Generation and Verification of Plans with Loops

by

Yuxiao Hu

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2012 by Yuxiao Hu
Abstract

Generation and Verification of Plans with Loops

Yuxiao Hu

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2012

This thesis studies planning problems whose solution plans are program-like structures that contain branches and loops. Such problems are a generalization of classical and conditional planning, and usually involve infinitely many cases to be handled by a single plan. This form of planning is useful in a number of applications, but meanwhile challenging to analyze and solve. As a result, it is drawing increasing interest in the AI community.

In this thesis, I will give a formal definition of planning with loops in the situation calculus framework, and propose a corresponding plan representation in the form of finite-state automata. It turns out that this definition is more general than a previous formalization that uses restricted programming structures for plans.

For the verification of plans with loops, we study a property of planning problems called finite verifiability. Such problems have the property that for any candidate plan, only a finite number of cases need to be checked in order to conclude whether the plan is correct for all the infinitely many cases. I will identify several forms of finitely-verifiable classes of planning problems, including the so-called one-dimensional problems where an unknown and unbounded number of objects need independent processing. I will also show that this property is not universal, in that finite verifiability of less restricted problems would mean a solution to the Halting problem or an unresolved mathematical conjecture.

For the generation of plans with loops, I will present a novel nondeterministic al-
algorithm which essentially searches in the space of the AND/OR execution trees of an incremental partial plan on a finite set of example instances of the planning problem. Two different implementations of the algorithm are explored for search efficiency, namely, heuristic search and randomized search with restarts. In both cases, I will show that the resulting planner generates compact plans for a dozen benchmark problems, some of which are not solved by other existing approaches, to the best of our knowledge.

Finally, I will present generalizations and applications of the framework proposed in this thesis that enables the analysis and solution of related planning problems recently proposed in the literature, namely, controller synthesis, service composition and planning programs. Notably, the latter two require possibly non-terminating execution in a dynamic environment to provide services to coming requests. I will show a generic definition and planner whose instantiation accommodates and solves all the three example applications. Interestingly, the instantiations are competitive with, and sometimes even outperform, the original tailored approaches proposed in the literature.
Acknowledgements

I would like to express my tremendous appreciation to my thesis supervisor, Professor Hector Levesque, for all the guidance and help that he offered in the long journey towards this degree. His open attitude, insightful directions, encouraging discussions and caring support have made my five-year of otherwise uneventful exploration in the world of research both inspiring and enjoyable.

I would also like to thank Professors Fahiem Bacchus, Eric Hehner and Sheila McIlraith for their advice and help in my research and teaching experiences, especially for their feedback in my final thesis and earlier research work.

Many thanks to Professors Yongmei Liu and Giuseppe De Giacomo, who brought me fresh ideas and offered much cooperation during their Sabbatical visits. Giuseppe initiated the discussion on applying my earlier work to a much broader setting, and greatly helped me to finally materialize the ideas in this thesis.

I want to say “thank you” to all fellow graduate student in the AI lab of University of Toronto, including Jorge Baier, Arnold Binas, Christian Fritz, Eric Hsu, Christian Muise, Shirin Sohrabi, and Stavros Vassos. It was you who made my life as a graduate student colorful and fun.

I also want to thank Professor Gerhard Lakemeyer, who was my supervisor in my Master’s program at RWTH Aachen, and encouraged me to further my research in this Ph.D. program. Without you, this thesis would not have existed.

Last but not least, I want to express my thanks to my wife Jessica, and my parents. Your love, encouragement and support gave me the most confidence and power to overcome difficulties, and will always be my most precious asset.

Thank you, dear all.
Contents

1 Introduction ........................................ 1
   1.1 Approach and Contribution .................... 3
   1.2 Thesis Outline ................................ 5

2 Related Work ....................................... 6
   2.1 From Classical Planning to Planning with Loops 6
   2.2 Generating Plans with Loops .................. 8
       2.2.1 Deductive Approaches ..................... 8
       2.2.2 Non-deductive Approaches ................. 10
   2.3 Reasoning about Correctness of Plans with Loops 14
   2.4 Other Related Work ............................ 16

3 Formal Preliminaries ............................... 19
   3.1 Situation Calculus ............................... 19
       3.1.1 The Language ............................... 20
       3.1.2 Basic Action Theories ..................... 21
       3.1.3 The Projection Problem .................... 24
   3.2 Reasoning about Plans with Loops ............... 27
       3.2.1 Robot Programs ............................. 29
       3.2.2 FSA Plans ................................. 32
       3.2.3 Comparison of the Plan Representations .... 35
3.3 Concluding Remarks ........................................... 40

4 Finite Verifiability .............................................. 41
  4.1 Finite Verifiability with Decreasing Parameters .......... 42
    4.1.1 One-Dimensional Planning Problems .................. 42
    4.1.2 k-Dimensional Planning Problems ..................... 56
  4.2 Problems with Restricted Increasing Parameters .......... 58
  4.3 Non-Finitely Verifiable Problems ............................. 67
    4.3.1 One-Dimensional Problems with Increasing Effects .... 67
    4.3.2 Two-Dimensional with Unrestricted Incrementability ... 72
  4.4 Concluding Remarks ...................................... 76

5 Generating Plans with Loops ................................... 78
  5.1 Review of Levesque’s KPLANNER ................................. 79
  5.2 Basic FSAPLANNER Algorithm ................................ 82
    5.2.1 Algorithm ........................................ 83
    5.2.2 Implementation and Experiments ....................... 87
    5.2.3 Discussion ...................................... 89
  5.3 Planning via Heuristic Search ................................ 91
    5.3.1 Adding Heuristics to the Basic Algorithm ............ 92
    5.3.2 Experimental Results ................................ 96
  5.4 Planning via Random Restart ................................ 101
    5.4.1 The Random-Restart Algorithm ......................... 102
    5.4.2 Experimental Results ................................ 105
  5.5 Concluding Remarks ...................................... 105

6 Generalizations and Applications ............................. 107
  6.1 A Unifying Definition of Generalized Planning ............. 109
    6.1.1 Relationship to Existing Forms of Planning .......... 117
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.2</td>
<td>One-Dimensional Planning Problems Revisited</td>
<td>121</td>
</tr>
<tr>
<td>6.2</td>
<td>A Generic Algorithm for Controller Synthesis</td>
<td>125</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Problem Representation</td>
<td>129</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Generic Solver</td>
<td>131</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Examples</td>
<td>134</td>
</tr>
<tr>
<td>6.3</td>
<td>Concluding Remarks</td>
<td>144</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>145</td>
</tr>
<tr>
<td>A</td>
<td>Example Plans</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>159</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

An important area of artificial intelligence is automated planning, where the task is to build agents that can act in a dynamic environment to achieve certain goals that otherwise require human intelligence.

The majority of research in planning focuses on classical planning, where the exact scenario is known, including complete information about the initial state of the world, deterministic actions and a reachability goal. So no run-time sensing or observation is necessary, and a solution plan is just a sequence of actions, which, if performed in that order from the initial state, will lead to a state where the goal condition is satisfied. However, the plan usually works for that given specific scenario only, and may fail on a slightly different problem.

Consider a simple logistics domain where a truck can move objects around one at a time. A classical planning example may involve object $o_1$ initially at home and object $o_2$ in the office, and request to make $o_1$ be in the office and $o_2$ at home. The following action sequence effectively solves this problem: first move the truck to home, load $o_1$, then move to the office, unload $o_1$ and load $o_2$, and finally move home and unload $o_1$ there. However, it will not work if $o_2$ was initially at home, or if we have five objects instead of two, although they are in a sense similar and related to the original problem.
This raises the question whether there is a more general form of plan which solves the class of "similar problems" altogether. After all, if a human is asked to describe what strategy can be used to solve all logistics problems, one would probably give a procedure like the following:

1. find the location of an unprocessed object, if no such object exists, stop;
2. move the truck to that location;
3. load the object;
4. find its destination;
5. move to the destination;
6. unload the object from the truck;
7. go to step 1.

This informal plan has a branching point at Step 1 and a loop formed by the goto instruction at Step 7. It is not hard to see that if we follow the steps in this procedural plan, we will finally stop in a state where the goal is achieved for any logistics problem, no matter how many objects there are and where they initially are and need to be located.

Sometimes, it is less obvious whether a general plan exists for a planning domain. For example, consider the following “striped tower” puzzle.

In a blocks world with three stacks $A$, $B$ and $C$, a robot can tell whether a stack is empty, and the colour of the block in its hand, if it is holding one. It can also pick up a top block on $A$ and $B$ if its hand is empty, and put a block in its hand onto $B$ and $C$. (Note that it is not allowed to pick from $C$ or put onto $A$.) Initially, $A$ contains an equal number of red and blue blocks stacked in an arbitrary order, and $B$ and $C$ are both empty. The goal is to
make all the blocks be on stack $C$ so that $C$ is a striped tower, i.e., $C$ contains alternating blue and red blocks with a red block at the bottom.

It is possible, but somewhat challenging, to manually come up with a strategy that always achieves the goal no matter how many blocks of either colour is initially on $A$ and in what order they are stacked. An automatically generated plan is shown in the Appendix, which I will discuss in more detail in Chapter 5.

Given the well-established results in reasoning about actions and classical planning, it is interesting to study this new, general form of planning, which is the main motivation of my work here. In this thesis, I will investigate this type of planning that involves branches and loops. More specifically, I will try to answer the following questions:

- What is a planning problem that may require loops in a plan? How do we define plans with loops and their semantics?

- When is a plan correct for a planning problem? Is it possible to validate a program-like plan for a problem by a finite means? If so, under what circumstances can we do so?

- Can we automatically and efficiently generate plans with loops that solves a class of classical planning problems?

- Are there other applications of plans with loops? If so, how do we understand and generate such plans in the more general setting?

1.1 Approach and Contribution

The approach described in this thesis is based on Levesque’s earlier work on the formalization of planning in the presence of loops [Levesque, 1996] and his KPLANNER [Levesque, 2005]. The main contributions of this thesis include the following aspects.
• We give a formal characterization of planning in the presence of loops in the situation calculus framework, which includes a new plan representation in the form of finite-state machines. We show that this definition is more general than Levesque’s robot programs [Hu and Levesque, 2009], which justifies using this definition in the body of this thesis.

• For reasoning about the correctness of plans with loops, we study a property of the planning problems called finite verifiability, which enables us to test a plan for only finitely many cases in order to conclude whether it is correct for all the possibly infinitely many cases. We show several problem classes that have this property, including the one-dimensional problems which model all the planning problems involving an unbounded number of similar objects that require independent processing, like the logistics problem above [Hu and Levesque, 2010]. We also show that finite verifiability is not a universal property, in that finite verifiability for certain types of problems would translate to a solution to the halting problem or a proof of a long-standing mathematical conjecture.

• We propose a nondeterministic algorithm for generating plans with loops, which translates to a search algorithm in the space of execution trees of incremental finite-state plans. I will investigate two implementations of this search algorithm, namely, heuristic search [Hu and Levesque, 2009] and randomized search with restarts. Both planners are tested against a suite of a dozen benchmark problems of varying difficulty. The planners are able to solve most of the problems efficiently, including many challenging ones that are not solved by other related approaches.

• We generalize the results to other generalized planning problems that require loops in their plans. We give a definition of generalized planning that is independent of specific formalisms, and show that many existing forms of standard planning and planning with loops are instances of this definition [Hu and De Giacomo, 2011a].
Chapter 1. Introduction

It is also able to model problems with temporally-extended and long-running goals. Finally, we will present a generic algorithm for solving generalized planning problems under this definition with a reasonable finiteness assumption, and show that instantiations of the algorithm results in efficient planners on three recently proposed application domains, which sometimes outperforms the tailored approaches [Hu and De Giacomo, 2011b].

1.2 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, I will survey the existing approaches and other related work. Chapter 3 reviews the situation calculus, the formal foundation for later chapters, including a survey of Levesque’s earlier work on planning with loops.

The second half of Chapter 3 and Chapters 4–6 are the main body of this thesis, and each addresses one contribution listed above.

Chapter 7 summarizes the whole thesis.

The problem definition and comparison to Levesque’s earlier formalization in Chapter 3 was first discussed in [Hu and Levesque, 2009]. This definition was later formalized in [Hu and Levesque, 2010], where it was used as the foundation for the finite verifiability of one-dimensional planning problems covered in Chapter 4 of this thesis. The FSA-PLANNER algorithm in Chapter 5 was first presented in [Hu and Levesque, 2009], along with its comparison with KPLANNER and a discussion of the heuristic search extension to the algorithm. A variant of the generalized definition of planning with loops in Chapter 6 was published in [Hu and De Giacomo, 2011a], and its associated generic solver in [Hu and De Giacomo, 2011b].
Chapter 2

Related Work

In this chapter, I will review the existing approaches to planning with loops and other related work. The survey starts from classical planning, and investigates its extensions and generalizations that finally lead to planning with loops. On the latter, I will first discuss the existing approaches to generating them, categorized into deductive and non-deductive methods. Then, theoretical results on reasoning about plan correctness and other related work will be reviewed.

2.1 From Classical Planning to Planning with Loops

In artificial intelligence, automated planning usually refers to classical planning, which involves finding a sequence of legal actions whose execution will change the world from the specified initial state to one that satisfies some goal condition [Russell and Norvig, 2003].

A classical model for planning is STRIPS [Fikes and Nilsson, 1971], where the problem is described by a set of conditions that characterize what is true in a state, a set of action operators that transforms a state into another, an initial state, and a goal condition. States and conditions are represented by a list of atomic formulas that hold in them. An action operator consists of a precondition that must be satisfied before the action can be executed, an add list of facts that will become true after its execution and a
Chapter 2. Related Work

delete list of facts that will cease to be true. A plan for a STRIPS problem is a sequence of ground actions, which is guaranteed to be executable from the initial state, and the goal condition is satisfied after the execution.

The STRIPS model of planning was extended by Pednault in his ADL to handle negation, disjunctive and quantified conditions, conditional effects, etc [Pednault, 1989]. With the introduction of the International Planning Competition in 1998, McDermott et al. proposed PDDL [Ghallab et al., 1998], a standard language for describing planning domains and problems that includes STRIPS and ADL as fragments. Largely due to this bi-annual competition, a number of efficient classical planners, most of which are based on heuristic forward search, have been developed [Bonet and Geffner, 2001b, Hoffmann and Nebel, 2001, Helmert, 2006, Richter and Westphal, 2010]. All the work in this branch shares one thing in common: the solution to the planning problem is a sequence of actions, due to the determinism of actions and states. We sometimes call this type of problem sequential planning.

Another type of planning where the resulting plan is a sequence of actions is conformant planning [Smith and Weld, 1998, Bonet and Geffner, 2000], where both the states and the actions may be nondeterministic. The task is to find a sequential plan, whose execution guarantees to achieve the goal, no matter what the true state and action effect turn out to be within the nondeterminism. In conformant planning, the world state is assumed to be unobservable, i.e., there is no way for either the planner or the plan executor to rule out any possible world within the nondeterminism.

If one assumes that the world is observable, then the plan needs no longer to be a sequence, but instead, may contain branches based on the conditions at run time in the real world. This type of problem is called conditional planning (or contingent planning) [Bertoli et al., 2001, Petrick and Bacchus, 2004, Hoffmann, 2005]. The resulting tree-like plans are able to differentiate among finitely many cases and guarantee that the goal is achieved in all those cases. For example, the logistics problem with a fixed number of
objects to deliver and the striped tower problem with a known number of blocks discussed in Chapter 1 can both be represented and solved as a conditional planning problem.

