parameters (
        ?spec - program
        ?e - statement
        ?w - var
        ?postrep - statement)
:precondition
        (and
            (not (check_assignment ?spec))
            (not (completed ?spec))
            (inframe ?spec ?w)
            (getpost ?spec ?post)
            (not (used ?postrep))
        )
:effect
        (and
            (need_replace_check ?post)
            (toplevel ?post)
            (replacevar ?post ?w)
            (replaceterm ?post ?e)
            (getreplacement ?post ?postrep)
        )
)

(:action assignment
:parameters (
        ?stat - statement
        ?code - program)
:precondtion
        (and
            (toplevel ?stat)
            (ready_to_test ?stat)
            (getspec ?stat ?spec)
            (getpre ?spec ?pre)
            (getreplacement ?stat ?replacement)
            (implication ?pre ?replacement)
            (replacevar ?stat ?w)
            (replaceterm ?stat ?e)
            (unused ?code)
        )
:effect
        (and
            (used ?code)
            (not (toplevel ?stat))
            (not (ready_to_test ?stat))
            (completed ?spec)
            (spec_code ?spec ?code)
```

```
          (left ?code ?w)
          (right ?code ?e)
          (ctype ?code assigntype)
     )
)
```

Appendix C

Appendix C - Minimum of List Domain

The following domain file describes all the operators that were used in the experiment "Selecting the Minimum from a list". There is some overlap with the "Sequential Composition Test". However, none of the sequential composition operators were actually required for the minimum list experiment. Nevertheless, the domain that was used for the test run is produced here in its entirety.

```
(define (domain minoflist)
(:requirements :typing :fluents :equality :conditional-effects :adl)
(:types atype formula program variableset codeexpression cnt)
(:types variable specformula - formula)

(:predicates
        (replaceuse ?e - formula)
        (replacedone ?v - variable)
        (inframe ?vs - variableset ?f - formula)
        (constant ?f - formula)
        (testassign)
        (used ?spec - program)
        (usedcode ?c - codeexpression)
        (usedformula ?f - formula)
        (getpre ?spec - program ?pre - specformula)
        (getpost ?spec - program ?pre - specformula)
        (getparms ?spec - program ?parmlist - variableset)
        (inset ?varset - variableset ?var - variable)
        (completed ?spec - program)
        (coded-spec ?spec - program ?c - codeexpression)
        (code-left ?code - codeexpression ?w -formula)
        (code-right ?code - codeexpression ?e - formula)
        (code-type ?code - codeexpression ?type - atype)
        (code-spec ?spec2 - program ?c - codeexpression)
        (follows ?spec1 ?spec2 - program)
        (replacevar ?stat - formula ?w - variable)
        (replaceterm ?stat - formula ?e - formula)
;       (need_replace_check ?post - formula)
        (need_replace_check_left ?post ?top - formula)
        (need_replace_check_right ?post ?top - formula)
        (toplevel ?specformula - formula)
;       (getreplacement ?post ?postrep - formula)
        (right ?newstat ?newstat_right - formula)
        (left ?newstat ?newstat_right - formula)
        (ctype ?stat - formula ?t - atype)
        (complex ?stat - formula)
        (done_replace_check_left ?stat - formula)
        (done_replace_check_right ?stat - formula)
        (ready_to_test ?stat - formula)
        (check-assignment ?spec - program)
        (makenewleft ?stat ?leftstat - formula)
;       (replace ?f - formula)
        (replace_left ?f - formula)
        (replace_right ?f - formula)
        (cleanup ?spec1 ?spec2 - program)
        (same-left ?f1 ?f2 - formula)
        (same-right ?f1 ?f2 - formula)
        (same ?f1 ?f2 - formula)
        (mark ?f - formula)
        (informula ?f1 ?f2 - formula)
        (loop ?spec - program)
        (loopcall ?spec - program ?c - codeexpression)
) ;41

(:functions (steps) (repdepth ?stat - formula))