Plans with loops are required, when an unknown and unbounded property exists in a planning problem and thus infinitely many cases need to be handled in a single plan. Generating such plans will be the subject of the next section.

2.2 Generating Plans with Loops

In this section, I will survey existing work on generating plans with loops, which falls roughly into two categories, namely, deductive and non-deductive (or inductive) approaches.

2.2.1 Deductive Approaches

Deductive approaches to planning typically generate a plan as a by-product of proving a mathematical theorem. A short survey on deductive planning is conducted by Biundo in [Biundo, 1994]. Most work on deductive planning does not attempt to generate loopy or recursive plans, but here are a few exceptions.

In an early work, Manna and Waldinger proposed a tableau-based sequent calculus to deductively synthesize applicative programs [Manna and Waldinger, 1980]. Domain axioms and the program specification are represented by sequents, along with a mathematical induction rule. Each sequent is a three-column table representing assertions, goals and outputs. The associated deduction rules preserve correctness and can be used to obtain new sequents. The deduction terminates whenever the assertion FALSE or the goal TRUE is derived. At this point, the content in the “outputs” column of this sequent is a program that satisfies the problem specification. Manna and Waldinger later extended this tableau-based approach with a variant of the situation calculus (explained in Chapter 3) for representing actions dynamics, so as to synthesize recursive plans in planning.
domains [Manna and Waldinger, 1987]. The resulting planner relies on the sequents to match a well-founded induction rule, and the planning process involving theorem proving is typically slow.

With a similar goal, Stephan and Biundo proposed a deduction-based refinement planning approach based on a temporal planning logic [Stephan and Biundo, 1996]. Their idea of refinement planning is to start from a non-constructive problem specification, and gradually refine it to generate an executable plan. Since representations of specifications and plans are on the same linguistic level in temporal planning logic, the object of the refinement is a mixed representation of the two, where some part of the specification is replaced by executable plan segments in each of the refinement steps. Due to the soundness of the refinement rule, the plan that they finally derive is guaranteed to be provably correct. Unlike Manna and Waldinger’s approach [Manna and Waldinger, 1987], loops are not introduced in the refinement steps, but instead prefabricated in the initial abstract problem specification. As a result, this approach actually refines human-designed abstract plans that contain loops initially, instead of creating them from a declarative specification automatically.

In both approaches above, planning is an interactive process, where a human is needed to provide either a loop invariant or a high level plan. Indeed, no complete, fully automatic procedure exists for deductive iterative planning, since plans with recursion are Turing complete. However, if we are only after an incomplete algorithm which can solve interesting special cases, such deductive approach may still work, armed with the right heuristics.

Along this line, Magnusson and Doherty proposed a deductive planning framework for maintenance-goal problems in temporal action logic [Magnusson and Doherty, 2008]. In order to automatically obtain the loop invariants, they incorporated a regularity heuristic and a synchronization heuristic to help the deductive reasoner find useful invariants. An induction rule is then used to automatically form a plan with loops, after the induction
hypotheses as well as the base cases are identified. Magnusson and Doherty proved that both the heuristics and the induction rule are sound, \textit{i.e.}, whenever their algorithm outputs a plan, it is guaranteed to be correct. They also showed that their planner solves a practical surveillance problem based on this approach.

### 2.2.2 Non-deductive Approaches

Non-deductive approaches typically do not rely on axioms for an unbounded property, but instead generate plans with loops from example problems or plans. So they sidestep the reasoning about the induction hypothesis and loop termination. As a result, non-deductive approaches are typically more efficient than their deductive counterparts, but meanwhile offer a weaker guarantee on the plan’s correctness in the general setting.

#### Generate and Test

An early non-deductive planner is Kplanner, which automatically identifies plans with loops for planning problems with a “planning parameter,” \textit{i.e.}, an unknown and unbounded numerical property in the planning domain [Levesque, 2005].

To solve this type of problem, Kplanner first generates a conditional plan that works for a small parameter $N_1$, wind it to form loops, and then tests whether the loopy plan also works for a larger parameter $N_2$. If so, the plan is returned as the output; otherwise, Kplanner returns to the generation phase and another candidate plan is enumerated.

As we can see, the output plan is not guaranteed to work for all possible values of the planning parameter, but instead only for $N_1$ and $N_2$. However, for many practical problems, the resulting plan is indeed a valid solution that works for all values. I will review Kplanner in more detail later in the thesis, including its problem and plan representation in Chapter 3 and its planning algorithm in Chapter 5.
Identifying Regularity in Partial-Order Plans

Winner and Veloso developed an algorithm for finding “domain specific planners” (dsPlanners) for a class of ADL problems [Winner and Veloso, 2003, Winner and Veloso, 2007].

The input to their first algorithm, DISTILL [Winner and Veloso, 2003], is the domain specification, along with solutions to some example problems, which are processed sequentially to revise the current dsPlanner starting from an empty program. The output of DISTILL is a compact conditional plan that solves all the examples and likely other similar problems.

To accommodate an example solution, the plan is first parametrized to best match the current dsPlanner, and then converted into a new dsPlanner by introducing if statements for selecting objects for the parameters. The conditions for the if statement are the initial, current and goal state terms that are relevant to the plan, which can be obtained using the minimal annotated consistent partial ordering [Winner and Veloso, 2002] of the observed plan. This partial ordering information is also stored for later merging the existing dsPlanner with the newly obtained one.

To merge the two dsPlanners, the DISTILL algorithm searches through each of the if statements in the current program, and tries to find a matching in the new dsPlanner. If such a matching is found, then the two if statements are combined; otherwise, a new if statement is appended to the end of the current dsPlanner.

DISTILL, in its simplest form, cannot introduce loops to the resulting dsPlanners. To overcome this drawback, Winner and Veloso later proposed a variant of the algorithm called LoopDISTILL, which is able to automatically identify parallel and serial loops [Winner and Veloso, 2007].

The basic idea is similar to that of DISTILL. Here, instead of finding a dsPlanner for the whole parametrized plan, it identifies the largest matching subplan, and converts the repeating occurrences of these subplans into a loop. This procedure is repeated greedily, until no further loops can be formed. Unlike in the DISTILL algorithm, no procedure for
merging dsPlanners with loops is given in the paper, so LoopDistILL essentially only uses one single plan for generalization, and it is unclear how multiple example plans can be taken into account.

Winner and Veloso claim that LoopDistILL can synthesize dsPlanners with complex but non-nested loops.

**Using Role-Based Object Abstraction**

With a similar goal to find loopy plans for a class of STRIPS-like problems, Srivastava et al. proposed to use state aggregation to group objects in the same role into equivalence classes, and obtain an abstract state representation [Srivastava et al., 2007].

In their formalism, a role is defined as a conjunction of literals consisting of every abstraction (unary) predicate or its negation. As a result, a domain with \( N \) unary predicates can have at most \( 2^N \) roles and thus at most \( 2^N \) abstract objects, no matter how many actual objects exist in the domain.

To obtain a loopy plan, their algorithm only needs one concrete plan containing sufficient unrollings of some loops. From this plan, all the actual objects are replaced by their corresponding abstract objects. After this replacement, the repeating pattern of the resulting abstract actions becomes obvious, so loops can be obtained by folding the repeating sections in the abstract plan [Srivastava et al., 2008].

Srivastava et al. require that independent actions within each repetition must happen in the same order so that loops can be detected. Like Winner and Veloso’s approach, their algorithm only detects non-nested loops. To remedy this, an extension of the algorithm was proposed that takes into account multiple example plans, which may lead to simple loops with shortcuts, a special type of nested loops [Srivastava et al., 2010b].
Compilation Approaches

Bonet et al. recently proposed a technique to synthesize finite-state controllers for a propositional planning problem, so that the controller not only solves the given problem, but usually also generalizes to similar problems [Bonet et al., 2009]. This is achieved by adding the dynamics of finite-state controllers to the problem specification, and compiling the resulting problem into a conformant planning problem. This new problem has the property that it has a conformant plan if and only if the finite-state controller encoded in the plan is a correct solution to the original planning problem. Due to this property, a plan with loops can be extracted from the conformant plan for the compiled problem. Although the resulting controllers are only guaranteed to be correct for the example problems from which they are generated, Bonet et al. showed that in many cases, they indeed generalize to other similar problems.

For the same type of problems, Pralet et al. proposed a constraint based approach, which compiles the dynamics of the planning problem into a set of constraints, and uses a simulate and branch algorithm to update and validate the set of constraints, until a controller is found [Pralet et al., 2010]. They also show that the resulting planner is more general than that of Bonet et al., since action precondition and safety-oriented goals, a restricted type of temporally-extended goals, are also allowed.

Explanation-Based Generalization

Shavlik’s BAGGER2 views the problem of generating recursive and iterative concepts as an explanation-based learning problem extended with the ability of “generalizing to $N$” [Shavlik, 1990], and the system applies naturally in planning domains such as blocks world. Based on a similar idea, Schmid and Wysotzki learn iterative macro operators for a planning domain [Schmid and Wysotzki, 2000] by inductive program synthesis [Kitzelmann and Schmid, 2006]. Unlike BAGGER2, where the generalization is done over the proof tree of the background theory, they assume basic data type structures (natu-
eral numbers, lists, sets, etc), explore problems of small complexity, and generate finite programs from the universal plans of these problems, whose syntactical structure is then explored to generate loops.

### 2.3 Reasoning about Correctness of Plans with Loops

While the deductive approaches above proves plan correctness in general in the planning process itself, plans generated by the non-deductive approaches come with a much weaker guarantee, usually provably correct only for the examples used to generate them. It is thus not obvious how well the plans generalize to other instances of the planning problem. As a result, reasoning about correctness is a separate but important issue for those planners. Below, I will show a few theoretical results in the literature on this topic.

**“Simple” Planning Problems**

For Kplanner, Levesque identified a class of “simple” problems for which general correctness can be guaranteed by the planner if a plan works for all planning parameters up to a finite bound. With his theorem (Theorem 1 on Page 6 of [Levesque, 2005]), Kplanner is able to generate provably correct plans for simple problems. However, the definition of “simple” relies on three properties on plan execution, and not the planning problem itself, so given a problem specification for Kplanner, it is not obvious whether it is a simple problem or not. This is one of the major motivations for the work in Chapter 4 of this thesis, where we obtain several classes of planning problems, whose verifiability can be determined solely by the syntax of the problem specification.

**Finite Model Checking**

Lin proposed the concept of finitely-verifiable logical theories [Lin, 2007]. Intuitively, a finitely-verifiable class of sentences has the property that whether or not a sentence in
the class is a theorem can be checked with respect to a finite set of models of the theory.

This idea, when applied to the action domain, can be used to prove goal achievability in a set of initial states [Lin, 2008]. In particular, to see if a plan with loops solves all the problems in a finitely-verifiable planning domain, one can use Lin’s result, and identify the finite set of models that is sufficient for the judgement. Then, the correctness of the plan can be concluded by model checking. However, his original result only works for plans with a single global loop, which is another motivation of our work in Chapter 4 where no special form on the plans will be assumed.

Reducing to Abacus Programs

Abacus programs are register machines whose registers can be incremented and decremented by 1 and tested against 0. Srivastava et al. showed that if the loops in an abacus program satisfy certain restrictions, then a precondition can be inferred so that the abacus program, if started in a state that satisfies the condition, is guaranteed to terminate [Srivastava et al., 2010a]. Based on this result, they proved that a special class of STRIPS planning problems called “extended-LL domains” can be reduced to abacus programs, so if their plan satisfies the loop restrictions, then its correctness condition can be effectively generated. This independent result is closely related to our finite verifiability results in Chapter 4. One of the main differences, however, is that they require the plans to have a certain shape for their results to apply, whereas we study a property of the planning problem, independent and regardless of the actual shape of the plan.

Verification of Program Correctness

Another related area to the verification of plans with loops is work on program correctness in programming languages. This includes the model checking-based approaches [Clarke et al., 1999] and logical deduction on a theory of computer programs [Hoare, 1969, Hehner, 1993]. However, most work there deals with a predefined programming language,
usually involving assignments and arithmetic evaluations, whereas planning usually requires much richer dynamics. As a result, although research in verification of program correctness sheds light on planning with loops, it is not obvious how to directly apply existing results there for planning.

### 2.4 Other Related Work

Apart from the work listed so far, the following areas are also related to the theme of this thesis.

**Program Synthesis**

As mentioned above, Manna and Waldinger obtained a deductive planner from a sequent calculus for program synthesis. In general, program synthesis is the task of automatic derivation of a program to meet a given formal specification of its behavior [Manna and Waldinger, 1992]. Depending on their functionality, the programs are categorized as applicative or imperative. An applicative (or functional) program calculates an output based on the given input, producing no side effect during the computation except for the necessary modification of internal data structure. Imperative programs, on the other hand, may alter the external data structure or to produce other side effects, apart from possibly returning an output of some sort. Planning with loops can be considered as an instance of imperative program synthesis. So in principle, if we have an efficient algorithm for program synthesis, the corresponding planning problem is solved. Unfortunately, there is still a long way to go in that field before this can be realized. In fact, as we shall show in Chapter 5, a number of simple programming tasks, *e.g.*, computing the factorial of a number, sorting an integer list, can be encoded as a planning with loops problem, and our planners are able to solve practically. This indicates that the results in this thesis may be useful in the future research of program synthesis.
Grammar Induction

It is the incomplete knowledge about the world states and nondeterminism in the actions that forces a general solution to the planning problem to contain branches and loops. For a given world and a fixed combination of choices, however, a sequence of actions will suffice to achieve the goal. As a result, a solution to the planning problem can also be characterized by the set of all possible sequential plans that are needed to handle all the cases and that can be generated by a plan with loops. If we consider the set of all actions as the alphabet of a language and each sequential plan as a string in the language, planning is cast into a grammar induction problem (Section 8.7 of [Duda et al., 2001]). From this perspective, the task is simply to find the underlying grammar that can generate the observed strings from a language.

More specifically, there are three types of inputs to the learner according to [Gold, 1967]: positive examples only, positive and negative examples, or oracle feedback. Our planning falls into the category “oracle feedback,” since the dynamic environment serves as an oracle that tells the planner whether a sequence of actions is legal and whether it achieves the goal.

Repeated-Attempts Problems

Another related type of problems involves nondeterministic domains, where the outcomes of actions are independent, so repeated attempts may be needed for a desired outcome of the action to occur.

For example, when picking up a block on the table, two possible outcomes may happen. In one case, the block is successfully picked up, and in the other, it remains on the table [Liem Ngo, 1995]. So, if one wants to guarantee the block in hand, the pick-up action will need to be performed repeatedly. A similar problem is in the omelet domain [Bonet and Geffner, 2001a] where a sequence of eggs exists and each egg may be good or bad independent of the others.
Cimatti et al. proposed three criteria for plan validity in such nondeterministic domains [Cimatti et al., 2003]. A weak plan is one that may achieve the goal, but not guaranteed to do so; a strong plan is one that always achieves the goal, no matter what output each action generates; a strong cyclic plan is one that guarantees to achieve the goal under a “fairness assumption” and possibly after repeated trial and error. According to this definition, all repeated-attempts problems that need loop plans can at best have strong cyclic solutions, and not strong solutions. This is due to the outcome-independence assumption that they insist on the action definition. The planning problems that we focus on in this thesis, in contrast, need to have “strong” solutions. For example, in the logistics and the striped tower problems in Chapter 1, the action outcomes are determined by the true world state and are not independent across the action’s happenings, yet we need to achieve the goal with certainty for each case.