(:action skipcheck
:parameters (
    ?spec - program,
    ?pre ?post - specformula
    ?c - codeexpression)
:precondition
    (and
        (> (steps) 12)
        (used ?spec)
        (not (usedcode ?c))
        (not (completed ?spec))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (not (mark ?post))
        (or
            (= ?pre ?post)
            (same ?pre ?post)
        )
    )
:effect
    (and
        (completed ?spec)
        (code-type ?c Skip)
        (code-spec ?spec ?c)
    )
)

(:action make-true-post
:parameters (
    ?spec - program
    ?pre ?post - specformula
    ?leftside
    ?rightside - formula
    ?ftype - atype)
:precondition
    (and
        (> (steps) 12)
        (used ?spec)
        (not (completed ?spec))
        (not (toplevel ?post))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (complex ?post)
        (left ?post ?leftside)
        (right ?post ?rightside)
        (ctype ?post Assigntype)
        (not (mark ?post))
        (or
            (= ?leftside ?rightside)
            (same ?leftside ?rightside)
        )
    )
:effect
    (and
        (not (getpre ?spec ?pre))
        (not (getpost ?spec ?post))
        (getpre ?spec True)
        (getpost ?spec True)
    )
)

(:action make-true2-post
:parameters (
    ?spec - program
    ?pre ?post - specformula
    ?leftside ?rightside - formula
    ?ftype - atype)
:precondition
    (and
        (> (steps) 12)
        (used ?spec)
        (not (completed ?spec))
        (not (toplevel ?post))
        (getpre ?spec ?pre)
        (getpost ?spec ?post)
        (complex ?post)
        (left ?post ?leftside)
        (right ?post ?rightside)
        (ctype ?post Andtype)
        (not (mark ?post))
        (= ?leftside ?rightside)
        (= ?leftside Trueformula)
    )
:effect
    (and
        (not (getpre ?spec ?pre))
        (not (getpost ?spec ?post))
        (getpre ?spec True)
        (getpost ?spec True)
    )
)


(:action finish_assignment
:parameters (
    ?spec ?spec2 - program
    ?post - specformula
    ?w - variable
    ?e - formula
    ?c - codeexpression)
:precondition
    (and
        (or (= (steps) 4) (= (steps) 8)(= (steps) 12))
        (testassign)
        (not (used ?spec2))
        (check-assignment ?spec)
        (getpost ?spec ?post)
        (toplevel ?post)
        (or
            (replace_right ?post)
            (replace_left ?post)
        )

        (not (usedcode ?c))
        (replacevar ?post ?w)
        (replaceterm ?post ?e)
        (not (= ?w ?e))

    )
:effect
    (and
        (increase (steps) 1)
        (not (replacevar ?post ?w))
        (not (replaceterm ?post ?e))
        (not (testassign))
        (not (toplevel ?post))
        (follows ?spec ?spec2)
        (not (check-assignment ?spec))
        (cleanup ?spec ?spec2)
        (forall (?f - formula)
            (when (and (informula ?post ?f)(complex ?f))
                (mark ?f)
            )
        )

        (forall (?f - formula)
            (and
                (not (need_replace_check_left ?f ?post))
                (not (need_replace_check_right ?f ?post))
            )
        )

        (when
            (replace_right ?post)
            (not (replace_right ?post))
        )

        (when
            (replace_left ?post)
            (not (replace_left ?post))
        )
        (usedcode ?c)
        (used ?spec2)
        (completed ?spec2)
        (code-spec ?spec2 ?c)
        (code-left ?c ?w)
        (code-right ?c ?e)
        (code-type ?c Assigntype)
        (replaceuse ?e)
    )
)

(:action assignment_check
:parameters (
    ?spec - program,
    ?e - formula,
    ?w - variable,
    ?post - specformula
    ?vs - variableset)
:precondition
    (and
        (not (testassign))
        (used ?spec)
        (not (completed ?spec))
        (not (check-assignment ?spec))
        (getparms ?spec ?vs)
        (inset ?vs ?w)
        (getpost ?spec ?post)
        (complex ?post)
        (not (complex ?e))
        (not (= ?w ?e))
        (not (replaceuse ?e))
        (not (replacedone ?w))
        (or
            (inframe ?vs ?e)
            (constant ?e)
            (informula ?post ?e)
        )
;       (not (usedformula ?e))
    )
```

```
:effect
    (and
        (increase (steps) 1)
        (testassign)
        (check-assignment ?spec)
        (need_replace_check_left ?post ?post)
        (need_replace_check_right ?post ?post)
        (toplevel ?post)
        (replacevar ?post ?w)
        (replaceterm ?post ?e)
    )
)


(:action replace_check_left
:parameters (
    ?top - specformula
    ?stat ?leftstat - formula
    ?w - variable ?e - formula)
:precondition
    (and
        (testassign)
        (usedformula ?stat)
        (usedformula ?leftstat)
        (toplevel ?top)
        (need_replace_check_left ?stat ?top)
        (left ?stat ?leftstat)
        (informula ?stat ?w)
        (replacevar ?top ?w)
        (replaceterm ?top ?e)
        (not (= ?w ?e))
    )

:effect
    (and
        (increase (steps) 1)
        (not (need_replace_check_left ?stat ?top))
        (when (not (complex ?leftstat))
            (done_replace_check_left ?stat)
        )
        (when (and (not (complex ?leftstat))(= ?leftstat ?w))
            (and
                (not (left ?stat ?leftstat))
                (left ?stat ?e)
                (replace_left ?top)
                (usedformula ?e)
                (replacedone ?w)
                (not (informula ?stat ?w))
                (not (informula ?top ?w))
                (informula ?stat ?e)
                (informula ?top ?e)
            )
        )
        (when (complex ?leftstat)
            (and
            (need_replace_check_left ?leftstat ?top)
            (need_replace_check_right ?leftstat ?top)
            )
        )
    )
)

(:action replace_check_right
:parameters (
    ?top - specformula
    ?stat ?rightstat - formula
    ?w - variable
    ?e - formula)
:precondition
    (and
        (testassign)
        (usedformula ?stat)
        (usedformula ?rightstat)
        (toplevel ?top)
        (need_replace_check_right ?stat ?top)
        (right ?stat ?rightstat)
        (informula ?stat ?w)
        (replacevar ?top ?w)
        (replaceterm ?top ?e)
        (not (= ?w ?e))
    )

:effect
    (and
        (increase (steps) 1)
        (not (need_replace_check_right ?stat ?top))
        (when (not (complex ?rightstat))
            (done_replace_check_right ?stat)
        )
        (when (and (not (complex ?rightstat))(= ?rightstat ?w))
            (and
                (not (right ?stat ?rightstat))
                (right ?stat ?e)
                (replace_right ?top)
                (usedformula ?e)
                (replacedone ?w)
                (not (informula ?stat ?w))
                (informula ?stat ?e)
                (not (informula ?top ?w))
                (informula ?top ?e)
            )
        )
        (when (complex ?rightstat)
            (and
            (need_replace_check_left ?rightstat ?top)
            (need_replace_check_right ?rightstat ?top)
            )
        )
    )
)

(:action FOL_swap
:parameters (
    ?top - specformula
        ?f1 ?leftstat ?rightstat - formula
        ?t - atype)
:precondition
    (and
        (> (steps) 12)
        (not (testassign))
        (informula ?top ?f1)
        (not (toplevel ?top))
        (ctype ?f1 ?t)
        (left ?f1 ?leftstat)
        (right ?f1 ?rightstat)
        (not (= ?leftstat ?rightstat))
        (or
            (= ?t Assigntype)
            (= ?t Ortype)
            (= ?t Andtype)
        )
    )
:effect
    (and
;       (increase (steps) 1)
        (not (left ?f1 ?leftstat))
        (not (right ?f1 ?rightstat))
        (left ?f1 ?rightstat)
        (right ?f1 ?leftstat)
        (forall (?stest - formula)
            (and
            (not (same-left ?f1 ?stest))
            (not (same-left ?stest ?f1))
            (not (same-right ?f1 ?stest))
            (not (same-right ?stest ?f1))
            )
        )
    )
)