Similarly, decision-theoretic planning [C. Boutilier and Hanks, 1999], which are essentially repeated-attempts problems with possibilities associated to each action’s outcome and where the goal is to maximize some utility measure, is related to but different from our planning pursuit.
Chapter 3

Formal Preliminaries

In this Chapter, I will present the formal framework for representing and reasoning about planning problems and plans that require loops, on which the rest of this thesis is based.

The first half of this chapter is a review of the situation calculus [McCarthy, 1963] using Reiter’s formalization with basic action theories [Reiter, 2001] and Scherl and Levesque’s treatment of knowledge and sensing actions [Scherl and Levesque, 2003].

In the second half, I will discuss the representation of planning problems and plans that require loops. This includes a review of Levesque’s robot programs [Levesque, 1996], and a new alternative definition using finite-state automaton-like plans (FSA plans). Finally, I will show that the new plan representation is more general than robot programs.

3.1 Situation Calculus

The situation calculus is a first-order, multi-sorted logical language with equality and limited second-order features, for representing and reasoning about dynamical environments. It was introduced by John McCarthy [McCarthy, 1963] and further formalized later by Raymond Reiter [Reiter, 2001] to include a solution to the frame problem [McCarthy and Hayes, 1969]. In this thesis, I will mainly follow Reiter’s presentation, extended with the treatment of sensing actions introduced by [Scherl and Levesque, 2003].
3.1.1 The Language

Objects in the situation calculus are of three disjoint sorts: situation for situations, action for actions, and object for everything else.

The alphabet of the language contains the following elements.

- A countably infinite set of variables for each sort. Following Reiter’s naming convention, I typically use $s$ and $a$ (possibly with subscripts or superscripts) to denote variables of sort situation and action, respectively, and other small letters for variables of sort object.

- Two function symbols of sort situation, namely, the constant $S_0$ for the initial situation, and a binary function $do : action \times situation \rightarrow situation$, where $do(a, s)$ is intended to denote the situation obtained after performing action $a$ in situation $s$. The notation $do([a_1, \cdots, a_n], s)$ is used as an abbreviation for $do(a_n, do(\cdots, do(a_1, s)))$.

- For each $n \geq 0$, a finite or countably infinite set of function symbols of sorts $(action \cup object)^n \rightarrow (action \cup object)$. These are the situation-independent functions called rigid functions.

- For each $n \geq 0$, a finite or countably infinite set of function symbols of sorts $(action \cup object)^n \times situation \rightarrow (action \cup object)$. These are the situation-dependent functions called functional fluents or fluent functions.

- For each $n \geq 0$, a finite or countably infinite set of predicate symbols of sorts $(action \cup object)^n$. These are the situation-independent predicates.

- For each $n \geq 0$, a finite or countably infinite set of predicate symbols of sorts $(action \cup object)^n \times situation$. These are the situation-dependent predicates called relational fluents or fluent predicates.

- A special binary predicate symbol $\sqsubset : situation \times situation$ defining the ordering of situations. $s \sqsubset s'$ is intended to mean that $s$ is a situation prior to $s'$, i.e., $s' = \ldots \sqsubset s \sqsubset s'$.
do(\alpha, s) \text{ for some non-empty action sequence } \alpha. \ s \sqsubseteq s' \text{ is used as an abbreviation for } s \sqsubseteq s' \lor s = s'.

- A special binary predicate symbol \textit{Poss : action} \times \textit{situation} defining the executability of actions in situations. Poss(a, s) is intended to mean that action \( a \) is legally executable in situation \( s \).

- A special binary function symbol \textit{sr : action} \times \textit{situation} \to \textit{object} defining the sensing results of actions in situations. \( \text{sr(a, s)} = r \) is intended to mean that action \( a \) will return sensing result \( r \) in situation \( s \).

- The standard logical symbols \( \neg \), \( \land \) and \( \exists \) (with other logical symbols like \( \lor \) and \( \forall \) being abbreviations), and equality \( = \). As a notational convention, the leading universal quantifiers may be omitted in logical sentences, \textit{i.e.}, all free variables in a sentence are assumed to be universally quantified.

A formula \( \phi \) is said to be \textit{uniform in} \( s \) if it does not mention \( \textit{Poss, sr, } \sqsubseteq \text{ or any situation term other than } s \). A fluent formula \( \phi \) with all situation arguments eliminated is called a \textit{situation-suppressed} formula, and \( \phi[s] \) denotes the uniform formula with all situation arguments restored with term \( s \).

Sometimes, we need to replace a term in a formula with another term. For this purpose, \( W|^{t_1}_{t_2} \) is used to denote the formula obtained by simultaneously replacing all occurrences of term \( t_1 \) in formula \( W \) by term \( t_2 \).

\subsection{3.1.2 Basic Action Theories}

The language of the situation calculus introduced above is used to represent dynamic environments. This is done by writing a set of axioms in the language, so that the properties of the environment are captured. The axiomatization proves non-trivial, since several challenging problems need to be addressed, including the qualification problem, the ramification problem, and the frame problem [McCarthy and Hayes, 1969].
In this thesis, I shall use a well-established type of situation calculus axiomatization in the so-called basic action theories (BATs) that provides a solution of the problems for a wide range of applications [Reiter, 2001]. In order to handle sensing, we use also include a set of sensing result axioms introduced by [Scherl and Levesque, 2003].

In this thesis, we define the basic action theory as a set of axioms of the form

\[ D = FA \cup D_{pre} \cup D_{ssa} \cup D_{sr} \cup D_{una} \cup D_0, \]

where

- \( FA \) is the following set of domain-independent axioms defining the legal situations:

\[
\begin{align*}
    & do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \land s_1 = s_2 & (3.1) \\
    & (\forall P).P(S_0) \land (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s) P(s) & (3.2) \\
    & \neg(s \sqsubset S_0) & (3.3) \\
    & s \sqsubset do(a, s') \supset s \sqsubseteq s' & (3.4)
\end{align*}
\]

These axioms force the structure of the situations in any interpretation of the basic action theory to be a tree with a well-founded ordering according to the \( do \) function. The foundation axioms essentially ensure that the space of situations is the smallest set containing the initial situation and closed under \( do \). Note that the second-order axiom (3.2) is needed to capture the transitive closure property.

- \( D_{pre} \) is a set of action precondition axioms, one for each action symbol \( A \) of the form

\[
Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s).
\]

For example, in the logistics world introduced in Chapter 1, the following says that
load is possible in s iff the truck is at the source location and not loaded:

\[ \text{Poss}(\text{load}, s) \equiv \text{loc}(s) = \text{src}(s) \land \neg\text{Loaded}(s). \]

• \( \mathcal{D}_{ssa} \) is a set of successor state axioms (SSAs), one for each fluent predicate symbol \( F \) of the form

\[ F(\bar{x}, \text{do}(a, s)) \equiv \Phi_F(\bar{x}, a, s), \]

and one for each fluent function symbol \( f \) of the form

\[ f(\bar{x}, \text{do}(a, s)) = y \equiv \Phi_f(\bar{x}, a, y, s). \]

For example, the following says that the truck is loaded iff a load action is just performed, or it was loaded already and no unload action is performed:

\[ \text{Loaded}(\text{do}(a, s)) \equiv a = \text{load} \vee \text{Loaded}(s) \land a \neq \text{unload}, \]

and the following says that the location of the truck after performing \( a \) in \( s \) is either the destination of \( \text{move} \) if \( a \) is a move action, or its old location if \( a \) is not:

\[ \text{loc}(\text{do}(a, s)) = y \equiv a = \text{move}(y) \vee \text{loc}(s) = y \land \neg \exists z. (a = \text{move}(z)). \]

• \( \mathcal{D}_{sr} \) is a set of sensing result axioms, one for each action, of the form

\[ \text{sr}(A(\bar{x}), s) = r \equiv \Theta_A(\bar{x}, r, s). \]

For example, the following says that \( \text{check} \) returns yes if the number of objects left
is 0, and no otherwise:

\[ \text{sr}(\text{check}, s) = r \equiv r = \text{yes} \land \text{left}(s) = 0 \lor \]
\[ r = \text{no} \land \text{left}(s) \neq 0. \]

- \( D_{\text{una}} \) is a set of unique names axioms for actions of the form

\[ A(\vec{x}) \neq A'(\vec{y}) \]

for distinct action symbols \( A \) and \( A' \), and

\[ A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}. \]

- \( D_0 \) is the initial knowledge base stating facts about \( S_0 \).

### 3.1.3 The Projection Problem

Given a situation-calculus axiomatization of a dynamic world, an important reasoning task is the so-called projection problem, i.e., determining whether a certain condition holds after a sequence of actions is performed from the initial situation. Formally, given a basic action theory \( D \) and a sentence \( \phi \) in the situation calculus that may contain situation terms other than \( S_0 \), the projection problem is to decide whether \( D \models \phi \).

For example, in automated planning, a typical query is whether the goal condition is achieved in a particular situation after a sequence of actions is executed. In the transportation world, such a query may be

\[ D \models \text{loc}(\text{do}([\text{load}, \text{move(office)}, \text{unload}], S_0)) = \text{office}. \]

There are two ways for solving the projection problem in the situation calculus,
namely, by regression and by progression.

Regression

Intuitively, in order to prove an entailment of a formula involving future situations by regression, one obtains an equivalent formula that only mentions the initial situation $S_0$. This way, the projection problem is reduced to an entailment problem in standard first-order logic.

Regression can be used for a wide range of regressible formulas. A situation calculus formula $W$ is regressible [Reiter, 2001] iff

1. each term of sort situation in $W$ has the syntactic form $do([a_1, \ldots, a_n], S_0)$, where $n \geq 0$ and all $a_1, \ldots, a_n$ are terms of sort action;

2. for each atom $Poss(a, s)$ and function $sr(a, s)$ in $W$, $a$ has the form $A(t_1, \ldots, t_n)$ for some $n$-ary action function symbol $A$;

3. $W$ does not quantify over situations;

4. $W$ does not mention $\Box$ or $s_1 = s_2$ where the terms $s_1$ and $s_2$ are of sort situation.

On regressible formulas, the regression operator $R$ is defined recursively as [Reiter, 2001]

\begin{itemize}
  \item $R[W] = R[\Pi_A(\vec{t}, \sigma)]$ if $W$ is the atom $Poss(A(\vec{t}), \sigma)$ and the precondition axiom for the action symbol $A$ is $Poss(A(\vec{x}, s)) \equiv \Pi_A(\vec{x}, s)$;
  \item $R[W] = R[\Theta_A(\vec{t}, \rho, \sigma)]$ if $W$ is the formula $sr(A(\vec{t}), \sigma) = \rho$ and the sensing result axiom for the action symbol $A$ is $sr(A(\vec{x}, s)) = r \equiv \Theta_A(\vec{x}, r, s)$;
  \item $R[W] = W$ if $W$ is a situation-independent atom, or mentions only fluents of the form $F(\vec{t}, S_0)$;
  \item $R[W] = R[\Phi_F(\vec{t}, \alpha, \sigma)]$ if $W$ is the atom $F(\vec{t}, do(\alpha, \sigma))$ and the successor state axiom for the fluent predicate symbol $F$ is $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$;
\end{itemize}
• $\mathcal{R}[W] = \mathcal{R}[(\exists y). \phi_f(\vec{t}, y, \alpha, \sigma) \land W[y^f(\vec{t}, do(\alpha, \sigma))]]$ if $W$ mentions the term $f(\vec{t}, do(\alpha, \sigma))$

where $f$ is a fluent function symbol whose successor state axiom is $f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s)$;

• $\mathcal{R}[\neg W] = \neg \mathcal{R}[W]$;

• $\mathcal{R}[W_1 \land W_2] = \mathcal{R}[W_1] \land \mathcal{R}[W_2]$;

• $\mathcal{R}[(\exists v). W] = (\exists v). \mathcal{R}[W]$.

Given any regressible formula $W$, the regressed formula $\mathcal{R}[W]$ will be a formula whose only situation terms are $S_0$. The following regression theorem ensures that the projection problem can be reduced to proving entailment in first-order logic.

**Theorem 1** (The Regression Theorem[Reiter, 2001]). For any basic action theory $\mathcal{D} = \mathcal{FA} \cup \mathcal{D}_{pre} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{sr} \cup \mathcal{D}_{una} \cup \mathcal{D}_0$ and regressible formula $W$,

$$
\mathcal{D} \models W \iff \mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[W].
$$

**Progression**

Instead of regressing the query formula, an alternative method is to progress the initial database of the basic action theory to the situation in which the query formula is uniform. Since the progressed database axioms and the query formula are uniform in the same situation, first-order theorem proving can again be used to solve the projection problem.

A set of sentences $\mathcal{D}_\alpha$ is said to be the progression of $\mathcal{D}_0$ through an action sequence $\alpha$ to $S_\alpha = do(\alpha, S_0)$ with respect to $\mathcal{D}$ iff [Reiter, 2001]

• $\mathcal{D}_\alpha$ is uniform in $S_\alpha$;

• $\mathcal{D} \models (\mathcal{D} \setminus \mathcal{D}_0) \cup \mathcal{D}_\alpha$;
• for an observer standing in situation $D_\alpha$ and looking into the future, she cannot distinguish between a model for the original basic action theory $D$ and one for the new theory $(D \setminus D_0) \cup D_\alpha$.

Lin and Reiter showed that progression is not always first-order definable, and identified several special cases in which it is [Lin and Reiter, 1997].

### 3.2 Reasoning about Plans with Loops

As introduced in Chapter 1, plans with loops are interesting due to their generality and applicability to multiple instances of a planning problem. To rigorously study this type of planning problems, a formal characterization is needed.

By appealing to the situation calculus framework introduced earlier, it is relatively straightforward to formalize such problems with a basic action theory. All we need to know is the action dynamics of the planning problem and the goal to be achieved. In this thesis, I mainly focus on reachability goals, where the task is to fulfil a certain condition in the final situation obtained by executing the plan, although in Chapter 6, I will generalize to temporally-extended and long-running goals. The following is a precise definition of the planning problem we consider for reachability goals.

**Definition 1** (The Planning Problem). A planning problem is a pair $(D, G)$, where $D$ is a basic action theory, and $G$ is a situation-suppressed formula in the situation calculus.