(:action make_same_left
:parameters (
    ?f1 ?f2 ?leftf1 ?leftf2 - formula)
:precondition
    (and
        (> (steps) 12)
        (not (testassign))
        (complex ?f1)
        (complex ?f2)
        (not (= ?f1 ?f2))
        (not (toplevel ?f1))
        (not (toplevel ?f2))
```

```
          (left ?f1 ?leftf1)
          (left ?f2 ?leftf2)
          (or
              (= ?leftf1 ?leftf2)
              (and
              (same ?leftf1 ?leftf2)
              (same ?leftf2 ?leftf1)
              )
          )
          (not (and
              (same-left ?f1 ?f2)
              (same-left ?f2 ?f1)
          ))

     )
:effect
     (and
          (increase (steps) 1)
          (same-left ?f1 ?f2)
          (same-left ?f2 ?f1)
     )
)


(:action make_same_right
:parameters (
    ?f1 ?f2 ?rightf1 ?rightf2 - formula)
:precondition
     (and
          (> (steps) 12)
          (not (testassign))
          (complex ?f1)
          (complex ?f2)
          (not (= ?f1 ?f2))
          (not (toplevel ?f1))
          (not (toplevel ?f2))
          (right ?f1 ?rightf1)
          (right ?f2 ?rightf2)
          (or
              (= ?rightf1 ?rightf2)
              (and
              (same ?rightf1 ?rightf2)
              (same ?rightf2 ?rightf1)
              )
          )
          (not (and
              (same-right ?f1 ?f2)
              (same-right ?f2 ?f1)
          ))
     )
:effect
     (and
          (increase (steps) 1)
          (same-right ?f1 ?f2)
```

```
        (same-right ?f2 ?f1)
    )
)

(:action make_same
:parameters (?f1 ?f2 - formula)
:precondition
    (and
        (> (steps) 12)
        (not (testassign))
        (complex ?f1)
        (complex ?f2)
        (not (toplevel ?f1))
        (not (toplevel ?f2))
        (not (= ?f1 ?f2))
        (same-left ?f1 ?f2)
        (same-left ?f2 ?f1)
        (same-right ?f1 ?f2)
        (same-right ?f2 ?f1)
        (not (and
            (same ?f1 ?f2)
            (same ?f2 ?f1)
        ))
    )
:effect
    (and
        (increase (steps) 1)
        (same ?f1 ?f2)
        (same ?f2 ?f1)
        (not (mark ?f1))
        (not (mark ?f2))
    )
)

(:action make_formula_left_true
:parameters (
    ?f ?f1 ?leftf1 ?rightf1 - formula
    ?t - atype)
:precondition
    (and
        (> (steps) 12)
        (not (testassign))
        (not (toplevel ?f))
        (left ?f ?f1)
        (ctype ?f1 Assigntype)
        (left ?f1 ?leftf1)
        (right ?f1 ?rightf1)
        (or
            (= ?leftf1 ?rightf1)
            (same ?leftf1 ?rightf1)
        )
    )
:effect
    (and
        (increase (steps) 1)
        (left ?f Trueformula)
        (not (left ?f ?f1))
        (not (ctype ?f1 Assigntype))
        (not (left ?f1 ?leftf1))
        (not (right ?f1 ?rightf1))
        (when (mark ?f) (not (mark ?f)))
    )
)

(:action make_formula_right_true
:parameters (
    ?f ?f1 ?leftf1 ?rightf1 - formula
    ?t - atype)
:precondition
    (and
        (> (steps) 12)
        (not (testassign))
        (not (toplevel ?f))
        (right ?f ?f1)
        (ctype ?f1 Assigntype)
        (left ?f1 ?leftf1)
        (right ?f1 ?rightf1)
        (or
            (= ?leftf1 ?rightf1)
            (same ?leftf1 ?rightf1)
        )
    )
:effect
    (and
        (increase (steps) 1)
        (right ?f Trueformula)
        (not (right ?f ?f1))
        (not (ctype ?f1 Assigntype))
        (not (left ?f1 ?leftf1))
        (not (right ?f1 ?rightf1))
        (when (mark ?f) (not (mark ?f)))
    )