As a running example for the discussion in this chapter, let us consider the tree chopping problem introduced in [Levesque, 2005]:

The goal is to chop down a tree, and put away the axe. The number of chops needed to fell the tree is unknown, but a look action checks whether the tree is up or down. Intuitively, a solution involves first look and then chop whenever up is sensed. This repeats until down is sensed, in which case we store the axe, and are done.
This example can be readily formalized as $\langle D_{tc}, G_{tc} \rangle$ where $G_{tc}$ is the situation-suppressed formula $\text{axe} = \text{stored} \land \text{chops}_{\text{needed}} = 0$ and $D_{tc}$ is the basic action theory containing the following elements:

- **Precondition Axioms:**

  \[
  \text{Poss} (\text{look}, s) \equiv \text{TRUE} \\
  \text{Poss} (\text{chop}, s) \equiv \text{chops}_{\text{needed}}(s) \neq 0 \land \text{axe}(s) = \text{out} \\
  \text{Poss} (\text{store}, s) \equiv \text{axe}(s) = \text{out}
  \]

- **Successor State Axioms:**

  \[
  \text{axe}(\text{do}(a, s)) = x \equiv \begin{cases} 
  x = \text{stored} \land a = \text{store} \lor \\
  x = \text{axe}(s) \land a \neq \text{store} 
  \end{cases} \\
  \text{chops}_{\text{needed}}(\text{do}(a, s)) = x \equiv \begin{cases} 
  x = \text{chops}_{\text{needed}}(s) - 1 \land a = \text{chop} \lor \\
  x = \text{chops}_{\text{needed}}(s) \land a \neq \text{chop} 
  \end{cases}
  \]

- **Sensing Result Axioms:**

  \[
  \text{sr}(\text{look}, s) = r \equiv \begin{cases} 
  r = \text{up} \land \text{chops}_{\text{needed}} \neq 0 \lor \\
  r = \text{down} \land \text{chops}_{\text{needed}} = 0 
  \end{cases} \\
  \text{sr}(\text{chop}, s) = r \equiv r = \text{ok} \\
  \text{sr}(\text{store}, s) = r \equiv r = \text{ok}
  \]

- **Initial Situation Axiom:**

  \[
  \text{axe}(S_0) = \text{out} \land \text{chops}_{\text{needed}}(S_0) \geq 0
  \]
Given the problem definition, it remains to define what is a plan with loops and how it fits into the logical framework. In the rest of this section, I shall study two alternative representations of such plans, namely, the existing robot programs and the new FSA plans. In either case, characterizations of both the syntactic form and the semantics of the plans are elaborated. Finally, I conclude the section with a comparison of the generality between the two representations.

3.2.1 Robot Programs

Levesque defined a robot program language, whose syntax and meaning is as follows:

**Definition 2** (Robot program [Levesque, 1996, Levesque, 2005]). A robot program and its execution is defined inductively as

1. **nil** is a robot program executed by doing nothing;

2. for any primitive action A and robot program P, **seq**(A, P) is a robot program executed by first performing A, ignoring any sensing result, and then executing P;

3. for any primitive action A with possible sensing result R₁ to Rₖ, and for any robot programs P₁ to Pₖ, **case**(A, [if(R₁, P₁), · · ·, if(Rₖ, Pₖ)]) is a robot program executed by first performing A, and then on obtaining the sensing result Rᵢ, continuing by executing Pᵢ;

4. if P and Q are robot programs, and B is the result of replacing in P some of the occurrences of **nil** by **exit** and the rest by **next**, then **loop**(B, Q) is a robot program executed by repeatedly executing the body B until the execution terminates with **exit** (rather than **next**), and then going on by executing the continuation Q.

According to this definition, intuitively, the following robot program, \( p_{tc} \), is a solution to the tree-chopping problem introduced in Chapter 1:
\textbf{loop}(% \textbf{case} (look, [if (down, exit), if (up, seq (chop, next))]), seq (store, nil))

For notational convenience, robot programs are often written in a more readable, indented format. For example, $p_{te}$ can be presented as:

\begin{verbatim}
LOOP
    CASE look OF
    - down: EXIT
    - up: chop;
    NEXT
ENDC
ENDL;
store
\end{verbatim}

In order to capture the semantics of robot programs, an $Rdo(p, s_1, s_2)$ relation is defined to mean that robot program $p$, when executed in situation $s_1$, will legally terminate in situation $s_2$. Formally, $Rdo$ is the abbreviate of the following second-order formula [Levesque, 1996]

$$Rdo(p, s_1, s_2) \stackrel{\text{def}}{=} (\forall P). [\cdots \supset P(p, s_1, s_2, 1)],$$

where the ellipsis is the conjunction of the universal closure of the following:

1. Termination, normal case:
   $$P(\text{nil}, s, s, 1),$$
   $$P(\text{next}, s, s, 1);$$

2. Termination, loop body:
   $$P(\text{exit}, s, s, 0);$$

3. Ordinary actions:
   $$\text{Poss}(a, s) \land P(p', do(a, s), s', x) \supset P(\text{seq}(a, p'), s, s', x);$$
4. Sensing actions:

\[ \text{Poss}(a, s) \land \text{SR}(a, s) = r_i \land P(p_i, do(a, s), s', x) \supset P(\text{case}(a, [\text{if}(r_1, p_1), \ldots, \text{if}(r_k, p_k)]), s, s', x); \]

5. Loop, exit case:

\[ P(p', s, s'', 0) \land P(p'', s'', s', x) \supset P(\text{loop}(p', p''), s, s', x); \]

6. Loop, repeat case:

\[ P(p', s, s'', 1) \land P(\text{loop}(p', p''), s'', s', x) \supset P(\text{loop}(p', p''), s, s', x). \]

Using the \textit{Rdo} relation, we can now formalize the planning task for a given planning problem \langle D, G \rangle as finding a robot program \( p \) such that\(^1\)

\[ D \models \exists s. \text{Rdo}(p, S_0, s) \land G[s]. \quad (3.5) \]

This entailment intuitively means that for any possible initial situation \( s \), the found robot program \( p \) will terminate, and result in a situation \( s' \) in which the goal \( G \) is satisfied.

For example, let \( D_{tc} \) be the action theory of the tree-chopping domain defined above, then the following entailment can be proved.

\[ D_{tc} \models \exists s. \{ \text{Rdo}(p_{tc}, S_0, s) \land \text{tree}(s) = \text{down} \land \text{axe}(s) = \text{stored} \} \]

\(^1\)In the original definition, Levesque formalized the incomplete knowledge about the initial state explicitly using the knowledge fluent \( K \). This is a simplified definition where we assume the incomplete knowledge is implicitly specified in the initial situation axioms \( S_0 \). This is not a loss of generality, since we do not explicitly reason about knowledge in this thesis.
3.2.2 FSA Plans

The syntax of robot programs is designed in a similar fashion to common programming languages, yet there is no guarantee that the language is most compact or general. For example, the seq construct can be viewed as a special case of the case construct with one single if branch. This raises a question whether there is an alternative plan representation that has minimal assumptions on the syntax of the language. In the following, I shall present a finite-state automaton-like plan representation that follows this design principle.

**Definition 3 ([Hu and Levesque, 2009]).** An FSA plan is a tuple \( \langle Q, \gamma, \delta, Q_0, Q_F \rangle \), where

- \( Q \) is a finite set of program states;
- \( Q_0 \in Q \) is an initial program state;
- \( Q_F \in Q \) is a final program state;
- \( \gamma : Q^- \rightarrow A \) is a function, where \( Q^- = Q \setminus \{Q_F\} \) and \( A \) is the set of primitive actions;
- \( \delta : Q^- \times R \rightarrow Q \) is a function, where \( R \) is the set of sensing results, that specifies the program state to transition to for each non-final state and valid sensing result for the associated action.

The execution of an FSA plan starts from \( q = Q_0 \), and performs the action \( \gamma(q) \) associated with program state \( q \). On observing sensing result \( r \), it transitions to the new program state \( \delta(q, r) \). This repeats until \( Q_F \) is reached.

FSA plans can be visualized graphically, where every node \( q \) in the graph is a program state, labelled with its associated action \( \gamma(q) \). A directed edge labelled with \( r \) exists between \( q_1 \) and \( q_2 \) iff \( \delta(q_1, r) = q_2 \). This label can be omitted if \( r \) is the only possible sensing outcome of the action. The initial state \( Q_0 \) is denoted by an arrow pointing to it, and the final state \( Q_F \) by a double border. For example, Figure 3.1 shows an FSA plan for the tree chopping problem above.
As we can see, FSA plans are essentially Moore machines [Moore, 1956], since the next action to perform is determined solely by the current state of the machine. Alternatively, one can also define finite-state plans in the form of Mealy machines [Mealy, 1955], where the actions are determined by the transitions, and labelled on the edges. This latter representation is used in [Bonet et al., 2009] and [Pralet et al., 2010], and I will explore this direction in Chapter 6.

In order to represent FSA plans in the situation calculus, I assume that there is a sub-sort of object called program-state, with $Q_0$ and $Q_F$ being two constants of this sort, and two rigid function symbols $\gamma$ and $\delta$. I use a set of sentences $\mathcal{D}_{fsa}$ to axiomatize the plan:

**Definition 4.** $\mathcal{D}_{fsa}$ is a set of axioms consisting of

1. Domain closure axiom for program states

\[(\forall q). \{ q = Q_0 \lor q = Q_1 \lor \cdots \lor q = Q_n \lor q = Q_F \};\]
2. **Unique names axioms for program states**

\[ Q_i \neq Q_j \text{ for } i \neq j; \]

3. **Action association axioms**, one for each program state other than \( Q_F \), of the form

\[ \gamma(Q) = A \]

4. **Transition axioms** of the form \( \delta(Q,R) = Q' \)

To capture the desired semantics, a transition relation \( T^* \) is introduced, similar to \( Rdo \) for robot programs. Intuitively, \( T^*(q_1, s_1, q_2, s_2) \) means that from program state \( q_1 \) and situation \( s_1 \), the FSA plan may reach \( q_2 \) and \( s_2 \) at some point during the execution. The formal definition is given in Definition 5.

**Definition 5.** \( T^*(q_1, s_1, q_2, s_2) \) is an abbreviation for

\[(\forall T).\{\cdots \supset T(q_1, s_1, q_2, s_2)\},\]

where the ellipsis is the conjunction of the universal closure of the following:

- \( T(q, s, q, s) \)
- \( T(q, s, q'', s'') \land T(q'', s'', q', s') \supset T(q, s, q', s') \)
- \( \gamma(q) = a \land Poss(a, s) \land SR(a, s) = r \land \delta(q, r) = q' \supset T(q, s, q', do(a, s)) \)

Notice that this definition also uses second-order quantification to ensure that \( T^* \) is the least predicate satisfying the three properties above. This essentially constrains the set of tuples satisfying \( T^* \) to be the reflexive transitive closure of the one-step transitions in the FSA plan.

With this transition relation, I can now characterize the correctness of FSA plans as follows.
Definition 6 (Plan correctness). Given a planning problem \( (\mathcal{D}, G) \), a plan axiomatized by \( \mathcal{D}_{fsa} \) is correct iff

\[
\mathcal{D} \cup \mathcal{D}_{fsa} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s].
\]  

(3.6)

The definition essentially says that for an FSA plan to be correct, it must guarantee that for any model of \( \mathcal{D} \), the execution of the FSA plan will reach the final state \( Q_F \), and the goal is satisfied in the corresponding situation \( s \).

3.2.3 Comparison of the Plan Representations

To compare the expressiveness between robot programs and FSA plans, a notion of equivalence is needed. Intuitively, two plan representations are equivalent if, starting from the same situation, they always lead to the same set of final situations. This idea is formalized in the following definition.

Definition 7. A robot program \( p \) and an FSA plan axiomatized by \( \mathcal{D}_{fsa} \) are equivalent with respect to an action theory \( \mathcal{D} \) if and only if

\[
\mathcal{D} \models \forall(s_1, s_2). \ Rdo(p, s_1, s_2) \equiv [\mathcal{D}_{fsa} \supset T^*(Q_0, s_1, Q_F, s_2)]
\]

With this notion of equivalence, I first show that FSA plans are at least as expressive as robot programs.

Theorem 2. For any robot program \( p \), there exists an equivalent FSA plan.

Proof. By structural induction on robot program constructs.

1. If \( p \) is nil, then the following FSA plan is equivalent to \( p \).
2. If \( p \) is \( \text{seq}(A, P) \), then the following FSA plan constructed by creating a new initial program state labelled with \( A \) and transitioning to the initial program state of \( P \), is an FSA plan for \( p \).

![Diagram](image.png)

3. If \( p \) is \( \text{case}(A, [\text{if}(R_1, P_1), \ldots, \text{if}(R_k, P_k)]) \), then the following FSA plan is an FSA plan for \( p \), constructed by creating a new initial program state labelled with \( A \), on each sensing result \( R_i \) transitioning to the initial programs state of \( P_i \), and redirecting all edges to their respective final states to a new common final program state.

![Diagram](image2.png)

4. If \( p \) is \( \text{loop}(P, Q) \), then the following FSA plan is an FSA plan for \( p \), constructed by redirecting the edges to the final state of (the FSA plan for) \( P \). The edges
corresponding to a \textbf{next} instruction in the robot program are redirected to the initial state of $P$, and those corresponding to an \textbf{exit} instruction are redirected to the initial state of $Q$.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{robot_programs.png}
\caption{Robot programs that has no equivalent robot program.}
\end{figure}

Next, I will show that not all FSA plans have an equivalent robot program representation.

\textbf{Theorem 3.} \textit{There exists an FSA plan that has no equivalent robot program.}

\textit{Proof.} Consider the FSA plan shown in Figure 3.2(a). Suppose, for the sake of contra-
diction, that there is a smallest robot program \( P \) that is equivalent to the FSA plan (a), i.e., \( P \) is constructed by the minimum number of applications of the four basic syntactic constructs in Definition 2.

Since the first action is \( a \), \( P \) must be of the form either

\[
P' = \begin{cases} 
\text{CASE } a \text{ OF} \\
- \text{ fail: } P_1 \\
- \text{ done: nil} \\
- \text{ ok: } \cdots \\
\text{ENDC}
\end{cases}
\]

or

\[
P'' = \begin{cases} 
\text{LOOP} \\
\text{...} \\
\text{LOOP} \\
\text{CASE } a \text{ OF} \\
- \text{ fail: } P_2 \\
- \text{ ok: } P_3 \\
- \text{ done: EXIT} \\
\text{ENDC} \\
\text{ENDL;} \\
\text{EXIT} \\
\text{...} \\
\text{ENDL;} \\
\text{nil}
\end{cases}
\]

If \( P = P' \), then \( P_1 \) is a robot program that is equivalent to the FSA plan starting from program state \( a \), yet \( P_1 \) is smaller than \( P \), which contradicts the assumption that \( P \) is the smallest such robot program. As a result, \( P \neq P' \).
If, on the other hand, \( P = P'' \), then for the same reason as above, \( P_2 \) cannot be an independent robot program that starts from generating action \( a \). The only possibility is that \( P_2 = \text{NEXT} \). This means that \( P_3 \) must be a smallest robot program that is equivalent to the FSA plan in Figure 3.2(b).

Due to symmetry and the same argument as above, \( P_3 \) must be of the form

\[
P'''' = \text{LOOP} \cr \vdotswithin{\text{LOOP}} \cr \text{CASE } b \text{ OF} \cr \quad \text{- fail: NEXT} \cr \quad \text{- ok: } P_4 \cr \quad \text{- done: EXIT} \cr \text{ENDC} \cr \text{ENDL;} \cr \text{EXIT} \cr \vdotswithin{\text{ENDL:}} \cr \text{nil}
\]

Notice that \( P_4 \) can be neither EXIT nor NEXT, so it has to be an independent robot program that is equivalent to FSA plan (a), but \( P_4 \) is smaller than \( P \), contradicting the assumption that \( P \) is the smallest. As a result \( P \neq P'' \).

This proves that no robot program exists that is equivalent to the FSA plan shown in Figure 3.2(a).

Theorems 2 and 3 together show that FSA plans are strictly more general than robot programs given a fixed set of primitive actions. For this reason, I will use FSA plans as the plan representation in the rest of this thesis.
One important thing to notice is that the claim in Theorem 3 is based on the assumption that the robot program and the FSA plan act in the same, restricted action theory. If the action theory is complex enough to encode the “Turing machine actions,” then robot programs, as well as FSA plans, can be Turing complete, and thus general and expressive enough to represent arbitrary computable controller function [Lin and Levesque, 1998].

3.3 Concluding Remarks

This chapter lays the foundation for representing and reasoning about planning problems and plans that require loops.