)

(:action allcheck
:parameters (
    ?spec ?spec2 - program
    ?newvar ?listvar - variable
    ?post - specformula
    ?funcall - formula
    ?c ?c2 - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (loop ?spec))
        (not (usedcode ?c))
        (not (used ?spec2))
        (not (usedcode ?c2))
        (not (usedformula ?newvar))
        (getpost ?spec ?post)
        (left ?post ?funcall)
        (ctype ?post All)
        (right ?post ?listvar)
    )
:effect
    (and
        (usedcode ?c)
        (usedcode ?c2)
        (code-type ?c Loop)
        (code-left ?c ?listvar)
        (code-right ?c ?newvar)
        (code-spec ?spec2 ?c)
        (code-spec ?spec ?c2)
        (code-type ?c2 Endloop)
        (follows ?spec2 ?spec)
        (usedformula ?newvar)
        (not (ctype ?post All))
        (ctype ?post Apply)
        (not (right ?post ?listvar))
        (right ?post ?newvar)
    )
)

(:action apply
:parameters (
    ?spec - program ?post - specformula
    ?f ?leftf - formula
    ?t - atype)
:precondition
    (and
        (used ?spec)
        (getpost ?spec ?post)
        (left ?post ?f)
        (left ?f ?leftf)
        (ctype ?f ?t)
        (ctype ?post Apply)
    )
:effect
    (and
        (not (ctype ?post Apply))
        (ctype ?post ?t)
        (not (left ?post ?f))
        (left ?post ?leftf)
    )
)


(:action lte
:parameters (
    ?spec - program
    ?var - variable
    ?post - specformula
    ?comp ?comp2 - formula
    ?rightf - formula
    ?c - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (usedcode ?c))
        (getpost ?spec ?post)
        (left ?post ?var)
        (ctype ?post LTE)
        (right ?post ?rightf)
        (not (usedformula ?comp))
        (not (= ?comp ?comp2))
        (not (usedformula ?comp2))
    )
:effect
    (and
        (code-type ?c If)
        (code-left ?c ?comp)
        (left ?comp ?var)
        (right ?comp ?rightf)
        (code-right ?c ?comp2)
        (left ?comp2 ?var)
        (right ?comp2 ?rightf)
        (ctype ?comp2 Assigntype)
        (completed ?spec)
        (code-spec ?spec ?c)
        (usedcode ?c)
    )
)

(:action Move
:parameters (
    ?spec ?spec2 - program
    ?post - specformula
    ?f ?leftf ?rightf - formula
    ?c - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (usedcode ?c))
        (not (used ?spec2))
        (getpost ?spec ?post)
        (left ?post ?f)
        (ctype ?post Move)
        (right ?post ?rightf)
        (left ?post ?rightf)
    )
:effect
    (and
        (completed ?spec2)
        (not (left ?post ?f))
        (left ?post ?leftf)
        (code-type ?c Pair)
        (code-left ?c ?newvar)
        (code-right ?c ?newvar2)
    )
)

(:action Pair
:parameters (
    ?spec - program
    ?post - specformula
    ?f ?leftf ?rightf - formula
    ?c ?2 - codeexpression)
:precondition
    (and
        (used ?spec)
        (not (usedcode ?c2))
        (getpost ?spec ?post)
        (left ?post ?f)
        (ctype ?post Pair)
        (code-spec ?spec ?c)
        (left ?f ?leftf)
    )
:effect
    (and
        (not (left ?post ?f))
        (left ?post ?leftf)
        (code-type ?c2 Pair)
        (code-left ?c2 ?newvar)
        (code-right ?c2 ?newvar2)
        (usedcode ?c2)
        (not (code-spec ?spec ?c))
        (code-spec ?spec ?c2)
        (next ?c2 ?c)
    )
))
```