Reiter’s formalization of the situation calculus, together with Scherl and Levesque’s addition of knowledge and sensing actions, are introduced and will be used in the rest of this thesis for the representation and reasoning of planning environments and problems.

As for the general notion of plans, the new representation of FSA plans is introduced. Their graphical representation provides an intuitive visualization of the plans, and the formal specification itself proves more general than its existing alternative, namely, the robot programs. This justifies the use of FSA plans as the plan representation in this thesis.

Given the well-defined formal framework, the next questions are how to practically reason about and generate plans with loops in the framework, as well as how the technique involved can be used in real-world applications. These will be the focus of the next three chapters, which are also the main contributions of this thesis.
Chapter 4

Finite Verifiability

For any planning problem \( \langle \mathcal{D}, G \rangle \), an important reasoning task is to decide whether a given FSA plan correctly solves the problem. As discussed in Chapter 3, assuming that the FSA plan is axiomatized by \( \mathcal{D}_{fsa} \), this task is to prove whether

\[
\mathcal{D} \cup \mathcal{D}_{fsa} \models (\exists s). T^*(Q_0, S_0, Q_F, s) \land G[s]. \tag{3.6}
\]

The entailment in (3.6) serves as a succinct formal definition that precisely captures the notion of plan correctness in the presence of loops. However, it cannot directly be used to reason about the correctness of a plan with loops for at least two reasons. First, the definition of \( T^* \) requires second-order logic, so it is far from decidable in general. Besides, the need for loops in the plan is usually caused by some unbounded properties in the problem, \( e.g. \), the unknown number of chops needed to fell a tree, the unknown number of blocks on the table, \( etc. \) As a result, there can be infinitely many models of the basic action theory, so the naive approach of direct reasoning by case will not work.

Despite these intrinsic difficulties, it is still possible to study plan correctness for manageable and useful fragments of the general problem. Along this line, one particularly interesting problem is to identify for what types of planning problems can plan correctness be practically verified.
In this chapter, I shall study a property called finite verifiability of planning problems. More specifically, given a planning problem \( \langle D, G \rangle \), I want to understand, based on the syntactic properties of the basic action theory \( D \) and goal formula \( G \), whether it is sufficient to verify a plan for only finitely many cases in order to conclude whether any FSA plan is a correct solution to the problem.

The main contribution of this chapter is two-fold. First, in Sections 4.1 and 4.2, I identify several classes of planning problems the correctness of whose FSA plans can be verified finitely. Then, in Section 4.3, I prove that intrinsically difficult problems can be reduced to finite verifiability of slight generalizations of the finitely verifiable class, so these more general classes are not (or unlikely to be) finitely verifiable in general.

### 4.1 Finite Verifiability with Decreasing Parameters

I start by analyzing one type of planning problem, where all the fluents in the basic action theory take values from a finite set, except for one single fluent, called the planning parameter, which can take any non-negative integer value, but can only have restricted tests and action effects. We call this type of problems one-dimensional, and show that FSA plans for such problems are always finitely verifiable [Hu and Levesque, 2010]. I then extend the result to problems with multiple decreasing parameters, and obtain the finite verifiability result for them too.

#### 4.1.1 One-Dimensional Planning Problems

One-dimensional planning problems involve independently processing an unbounded number of objects: delivering packages according to their shipping labels, pushing the buttons on a safe according to the digits of a combination, keeping or discarding eggs according to their smell, etc.

For example, consider the following variant of the logistics problem: There are an
unknown and unbounded number of objects at their source locations (office or home), and the goal is to move them all to their destinations with a truck. The available actions include moving the truck to a location, loading and unloading an object, finding the source and destination locations of an object, and checking whether all objects have been processed. The goal is to deliver all objects to their respective destinations.

This example can be formalized in the situation calculus framework using the following basic action theory.

- **Precondition Axioms:**

  \[
  \begin{align*}
  Poss(move(x), s) & \equiv True \\
  Poss(load, s) & \equiv \text{loc}(s) = \text{source}(\text{parcels_left}(s), s) \land \\
  \text{loaded}(s) & = False \\
  Poss(unload, s) & \equiv \text{loaded}(s) = True \\
  Poss(find\_src, s) & \equiv \text{parcels_left}(s) \neq 0 \\
  Poss(find\_dest, s) & \equiv \text{parcels_left}(s) \neq 0 \\
  Poss(check\_done, s) & \equiv True
  \end{align*}
  \]

- **Successor State Axioms:**

  \[
  \begin{align*}
  \text{loaded}(\text{do}(a, s)) = x & \equiv x = True \land a = \text{load} \lor \\
  x & = False \land a = \text{unload} \lor \\
  x & = \text{loaded}(s) \land a \neq \text{load} \land a \neq \text{unload} \\
  \text{parcels_left}(\text{do}(a, s)) = x & \equiv \\
  x & = \text{parcels_left}(s) - 1 \land a = \text{unload} \lor \\
  x & = \text{parcels_left}(s) \land a \neq \text{unload}
  \end{align*}
  \]
source\((x, do(a, s)) = y \equiv source(x, s) = y \)

dest\((x, do(a, s)) = y \equiv dest(x, s) = y \)

loc\((do(a, s)) = x \equiv \exists y. x = y \land a = move(y) \lor \\
\quad x = loc(s) \land a \neq move(y) \)

misplaced\((do(a, s)) = x \equiv \\
\quad x = \text{True} \land a = \text{unload} \land loc(s) \neq \text{dest(parcels_left}(s, s) \lor \\
\quad x = \text{misplaced}(s) \land (a \neq \text{unload} \lor loc(s) = \text{dest(parcels_left}(s, s)) \)

• Sensing Result Axioms:

\[
\begin{align*}
\text{SR(find\_src, s)} &= r \equiv \text{source(parcels_left}(s, s) = r \\
\text{SR(find\_dest, s)} &= r \equiv \text{dest(parcels_left}(s, s) = r \\
\text{SR(check\_done, s)} &= r \equiv \\
& \quad r = \text{yes} \land \text{parcels_left}(s) = 0 \lor \\
& \quad r = \text{no} \land \text{parcels_left}(s) \neq 0 \\
\text{SR(move}(x, s) &= r \equiv r = \text{ok} \\
\text{SR(load, s)} &= r \equiv r = \text{ok} \\
\text{SR(unload, s)} &= r \equiv r = \text{ok}
\end{align*}
\]

• Initial Situation Axiom:

\[
\begin{align*}
\forall n. (\text{source}(n, S_0) = \text{home} \lor \text{source}(n, S_0) = \text{office}) \land \\
\forall n. (\text{dest}(n, S_0) = \text{home} \lor \text{dest}(n, S_0) = \text{office}) \land \\
\text{loc}(S_0) = \text{home} \land \text{loaded}(S_0) = \text{False} \land \\
\text{parcels_left}(S_0) \geq 0 \land \text{misplaced}(S_0) = \text{False}
\end{align*}
\]
• Goal Condition:

\[ \text{parcels\_left} = 0 \land \text{misplaced} = \text{FALSE} \]

This example, intuitively, falls into our characterization of “one-dimensional planning problems.” To define this notion formally, let us first start with finite problems.

**Definition 8** (Finite problem). A planning problem \( \langle D, G \rangle \) is finite if it does not contain any predicate symbol other than \( =, \sqsubset \) and \( \text{Poss} \), and the sort object has a domain closure axiom of the form

\[ \forall x. x = o_1 \lor \cdots \lor x = o_l. \]

Intuitively, a finite problem has finitely many objects in the domain. Therefore, the number of ground fluents as well as their range is finite. Note that according to this definition, all fluents are functional, so there are no predicate symbols other than \( =, \sqsubset \), and \( \text{Poss} \).

A one-dimensional problem is like a finite problem except that there is a special distinguished fluent (called the planning parameter) that takes value from a new sort \textit{natural number}, there is a finite set of distinguished actions (called the decreasing actions) which decrement the planning parameter, and some of the functions (called sequence fluents) have an index argument of sort \textit{natural number}, in addition to the situation argument. In the case of the logistic example, the planning parameter is \text{parcels\_left}, the decreasing action is \text{unload}, and the sequence fluents are \text{source} and \text{dest}. The idea of a one-dimensional planning problem is that the basic action theory is restricted in how it can use the planning parameter and sequence fluents.

**Definition 9.** A planning problem \( \langle D', G' \rangle \) is one-dimensional, or 1d, with respect to an integer-valued fluent \( p \), if there is a finite problem \( \langle D, G \rangle \) whose functions include fluent
$f_0, f_1, \ldots, f_m$, and whose actions include $A_1, \ldots, A_d$, such that $\langle D', G' \rangle$ is derived from $\langle D, G \rangle$ as follows:

1. Replace the fluent $f_0$ with planning parameter $p$:

   (a) replace the SSA for $f_0$ by one for $p$ of the form

   \[
p(do(a, s)) = x \equiv x = p(s) - 1 \land \text{Dec}(a) \lor
   x = p(s) \land \neg\text{Dec}(a),
   \]

   where $\text{Dec}(a)$ stands for $(a = A_1 \lor \cdots \lor a = A_d)$;

   (b) replace all atomic formulas involving the term $f_0(s)$ in the $\Pi$, $\Phi$, $\Theta$ and $G[s]$ formulas by $p(s) = 0$, where $s$ is the free situation variable in those formulas;

   (c) remove all atomic formulas mentioning $f_0(S_0)$ in $D_0$, and add $p(S_0) \geq 0$ instead.

2. For all $1 \leq i \leq m$, replace fluents $f_i$ with sequence fluents $h_i$:

   (a) replace all terms $f_i$ in the $\Pi$, $\Phi$, $\Theta$ and $G[s]$ formulas by $h_i(p(s), s)$, where $s$ is the free variable as above;

   (b) remove all successor state axioms for $f_i$, and add the successor state axiom for $h_i$ of the form

   \[
h_i(n, do(a, s)) = y \equiv n = p(s) \land \Phi_{f_i}(a, y, s) \lor
   n \neq p(s) \land y = h_i(n, s),
   \]

   where $\Phi_{f_i}(a, y, s)$ is the right-hand side of the original successor state axiom for $f_i$;

   (c) replace all $f_i$ in $D_0$ with $h_i(n, S_0)$, where $n$ is a universally quantified variable of sort natural number.
Observe, for example, that in a one-dimensional problem, the occurrence of the integer planning parameter is limited to its own successor state axiom, in $D_0$, and as an argument to a sequence fluent. Any other use of it is restricted to testing whether it is 0. Similarly, one can only apply a sequence fluent to the current object as determined by the planning parameter (other than in $D_0$ where the index must quantify over all natural numbers). This ensures that the objects can be accessed sequentially in descending order, and that they do not interact with one another. It is not hard to see that logistic conforms to these requirements.

**Correctness Results on One-Dimensional Problems**

Given a one-dimensional planning problem and a candidate FSA plan, we want to know whether the plan achieves the goal no matter what value the planning parameter $p$ and the sequence fluents $h_i$ take. Unfortunately, the naive approach that exhaustively tests for all possible settings does not work, since the planning parameter may take infinitely many possible values, all of which need to be taken into account.

In this section, I will prove a correctness result of the following form: if one can prove that an FSA plan is correct under the assumption that $p(S_0) \leq N$ for a constant $N$ calculated based on the problem specification and the plan, it will follow that the plan is also correct without this assumption. In other words, correctness of the plan for a finite number of initial values of $p$ up to $N$ is sufficient.

**Theorem 4.** Suppose $(D, G)$ is a one-dimensional planning problem with planning parameter $p$. Let $N_0 = 2 + n \cdot l^m$, where $n$ is the number of program states associated with decreasing actions in an FSA plan axiomatized by $D_{fsa}$, $m$ is the total number of finite and sequence fluents, and $l$ is the total number of values that they can take. Then we have:

If $D \cup D_{fsa} \cup \{p(S_0) \leq N_0\} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s],$

then $D \cup D_{fsa} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s].$
The intuition behind this theorem is that both the number of program states and the number of indistinguishable situations are finite, so if the initial planning parameter is large enough, it will discover two similar configurations during the execution of the FSA plan, which share the same program state and the fluents take identical values except for the planning parameter. This is illustrated by the two black dots in the original execution line in Figure 4.1, representing the configurations after executing $\gamma_1$ and $\gamma_2$ from the initial situation, respectively. This enables us to construct a new initial state with smaller $p$, where the execution of $\gamma_1$ directly leads to $\langle q, s_2 \rangle$, so that $\gamma_3$ can follow and achieve the goal. In this way, the correctness of the FSA plan for a “larger” model can be reduced to that of a “smaller” model, which has been verified according to the assumptions of the theorem.

In order to prove this theorem, I first need a few lemmas. The first property is that in the execution of an FSA plan, the planning parameter monotonically decreases, and visits all integers between the initial and final values of $p$.

**Lemma 1.** Let $\mathcal{M}$ be a model of a one-dimensional action theory $\mathcal{D}$ with planning parameter $p$. For all $n, n' \in \mathbb{N}$, if

$$\mathcal{M} \models T^*(q, s, q', s') \land p(s) = n \land p(s') = n',$$

then $n \geq n'$, and for any $n''$ satisfying $n' < n'' \leq n$, there exist a constant $q''$ and a
Chapter 4. Finite Verifiability

ground situation term $s''$, such that

$$
\mathcal{M} \models T^*(q, s, q'', s'') \land T^*(q'', q', s') \land Dec(\gamma(q'')) \land p(s'') = n''.
$$

**Proof.** This follows from the shape of the successor state axiom for $p$, which constrains the actions to either leave the value of $p$ unchanged or decrease it by 1. \hfill \Box

During the execution of the plan, there may be multiple program states and situations where the planning parameter has the value $n''$. Here, $Dec(\gamma(q''))$ identifies a unique configuration where $q''$ is a decreasing state, i.e., a state whose associated action is among $A_1, \ldots, A_d$.

The next few properties deal with replacing a situation in an interpretation with a similar situation in a different interpretation. This similarity measure is defined as follows.

**Definition 10** ($(i,j)$-similarity). Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be models of $\mathcal{D}$, $s_1$ and $s_2$ be ground situation terms, and $i, j$ be natural numbers. Situation $s_1$ in interpretation $\mathcal{M}_1$ is $(i,j)$-similar to $s_2$ in $\mathcal{M}_2$, denoted by $\mathcal{M}_1(s_1,i) \sim \mathcal{M}_2(s_2,j)$, if

- $\mathcal{M}_1$ and $\mathcal{M}_2$ have identical domain for actions and objects;
- for all rigid (non-fluent) functions $r$, $r^{\mathcal{M}_1} = r^{\mathcal{M}_2}$;
- for all sequence fluents $h$, $h^{\mathcal{M}_1}(i, s^{\mathcal{M}_1}) = h^{\mathcal{M}_2}(j, s^{\mathcal{M}_2})$;
- for all finite fluents $f$, $f^{\mathcal{M}_1}(s_1^{\mathcal{M}_1}) = f^{\mathcal{M}_2}(s_2^{\mathcal{M}_2})$;
- for $p$, $p^{\mathcal{M}_1}(s_1^{\mathcal{M}_1}) = i$ and $p^{\mathcal{M}_2}(s_2^{\mathcal{M}_2}) = j$.

Intuitively, the situations $s_1$ and $s_2$ are indistinguishable in their respective interpretations, except for the differences in the planning parameter, since all finite fluents, as
well as sequence fluents at the current indexes, take identical values. As a result, the
execution of an FSA plan at a particular program state will be indistinguishable in the
two cases.

The \( \langle i, j \rangle \)-similarity relation is commutative and transitive, as formalized in Lemma 2. The proof of this lemma simply follows from multiple applications of Definition 10.

**Lemma 2.**

1. If \( M_1 \langle s_1, i \rangle \sim M_2 \langle s_2, j \rangle \), then \( M_2 \langle s_2, j \rangle \sim M_1 \langle s_1, i \rangle \).

2. If \( M_1 \langle s_1, i \rangle \sim M_2 \langle s_2, j \rangle \) and \( M_2 \langle s_2, j \rangle \sim M_3 \langle s_3, k \rangle \),
then \( M_1 \langle s_1, i \rangle \sim M_3 \langle s_3, k \rangle \).

The \( \langle i, j \rangle \)-similarity relation only requires the sequence fluents to agree at indexes \( i \) and \( j \), respectively. Sometimes, I want to compare, between models, the values of sequence fluents at multiple indexes. For this purpose, the following \( \xi \)-relationship is introduced.

**Definition 11.** Let \( \xi \subset \mathbb{N} \times \mathbb{N} \) be a set of pairs of natural numbers, and \( M_1 \) and \( M_2 \) be
two models of \( D \). \( M_1 \) \( \xi \)-relates to \( M_2 \), denoted by \( M_1 \{ \xi \} M_2 \), if for all sequence fluents \( h \) and all \( \langle i, j \rangle \in \xi \), \( h^{M_1}(i, S_0) = h^{M_2}(j, S_0) \).

Notice that \( \langle i, j \rangle \)-similarity relates two situations (in their respective interpretations), whereas \( \xi \)-relationship compares two interpretations. Two interpretations can be \( \xi \)-related, even if the finite fluents and the planning parameters are very different.

Given the concepts of \( \langle i, j \rangle \)-similarity and \( \xi \)-relationship, the next lemma captures the following intuition: given a model \( M \) of \( D \), one can construct another model \( M' \) of \( D \), which is almost the same as \( M \), but with different values for the planning parameter and sequence fluents.

**Definition 12.** A binary relation \( R \) is functional, if for any \( x, y, z \) such that \( \langle x, z \rangle \in R \) and \( \langle y, z \rangle \in R \), \( x = y \) holds.
Lemma 3. For any one-dimensional theory $D$, natural numbers $n, n' \in \mathbb{N}$, and functional relation $\xi \subset \mathbb{N} \times \mathbb{N}$ such that $\langle n, n' \rangle \in \xi$, if $M$ is a model of $D$ with $M \models p(S_0) = n$, then there is another model $M'$ of $D$ such that $M(\xi)M' \text{ and } M(S_0, n) \sim M'(S_0, n')$.

Proof. Let $M_0$ be the sub-model of $M$ with all objects involving non-initial situations removed. I need only construct a model $M'_0$, which is the same as $M_0$ except that $M'_0 \models p(S_0) = n'$ and the sequence fluents $h$ all satisfy this: for any $k$ and any $\langle i, j \rangle \in \xi$, if $M_0 \models h(i, S_0) = k$ then $M'_0 \models h(j, S_0) = k$. The existence of $M'$ from $M$ then follows by applying the Relative Satisfiability Theorem (Theorem 1 on page 8 of [Pirri and Reiter, 1999]).

Intuitively, $M'$ in Lemma 3 is similar to $M$, in the sense that the initial values of all finite fluents are identical in both interpretations, the planning parameter is $n'$ instead of $n$, and the initial values of the sequence fluents in $M'$ at some indexes are mapped from their initial values in $M$ at possibly other indexes according to $\xi$. Later in the proof of the theorems, I shall use this property and show that the (in)correctness of an FSA plan for a “larger” interpretation can be reduced to the (in)correctness for a “smaller” interpretation.

Finally, I am ready to present a property about the execution of an FSA plan in a one-dimensional theory. Suppose I have two situations in two models where the planning parameter is positive in both, all sequence fluents agree at these and smaller indexes, and all the other fluents have identical values. Then starting from the same plan state, the execution of the FSA plan will be the same in each model. Lemma 4 formalizes this intuition.

Lemma 4. Let $M_1$ and $M_2$ be two models of a one-dimensional action theory $D$, and $s_1$ and $s_2$ be two situation terms. Suppose $M_1 \langle s_1, n_1 \rangle \sim M_2 \langle s_2, n_2 \rangle$ for $n_1, n_2 > k$, and $M_1(\xi)M_2$ where $\langle n_1 - i, n_2 - i \rangle \in \xi$ for all $0 < i \leq k$. For any states $q, q'$ and action
sequence $\sigma$ such that $p(s'_i) = n_i - k$, where $s'_i = do(\sigma, s_i)$, I have

$$M_1 \models T^*(q, s_1, q'_1, s'_1) \iff M_2 \models T^*(q, s_2, q'_2, s'_2).$$

Furthermore, $M_1(s'_1, n_1 - k) \sim M_2(s'_2, n_2 - k)$.

Proof. By induction over the length of $\sigma$. \qed

A special case of Lemma 4 is when $n_1 = n_2$. Then $s$ in $\mathcal{M}$ and $s'$ in $\mathcal{M}'$ become isomorphic, that is, they are not only similar, but also have the same value for $p$. In this case, the execution trace will be identical, even after the planning parameter reaches 0. This leads to the following:

Corollary 1. Let $\mathcal{M}_1(s_1, n) \sim \mathcal{M}_2(s_2, n)$ and $\mathcal{M}_1(\xi) \mathcal{M}_2$, where $\langle i, i \rangle \in \xi$ for all $0 \leq i \leq n$. For any constants $q, q'$ and action sequence $\sigma$, let $s'_i = do(\sigma, s_i)$, then

$$\mathcal{M}_1 \models T^*(q, s_1, q'_1, s'_1) \iff \mathcal{M}_2 \models T^*(q, s_2, q'_2, s'_2).$$

Moreover, $M_1(s'_1, n') \sim M_2(s'_2, n')$, for some $n'$.

With the properties above, I am ready to prove Theorem 4.

Proof of Theorem 4. Suppose, for the sake of contradiction, that $
\mathcal{D} \not\models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s].$

Then there is a smallest $n > N_0$ such that for some model $\mathcal{M}$ of $\mathcal{D}$ I have $\mathcal{M} \models p(S_0) = n \land \neg(\exists s. T^*(Q_0, S_0, Q_F, s) \land G[s])$.

According to Lemma 3, there exists another model $\mathcal{M}_1$ of $\mathcal{D}$ satisfying $\mathcal{M}(S_0, n) \sim \mathcal{M}_1(S_0, n - 1)$ and $\mathcal{M}(\xi_1) \mathcal{M}_1$, where $\langle i, i - 1 \rangle \in \xi_1$ for all $1 \leq i \leq n$.

Since $\mathcal{M}_1 \models p(S_0) = n - 1 < n$, according to our assumption, I get $\mathcal{M}_1 \models T^*(Q_0, S_0, Q_F, s) \land G[s]$ for some situation term $s$. Based on the value of $p(s)$, there are two cases:
Case 1: $M_1 \models p(s) = n' > 0$: By Lemma 4, $M \models T^*(Q_0, S_0, Q_F, s)$, and furthermore $M_1\langle s, n' \rangle \sim M\langle s, n' + 1 \rangle$, so $M \models G[s]$. This contradicts the assumption that $M \not\models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s]$.

Case 2: $M_1 \models p(s) = 0$: By Lemma 1, there is an action sequence $\sigma$ and decreasing state $q$ such that

$$M_1 \models T^*(Q_0, S_0, q, do(\sigma, S_0)) \land p(\sigma, S_0)) = 1.$$ 

By Lemma 4,

$$M \models T^*(Q_0, S_0, q, do(\sigma, S_0)) \land p(\sigma, S_0)) = 2.$$ 

Since $\sigma$ reduces $p$ from $n$ to 2 in $M$, by Lemma 1, it must contain more than $n \cdot l^n$ decreasing actions. So there must be two points in the execution of $\sigma$ with the same state: there is some program state $q'$ and $\sigma = \alpha\beta\gamma$ such that

$$M \models T^*(Q_0, S_0, q', do(\alpha, S_0)) \land T^*(q', do(\alpha, S_0), q', do(\alpha\beta, S_0)),$$

where $M\langle do(\alpha, S_0), u \rangle \sim M\langle do(\alpha\beta, S_0), v \rangle$, for some $u > v$. Furthermore,

$$M \models \neg\exists s. T^*(q', do(\alpha\beta, S_0), Q_F, s) \land G[s].$$

With another application of Lemma 3, there is a model $M_u$ of $\mathcal{D}$ such that $M\langle S_0, n \rangle \sim M_u\langle S_0, n - (u - v) \rangle$ and $M\{\xi_u\}M_u$, where $\langle i, i - (u - v) \rangle \in \xi_u$ for all $u \leq i \leq n$ and $\langle i, i \rangle \in \xi_u$ for all $0 \leq i < v$. By Lemma 4 again, I have that

$$M_u \models T^*(Q_0, S_0, q', do(\alpha, S_0)).$$
By Lemma 2, $\mathcal{M}\langle do(\alpha \beta, S_0), v \rangle \sim \mathcal{M}_u\langle do(\alpha, S_0), v \rangle$. Therefore, by Corollary 1,

$$\mathcal{M}_u \models \neg \exists s. T^*(q', do(\alpha, S_0), Q_F, s) \land G[s].$$

So I obtain

$$\mathcal{M}_u \models p(S_0) = n - (u - v) \land \neg \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s],$$

which contradicts the assumption that $\mathcal{M} \models p(S_0) = n$ has the smallest $n$ that fails the FSA plan.

Notice that in Theorem 4, the bound $N_0$ is exponential in the number of ground functions, and thus can be extremely large even for relatively simple action theories.

As a result, it is desirable to have a tighter bound for a similar finite-verifiability result. In order to do so, one could narrow down the number of values finite and sequence fluents may take by observing their successor state axioms, instead of assuming that they range over all finite objects. Suppose function $f_j$ (or $h_j(n, s)$) only takes $l_j$ different values, then we can prove a variant of Theorem 4 with a bound $N'_0 = 2 + n \prod_{j=1}^m l_j$, which is usually much smaller than $N_0$. I omit the details here, as $N'_0$ is still exponential in the number of functions.

To obtain a more practical bound in a similar flavor, I introduce another theorem, where the bound $N_t$ is not declaratively specified, but instead only spelled out using the necessary condition on $N_t$.

**Theorem 5.** Suppose $\langle D, G \rangle$ is a one-dimensional planning problem with planning parameter $p$, and that $D_{fba}$ axiomatizes some FSA plan. Let $\text{Seen}(q, s)$ denote the abbreviation for

$$\exists s'. T^*(Q_0, S_0, q, s') \land p(s') > 1 \land \bigwedge f(s) = f(s') \land \bigwedge h(p(s), s) = h(p(s'), s'),$$
where the first conjunction is over the finite fluents $f$, and the second over sequence fluents $h$. Suppose $N_t > 0$ satisfies

$$D \cup D_{fsa} \cup \{ p(S_0) = N_t \} \models \forall q, s. T^*(Q_0, S_0, q, s) \land p(s) = 1 \supset \text{Seen}(q, s).$$

Then we have the following:

If $$D \cup D_{fsa} \cup \{ p(S_0) \leq N_t \} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s],$$
then $$D \cup D_{fsa} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s].$$

Intuitively, $N_t$ has to be large enough so that a similar situation to the one that decrements the planning parameter from 1 to 0 occurs earlier in the execution trace. The main idea is that this condition alone suffices to guarantee the existence of the splicing points illustrated in Figure 4.1 above. Following this intuition, the proof for this theorem is sketched below, which is very similar to the one for Theorem 4, except for the way $\sigma$ is decomposed.

**Proof of Theorem 5.** Let $\mathcal{M}$ be the smallest interpretation on which the FSA plan fails to achieve the goal, and $\mathcal{M}(S_0, n) \sim \mathcal{M}_t(S_0, N_t)$. Further let

$$\mathcal{M}_t \models T^*(Q_0, S_0, q, do(\sigma, S_0)) \land p(do(\sigma, S_0)) = 1$$

where $q$ is a decreasing state, then by the constraint on $N_t$, there must be a decomposition $\sigma = \alpha \beta$ such that

$$\mathcal{M}_t \models T^*(Q_0, S_0, q, do(\alpha, S_0)) \land T^*(q, do(\alpha, S_0), q, do(\alpha \beta, S_0)), \quad \text{and} \quad \mathcal{M}_t(do(\alpha, S_0), u) \sim \mathcal{M}_t(do(\alpha \beta, S_0), v),$$

where $u > 1$ and $v = 1$. The rest of the proof follows as in Theorem 4. 
\[\square\]
Chapter 4. Finite Verifiability

It has been shown that for many one-dimensional planning problems, $N_t$ is very small, usually much smaller than $N_0$, so verification of a plan to this bound is extremely efficient and gives a correctness guarantee [Hu and Levesque, 2010]. For example, in the logistics example, using Theorem 4, we need to verify up to $N_0 = 514$ objects, whereas using Theorem 5, we only need $N_t = 2$. It is also shown that using these theoretical results enables the FSAPlanner, which I will elaborate in Chapter 5, to efficiently generate provably correct FSA plans for one-dimensional planning problems.

4.1.2 $k$-Dimensional Planning Problems

The requirement of having only one unbounded fluent in one-dimensional problems is a relatively restricting assumption, ruling out the possibility of having more than one unbounded and independent properties, such as multiple stacks of blocks in a blocks world setting.

In this section, I will generalize the finite verifiability of one-dimensional planning problems to $k$-dimensional, i.e., problems with $k$ monotonically decreasing integer parameters. I shall show that finite verifiability is also achievable for this more general class of problems.

Notice that I only consider decreasing parameters here, and will postpone the discussion on allowing for increasing parameters to the next section.

Definition 13. A planning problem $\langle D', G' \rangle$ is $k$-dimensional with respect to integer-valued fluents $p_1, \cdots, p_k$, if there is a finite problem $\langle D, G \rangle$ whose functions include fluents $f'_1, \cdots, f'_k$ and rigid $f_1, \cdots, f_m$, such that $\langle D', G' \rangle$ is derived from $\langle D, G \rangle$ as follows:

1. Replace the fluent $f'_i$ with planning parameter $p_i$: 
(a) replace the SSA for $f'_i$ by one for $p_i$ of the form

$$p_i(\text{do}(a, s)) = x \equiv x = p_i(s) - 1 \land \text{Dec}_i(a) \lor x = p_i(s) \land \neg\text{Dec}_i(a),$$

where $\text{Dec}_i(a)$ is of the form $a = A_1^{(i)} \lor \cdots \lor a = A_d^{(i)}$ and is true iff $a$ is one of the decreasing actions $A_1^{(i)}, \ldots, A_d^{(i)}$ for fluent $p_i$.

(b) replace all atomic formulas involving the term $f'_i(s)$ in the $\Pi$, $\Phi$, $\Theta$ and $G[s]$ formulas by $p_i(s) = 0$, where $s$ is the free situation variable in those formulas;

(c) remove all atomic formulas mentioning $f'_i(S_0)$ in $D_0$, and then add $p_i(S_0) \geq 0$ instead.

2. Replace the rigid $f_1, \ldots, f_m$ with sequence fluents $h_1, \ldots, h_m$:

(a) replace all terms $f_j$ in the $\Pi$, $\Phi$, $\Theta$ and $G[s]$ formulas by $h_j(p_{j'}(s), s)$, where $s$ is the free variable as above, and $1 \leq j' \leq k$ identifies the only planning parameters to index $h_j$;

(b) replace all $f_j$ in $D_0$ with $h_j(n_j, S_0)$, where $n_j$ is a universally quantified variable of sort natural number.

Definition 13 is very similar to that of one-dimensional planning problems, except that $k$ integer fluents exist instead of just one, and each sequence fluent $h_j$ has one fixed indexing parameter $p_{j'}$. Such action theories can be used to naturally model multiple unbounded and independently-decreasing properties, e.g., $k$ stacks of blocks, $k$ different supplies of products, etc.

Let us call decreasing a program state in an FSA plan for a $k$-dimensional planning problem, if the action associated with the state is $A_j^{(i)}$, a decreasing action for at least one planning parameter $p_i$. Then, similar to Theorem 4, we can obtain a finite verifiability theorem for $k$-dimensional planning problems.
Theorem 6. Suppose \( \langle D, G \rangle \) is a \( k \)-dimensional planning problem with planning parameters \( p_1, \ldots, p_k \). Let \( N_k = 1 + k \cdot (1 + n \cdot l^m \cdot 2^{k-1}) \), where \( n \) is the number of decreasing program states in an FSA plan axiomatized by \( D_{fsa} \), \( m \) is the total number of finite and sequence fluents, and \( l \) is the total number of values that they can take. Then we have:

If \( D \cup D_{fsa} \cup \{ \sum_{i=1}^{k} p_i(S_0) \leq N_k \} \models \exists s.\, T^*(Q_0, S_0, Q_F, s) \land G[s] \),

then \( D \cup D_{fsa} \models \exists s.\, T^*(Q_0, S_0, Q_F, s) \land G[s] \).

Proof sketch. Suppose, for the sake of contradiction, that there exists a model \( M \) of \( D \cup D_{fsa} \), in which \( \sum_{i=1}^{k} p_i(S_0) \) has the smallest value, such that

\( M \not\models \exists s.\, T^*(Q_0, S_0, Q_F, s) \land G[s] \).

Because \( D \cup D_{fsa} \cup \{ \sum_{i=1}^{k} p_i(S_0) \leq N_k \} \models \exists s.\, T^*(Q_0, S_0, Q_F, s) \land G[s] \), it follows that \( M \models \sum_{i=1}^{k} p_i(S_0) > N_k \). According to the pigeon hole principle, at least one of the planning parameters \( p_i \) has an initial value greater than \( 2 + n \cdot l^m \cdot 2^{k-1} \).

Following a similar argument to the one in the proof of Theorem 4, a contradiction will be reached with the assumption that \( M \) is a model with the smallest value for \( \sum_{i=1}^{k} p_i(S_0) \). The additional factor \( 2^{k-1} \) here is used to ensure that for any parameter \( p_i \), its value is either both zero or both non-zero in the two similar configurations found in the proof for the splicing operation.

\[ \square \]

### 4.2 Problems with Restricted Increasing Parameters

So far, I have studied finitely verifiable planning problems whose unbounded fluents can only be decreased. Given the positive result for such verifiable classes, it is interesting to ask whether problems with increasing effects may also be finitely verifiable.

For instance, in the blocks world, decreasing a planning parameter can be used to model the unstacking action that removes the topmost block of a stack. In order to
also model the stacking action that puts a new block onto a stack, the corresponding
planning parameter needs to be incremented. This happens to stack $B$ in the striped
tower problem, for example, when the robot is holding a block with the wrong colour
for stack $C$. The question is then, can we freely use increasing effects in a basic action
theory, yet still maintain the desired finite verifiability?

As we shall see in the next section, the addition of increasing capability can easily
make an action theory expressive enough to encode Turing machines in many cases, and
thus finite verifiability is impossible for those problems. However, in this section, I shall
identify a type of planning problem with very limited increasing capability, which is in
fact finitely verifiable.

Consider a planning problem with two non-negative integer planning parameters,
where one of the parameters, $p_1$, can only be decreased as before, but the other, $p_2$, can
be both increased and decreased by one. The initial value of $p_1$ may be any non-negative
integer, while $p_2$ always starts at 0. We further restrict that there are no sequence fluents
in the domain. As usual, the goal is achieved only when $p_1 = p_2 = 0$. Let us call this
type of planning problems 2-dimensional with limited incrementability.

Definition 14. A planning problem $\langle D', G \rangle$ is 2-dimensional with limited incrementabil-
ity if there is a 2-dimensional planning problem $\langle D, G \rangle$ with planning parameters $p_1$ and
$p_2$ and without any sequence fluent, such that $D'$ is derived from $D$ as follows:

- Replace the successor state axiom for $p_2$ by

$$p_2(do(a, s)) = y \equiv y = p_2(s) - 1 \land Dec_2(a) \lor$$
$$y = p_2(s) + 1 \land Inc(a) \lor$$
$$y = p_2(s) \land \neg Dec_2(a) \land \neg Inc(a),$$

where $Inc(a)$ is of the form $a = A_1 \lor \cdots \lor a = A_I$ and is true iff $a$ is one of the in-
creasing actions $A_1, \cdots, A_I$, none of which is also a decreasing action $A_1^{(2)}, \cdots, A_I^{(2)}$.
for $p_2$.

- Remove all atomic formulas mentioning $p_2(S_0)$ in $D_0$, and then add $p_2(S_0) = 0$.

With the added capability of increasing $p_2$, the planning parameter can potentially grow indefinitely. The question is, if one has verified that an FSA plan finally brings $p_2$ back to 0 and satisfies the goal $G$ for finitely many cases, is it guaranteed to do so for all the infinitely many other cases?

Intuitively, if the goal is achieved by some FSA plan, the execution will have two phases. In the first phase, $p_1$ gradually decreases to 0 while $p_2$ may change arbitrarily; in the second phase, $p_1$ will remain at 0, because it cannot be incremented by any action, and $p_2$ will reach 0 after some increasing and decreasing actions. The goal is achieved after both $p_1$ and $p_2$ have the value 0.

First, let us prove a lemma about the regularity in the first phase, due to the fact that $p_1$ always monotonically decreases to 0.

For a cleaner presentation, we ignore for now any fluent other than the planning parameters and will factor in the finite fluents in the Theorem 7. I will use the triple $\langle i, j, s \rangle$ to denote the configuration where $p_1 = i$, $p_2 = j$, and the FSA plan is in program state $s$. The notation $\langle i, j, s \rangle \xrightarrow{\alpha} \langle i', j', s' \rangle$ means configuration $\langle i', j', s' \rangle$ can be reached by executing the action sequence $\alpha$ from configuration $\langle i, j, s \rangle$. When the actual values are irrelevant, I sometimes use “$\ast$” to mean any sequence of zero or more actions, and “$\cdot$” to mean any value for the fluent.

**Lemma 5.** Given a two-dimensional problem with limited incrementability, and an FSA plan with $n$ program states, assuming there are no finite fluents, if $\langle i, 0, s \rangle \xrightarrow{\ast} \langle 0, \cdot, \cdot \rangle$ for $i = 0, 1, 2, \cdots, n^2$, then there exist $0 < \delta i \leq n^2$ and $0 \leq \delta j \leq n^2$ such that for all $i > n^2$,
there exist \(0 \leq i_0 \leq n^2, j_0 \geq 0, k > 0\) and \(s'\), satisfying \(i = i_0 + k \cdot \delta i\) and

\[
\begin{array}{l}
\langle i_0 + 0 \cdot \delta i, 0, s \rangle \xrightarrow{*} \langle 0, j_0 + 0 \cdot \delta j, s' \rangle \\
\langle i_0 + 1 \cdot \delta i, 0, s \rangle \xrightarrow{*} \langle 0, j_0 + 1 \cdot \delta j, s' \rangle \\
\vdots \quad \vdots \\
\langle i_0 + k \cdot \delta i, 0, s \rangle \xrightarrow{*} \langle 0, j_0 + k \cdot \delta j, s' \rangle
\end{array}
\]

Intuitively, this lemma says that if \(p_1\) always goes to 0 for sufficiently many different initial values \((0, 1, \cdots, n^2)\), then we can identify the periods \(\delta i\) for \(p_1\) and \(\delta j\) for \(p_2\), in the sense that if a base case \(\langle i_0, 0, s \rangle \xrightarrow{*} \langle 0, j_0, s' \rangle\) holds, then changing the initial value of \(p_1\) by a multiple of the period will result in changing the end value of \(p_2\) by the same multiple, i.e., \(\langle i_0 + k \cdot \delta i, 0, s \rangle \xrightarrow{*} \langle 0, j_0 + k \cdot \delta j, s' \rangle\). Moreover, any large enough initial value of \(p_1\) \((i > n^2)\) has a small base \(i_0\) \((0 \leq i_0 \leq n^2)\).

Proof. Consider the execution of \(p_1 = n^2\): suppose \(\langle n^2, 0, s \rangle \xrightarrow{*} \langle 0, j, s' \rangle\). Since each action can decrease \(p_1\) by at most 1, this requires at least \(n^2\) steps. So, let us consider the first \(n^2\) steps in this execution trace:

\[
\langle n^2, 0, s \rangle \xrightarrow{1} \langle i_1, j_1, s_1 \rangle \xrightarrow{1} \langle i_2, j_2, s_2 \rangle \xrightarrow{1} \cdots \xrightarrow{1} \langle i_n, j_n, s_n \rangle \xrightarrow{*} \\
\]

There are \(n^2 + 1\) configurations along this trace prefix. Depending on whether \(p_2 \geq n\) in some of these configurations, there are two cases:

- \(p_2 < n\) in all these \(n^2 + 1\) configurations:
  
  In this case, \(p_2\) can take at most \(n\) different values \((0, 1, \cdots, n-1)\), and the FSA plan can be in at most \(n\) different program states, so there are at most \(n^2\) different combinations. According to the pigeon hole principle, there must be at least two configurations in this prefix that have identical \(p_2\) value and in the same program state.

Let them be \(\langle u, j^*, s^* \rangle\) and \(\langle v, j^*, s^* \rangle\), i.e., \(\langle n^2, 0, s \rangle \xrightarrow{\alpha} \langle u, j^*, s^* \rangle \xrightarrow{\beta} \langle v, j^*, s^* \rangle\),
where $|\beta| > 0$. Obviously, $u > v$, since according to our assumption, $p_1$ can only be decreased, so $u \geq v$. Furthermore, $u \neq v$, since otherwise the execution would be an infinite loop, and $\langle 0, \cdot, \cdot \rangle$ would never be reached. Let $\delta i = u - v$, then for any $i > n^2 - v$, if $\langle i, 0, s \rangle \xrightarrow{\alpha} \langle i', j^*, s^* \rangle \xrightarrow{\beta} \langle i'', j^*, s^* \rangle \xrightarrow{*} \langle 0, j, s' \rangle$, then $\langle i - \delta i, 0, s \rangle \xrightarrow{\alpha} \langle i'', j^*, s^* \rangle \xrightarrow{*} \langle 0, j, s' \rangle$. If $i - \delta i > n^2 - v$, this can be applied repeatedly, until for some $k > 0$ and $i - k \cdot \delta i \leq n^2 - v$, $\langle i - k \cdot \delta i, 0, s \rangle \xrightarrow{*} \langle 0, j, s' \rangle$.

Now, let $i_0 = i - k \cdot \delta i$, $j_0 = j$ and $\delta j = 0$, then the lemma holds.

- $p_2 = n$ in at least one of these $n^2 + 1$ configurations:

In this case, identify the first configuration where $p_2 = n$, and from there backward, the last configurations where $p_2 = n - 1, n - 2, \cdots, 1, 0$, respectively, i.e.,

$$\langle n^2, 0, s \rangle \xrightarrow{*} \langle i(0)^{0}, 0, s(0)^{0} \rangle \xrightarrow{*} \langle i(1)^{1}, s(1)^{1} \rangle \xrightarrow{*} \cdots \langle i(n)^{n}, n, s(n) \rangle \xrightarrow{*} \cdots,$$

such that

- $\langle i(n)^{n}, n, s(n) \rangle$ is the first configuration where $p_2 = n$;

- all configurations between $\langle i(k)^{k}, k, s(k) \rangle$ and $\langle i(n)^{n}, n, s(n) \rangle$ satisfy $p_2 > k$ for $k = 0, 1, \cdots, n - 1$.

Figure 4.2 shows how $\langle i(k)^{k}, k, s(k) \rangle$ are identified in the execution trace.

Since there are $n + 1$ configurations of $\langle i(k)^{k}, k, s(k) \rangle$, but only $n$ different program states, there must be at least two of these configurations that are in the same state. Let them be $\langle i(u)^{u}, u, s^* \rangle$ and $\langle i(v)^{v}, v, s^* \rangle$, where $u < v$ and $i(u) > i(v)$. Let $\delta i = i(u) - i(v)$, $\delta j = v - u$, then for any $i > n^2 - i(v)$, if $\langle i, 0, s \rangle \xrightarrow{\alpha} \langle i', u, s^* \rangle \xrightarrow{\beta} \langle i'', v, s^* \rangle \xrightarrow{\gamma} \langle 0, j, s' \rangle$, then $\langle i - \delta i, 0, s \rangle \xrightarrow{\alpha} \langle i'', v - \delta j, s^* \rangle \xrightarrow{\gamma} \langle 0, j - \delta j, s' \rangle$. The rest of the proof follows like in the first case.
Lemma 6. Given an FSA plan with \( n \) program states for a two-dimensional problem with limited incrementability and no finite fluents, where \( p_1 \) is the decrement-only parameter and \( p_2 \) is the bidirectional parameter, if, for all \( j \geq 0 \), \( \delta j \geq 0 \) and \( k = 0, 1, 2, \ldots, n \), we have

\[
\langle 0, j + k \cdot \delta j, s \rangle \xrightarrow{\ast} \langle 0, 0, s' \rangle,
\]

then this transition holds for all \( k \geq 0 \).

Proof. The case for \( \delta j = 0 \) is trivial. The case for \( \delta j > 0 \) is very similar to the proof for Theorem 4, and I shall sketch below:

Since \( p_1 \) can only decrease and its initial value is 0, it will remain at 0 throughout the
execution. Now, consider the execution trace for \( k = n \):

\[
\langle 0, j + n \cdot \delta j, s \rangle = \langle 0, j + n \cdot \delta j, s_n \rangle \xrightarrow{*} \langle 0, j + (n-1) \cdot \delta j, s_{n-1} \rangle \xrightarrow{*} \ldots \ldots \langle 0, j + 1 \cdot \delta j, s_1 \rangle \xrightarrow{*} \langle 0, j + 0 \cdot \delta j, s_0 \rangle \xrightarrow{*} \langle 0, 0, s' \rangle,
\]

where \( \langle 0, j + k \cdot \delta j, s_i \rangle \) is the first configuration where \( p_2 = j + k \cdot \delta j \).

Now suppose, for the sake of contradiction, that there exists \( k \) such that

\[
\langle 0, j + k \cdot \delta j, s \rangle \not\rightarrow \langle 0, 0, s' \rangle.
\]

Among all such \( k \)'s, we pick the smallest \( k^* \). Obviously, \( k^* > n \), so the execution trace starting from this configuration must be of the form

\[
\langle 0, j + k^* \cdot \delta j, s \rangle = \langle 0, j + k^* \cdot \delta j, s_n \rangle \xrightarrow{*} \langle 0, j + (k^*-1) \cdot \delta j, s_{n-1} \rangle \xrightarrow{*} \ldots \ldots \langle 0, j + (k^*-n+1) \cdot \delta j, s_1 \rangle \xrightarrow{*} \langle 0, j + (k^*-n) \cdot \delta j, s_0 \rangle \not\rightarrow \langle 0, 0, s' \rangle.
\]

There are only \( n \) program states in the FSA plan, but there are \( n + 1 \) such \( s_i \)'s in the execution trace, so there must be at least two configurations that share the same program state. Let them be \( \langle 0, j + u \cdot \delta j, s^* \rangle \) and \( \langle 0, j + v \cdot \delta j, s^* \rangle \), where \( u > v \), i.e.,

\[
\langle 0, j + k^* \cdot \delta j, s \rangle \xrightarrow{\alpha} \langle 0, j + u \cdot \delta j, s^* \rangle \xrightarrow{\beta} \langle 0, j + v \cdot \delta j, s^* \rangle \not\rightarrow \langle 0, 0, s' \rangle.
\]
Then we can derive that

\[ \langle 0, j + (k^* - u + v) \cdot \delta j, s \rangle \xrightarrow{\alpha} \langle 0, j + v \cdot \delta j, s^* \rangle \not\rightarrow \langle 0, 0, s' \rangle, \]

contradicting the assumption that \( k^* \) is the smallest failing \( k \). As a result, the lemma holds.

**Lemma 7.** Given a two-dimensional problem with limited incrementability as in the previous lemma and an FSA plan with \( n \) states, if \( \langle i, 0, Q_0 \rangle \xrightarrow{*} \langle 0, 0, Q_F \rangle \) for \( i = 0, 1, 2, \cdots, n^2(n+1) \), then the transition holds for all \( i \geq 0 \).

**Proof.** I need to prove that for all initial value of \( p_1, \langle p_1, 0, Q_0 \rangle \xrightarrow{*} \langle 0, 0, Q_F \rangle \). Obviously, for \( p_1 \leq n^2(n+1) \), it is trivially true. So below we focus on the cases where \( p_1 > n^2(n+1) \).

Since \( \langle i, 0, Q_0 \rangle \xrightarrow{*} \langle 0, \cdot, \cdot \rangle \), for \( 0 \leq i \leq n^2 < n^2(n+1) \), according to Lemma 5, there exist \( 0 < \delta i \leq n^2 \) and \( 0 \leq \delta j \leq n^2 \) such that for all \( p_1 > n^2(n+1) > n^2 \), there exist \( i_0 < n^2 \) and \( j_0 \), such that \( p_1 = i_0 + k \cdot \delta i \), and

\[
\begin{align*}
\langle i_0 + 0 \cdot \delta i, 0, Q_0 \rangle &\xrightarrow{*} \langle 0, j_0 + 0 \cdot \delta j, s' \rangle \\
\langle i_0 + 1 \cdot \delta i, 0, Q_0 \rangle &\xrightarrow{*} \langle 0, j_0 + 1 \cdot \delta j, s' \rangle \\
&\quad \cdots \quad \cdots \\
\langle i_0 + k \cdot \delta i, 0, Q_0 \rangle &\xrightarrow{*} \langle 0, j_0 + k \cdot \delta j, s' \rangle
\end{align*}
\]

Since \( i_0 \leq n^2 \) and \( \delta i \leq n^2 \), we know that \( k \geq n \) and that \( i_0 + n \cdot \delta i \leq n^2(n+1) \). So according to our assumption,

\[
\begin{align*}
\langle i_0 + 0 \cdot \delta i, 0, Q_0 \rangle &\xrightarrow{*} \langle 0, j_0 + 0 \cdot \delta j, s' \rangle \xrightarrow{*} \langle 0, 0, Q_F \rangle \\
\langle i_0 + 1 \cdot \delta i, 0, Q_0 \rangle &\xrightarrow{*} \langle 0, j_0 + 1 \cdot \delta j, s' \rangle \xrightarrow{*} \langle 0, 0, Q_F \rangle \\
&\quad \cdots \quad \cdots \\
\langle i_0 + n \cdot \delta i, 0, Q_0 \rangle &\xrightarrow{*} \langle 0, j_0 + n \cdot \delta j, s' \rangle \xrightarrow{*} \langle 0, 0, Q_F \rangle.
\end{align*}
\]
According to Lemma 6,

\[ \langle 0, j_0 + k \cdot \delta j, s' \rangle \xrightarrow{\ast} \langle 0, 0, Q_F \rangle \]

As a result,

\[ \langle p_1, 0, Q_0 \rangle = \langle i_0 + k \cdot \delta i, 0, Q_0 \rangle \xrightarrow{\ast} \langle 0, j_0 + k \cdot \delta j, s' \rangle \xrightarrow{\ast} \langle 0, 0, Q_F \rangle \]

Since this holds for any value of \( p_1 \), the lemma is proved.

With the Lemmas above, I am now ready to present the formal argument that 2-dimensional planning problems with limited incrementability are finitely verifiable.

**Theorem 7.** Suppose \( \langle D, G \rangle \) is a 2-dimensional planning problem with limited incrementability, where \( p_1 \) is the decreasing planning parameter and \( p_2 \) may be increased. Let \( N_2 = n^2 \cdot l^{2m}(n \cdot l^m + 1) \), where \( n \) is the number of program states in an FSA plan axiomatized by \( D_{fsa} \), \( m \) is the number of finite functions, and \( l \) is the number of values that they can take. Then we have:

If \( \mathcal{D} \cup \mathcal{D}_{fsa} \cup \{ p_1(S_0) \leq N_2 \} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s] \),

then \( \mathcal{D} \cup \mathcal{D}_{fsa} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s] \).

**Proof.** Lemma 7 shows the correctness of Theorem 7 when there are no finite fluents in the domain.

If there exist \( m \) finite fluents, and each may take \( l \) different values, it can be reduced to the previous case by introducing a new FSA plan with exponentially many program states. More specifically, for each program state in the original FSA plan, create \( l^m \) program states, one for each possible assignment of the \( m \) finite fluents. Add an edge between any two such states iff an action changes the finite fluents from one state to the other while keeping the planning parameters unchanged.
With this construction, the original FSA plan is correct for the original planning problem with finite fluents if the new FSA plan is correct for the problem with all finite fluents removed. The bound is thus obtained.

4.3 Non-Finitely Verifiable Problems

In the previous two sections, I have shown that there are planning problems, the correctness of whose FSA plans can be verified by testing the plans on finitely many cases. However, all the problems had relatively strong restrictions on the shape of their basic action theories and goals. For example, the integer planning parameters can only be decremented (Section 4.1), or no sequence fluent can exist (Section 4.2).

It is thus interesting to ask whether we can relax those restrictions, and obtain finite verifiability for more general classes of planning problems. For example, are planning problems that have both increasing effects and sequence fluents finitely verifiable? Can we allow for more than one increasing planning parameter?

In this section, I shall show that the restrictions we had in the previous sections were indeed necessary, in that slight generalizations of the studied types of problems like those above will result in the loss of finite verifiability.

4.3.1 One-Dimensional Problems with Increasing Effects

First, let us generalize the one-dimensional planning problems in Section 4.1.1 to allow some actions to increment the planning parameter by 1. Let us call those actions the increasing actions.

**Definition 15.** A planning problem \((D', G)\) is one-dimensional-plus, if there is a one-dimensional problem \((D, G)\) with planning parameter \(p\) such that \(D'\) is obtained by re-
placing the successor state axiom for \( p \) in \( D \) with

\[
p(do(a,s)) = x \equiv x = p(s) - 1 \land \text{Dec}(a) \lor \\
x = p(s) + 1 \land \text{Inc}(a) \lor \\
x = p(s) \land \neg \text{Dec}(a) \land \neg \text{Inc}(a).
\]

As usual, we assume that there are two disjoint sets of actions called the decreasing and the increasing actions, and \( \text{Dec}(a) \) (respectively, \( \text{Inc}(a) \)) is true iff \( a \) is a decreasing (respectively, increasing) action.

With this small change in the restrictions of the basic action theory, it is tempting to believe that finite verifiability still holds for one-dimensional-plus problems. Similar to Theorem 4, we may have the following conjecture.

**Conjecture 1.** For any one-dimensional-plus planning problem \( \langle D, G \rangle \) with planning parameter \( p \), there exists a computable integer bound \( N \) such that the following holds:

\[
\text{If } D \cup D_{\text{fsa}} \cup \{p(S_0) \leq N\} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s], \text{ then } D \cup D_{\text{fsa}} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s].
\]

Unfortunately, this conjecture is not true, since there are one-dimensional-plus planning problems whose FSA plans cannot be finitely verified. We shall show this next.

**Theorem 8.** Conjecture 1 is false.

**Proof.** I shall prove this theorem, and thus the invalidity of Conjecture 1 by reducing the halting problem into finitely verifying an FSA plan for a one-dimensional-plus problem.

The rough idea is to show that given any Turing machine \( M \), I can design an FSA plan for a one-dimensional-plus planning problem such that the FSA plan always achieves the goal if and only if the Turing machine never halts.
Chapter 4. Finite Verifiability

The one-dimensional-plus planning problem models the dynamics of a Turing machine with a semi-infinite tape used to decide whether $M$ halts. It has four actions, namely, left, right, write($x$) and read, similar to the Turing-machine actions used by [Lin and Levesque, 1998]. It also has a sequence fluent $h$ used to simulate the tape of a Turing machine, and an integer planning parameter $p$ that serves as the pointer to the current location of the read-write head. The goal of the planning problem is always to reach the leftmost cell of the semi-infinite tape, i.e., $p = 0$.

The basic action theory of this planning problem contains the following axioms:

- **Precondition Axiom:**
  \[
  \text{Poss}(a, s) \equiv \text{True}
  \]

- **Successor State Axioms:**
  \[
  p(\text{do}(a, s)) = y \equiv \begin{cases} 
  a = \text{left} \land p(s) \neq 0 \land y = p(s) - 1 \lor \\
  a = \text{right} \land y = p(s) + 1 \lor \\
  (a \neq \text{left} \lor p(s) = 0) \land a \neq \text{right} \land y = p(s)
  \end{cases}
  \]
  \[
  h(p(s), \text{do}(a, s)) = y \equiv \begin{cases} 
  a = \text{write}(x) \land y = x \lor \\
  a \neq \text{write}(x) \land y = h(p(s), s)
  \end{cases}
  \]

- **Sensing Result Axioms:**
  \[
  \text{SR}(\text{left}, s) = r \equiv p(s) = 0 \land r = \text{fail} \lor \\
  p(s) \neq 0 \land r = \text{ok}
  \]
  \[
  \text{SR}(\text{right}, s) = r \equiv r = \text{ok}
  \]
  \[
  \text{SR}(\text{write}(x), s) = r \equiv r = \text{ok}
  \]
  \[
  \text{SR}(\text{read}, s) = r \equiv r = h(p(s), s)
  \]
• Initial Situation Axioms:

\[
p(S_0) \geq 0
\]
\[
\forall i. h(i, S_0) = \text{blank}
\]

Notice that the sequence fluent \( h \), which simulates the semi-infinite tape of the Turing machine, is initialized to “blank” at all indexes. We assume that only five symbols may be written to the tape, which are zero and one with and without a marker respectively, and a special boundary symbol. The marker is used to remember the current location of the read-write head when simulating the execution of another Turing machine, \( e.g., M \).

To prove Theorem 8, suppose, for the sake of contradiction, that Conjecture 1 is true. I shall use this action theory and a corresponding FSA plan to simulate a Turing machine, so that finite verifiability of this FSA plan would mean a solution to the Halting Problem.

More specifically, given any Turing machine \( M \), to decide whether \( M \) will halt eventually, I design the following FSA plan for the one-dimensional-plus problem defined above.

The first part of the FSA plan copies the tape content of \( M \), bounded by the special boundary symbol, to the sequence fluent \( h \) starting from the initial index \( p(S_0) \). This can be achieved by a number of write(\( x \)) and right move actions.

After this is done, the FSA plan simulates the execution of the Turing machine by copying the current tape content stored in \( h \) to the left side of the boundary symbol, modifying some cells as necessary to reflect the one-step execution of \( M \). This process is repeated until either left senses fail (when \( p(s) = 0 \) is reached) or \( M \) halts, as shown in Figure 4.3.

In the first case, the FSA plan always goes to the final state as soon as fail is sensed by left. As a result, when the FSA plan terminates, let \( B \) be the number of boundary symbols stored in \( h \), then it means that \( M \) does not halt within \( B - 2 \) steps.

In the second case, the FSA plan always goes to an infinite loop when \( M \) halts, so the
goal of the one-dimensional-plus planning problem will never be achieved in the future.

According to this construction, if a computable bound $N$ exists for finitely verifying the FSA plan for this one-dimensional-plus planning problem, as suggested in Conjecture 1, then I would have an algorithm for deciding whether $M$ eventually halts as follows:

Execute $M$ for $N - 2$ steps. If $M$ halts, then return “halt!” Notice that $N - 2$ is the maximum number of steps the $M$ can be simulated within the bound, since this happens only if $h$ contains the boundary symbol at every cell in the range.

Otherwise, the FSA plan must terminate and achieve the goal for $p(S_0) = 0, 1, \cdots, N$, since $M$ does not halt within $N - 2$ steps. According to Conjecture 1, the FSA plan will terminate and achieve the goal for all values of $p(S_0)$, which means $M$ never halts. Return “no halt!”

Due to the arbitrariness of $M$, we have a general algorithm to solve the Halting Problem, which is not possible. As a result, Conjecture 1 is false. □

Theorem 8 shows that planning problems with sequence fluents and freely changing indexes are not finitely verifiable in general, but this leaves open whether special cases of such problems may be finitely verifiable. For example, the verifiability the striped tower problem, where stack $B$ can be represented using an integer pointer and a sequence fluent, remains an unsolved and interesting question.
4.3.2 Two-Dimensional with Unrestricted Incrementability

In Section 4.2, I have shown that finite verifiability can be maintained when there is only one increasing planning parameter, provided no sequence fluent exists in the domain. Section 4.3.1 further shows that, indeed, having both an increasing parameter and a sequence fluent will result in the loss of finite verifiability.

Given these results, a natural question is, assuming that no sequence fluents exist in the domain, can we have more than one planning parameter that can both increase and decrease independently, yet finite verifiability stays?

To be more precise, let us consider the following two-dimensional planning problem with unrestricted incrementability.

- **Precondition Axioms:**

\[
\begin{align*}
\text{Poss}(inc_1, s) & \equiv True \\
\text{Poss}(inc_2, s) & \equiv True \\
\text{Poss}(dec_1, s) & \equiv p_1(s) \neq 0 \\
\text{Poss}(dec_2, s) & \equiv p_2(s) \neq 0 \\
\text{Poss}(test_1, s) & \equiv True \\
\text{Poss}(test_2, s) & \equiv True
\end{align*}
\]

- **Successor State Axioms:**

\[
\begin{align*}
p_1(\text{do}(a, s)) = y & \equiv a = inc_1 \land y = p_1(s) + 1 \lor \\
& a = dec_1 \land y = p_1(s) - 1 \lor \\
& a \neq inc_1 \land a \neq dec_1 \land y = p_1(s)
\end{align*}
\]

\[
\begin{align*}
p_2(\text{do}(a, s)) = y & \equiv a = inc_2 \land y = p_2(s) + 1 \lor \\
& a = dec_2 \land y = p_2(s) - 1 \lor \\
& a \neq inc_2 \land a \neq dec_2 \land y = p_2(s)
\end{align*}
\]
Chapter 4. Finite Verifiability

• Sensing Result Axioms:

\[ \text{SR}(\text{inc}_1, s) = r \equiv r = \text{ok} \]
\[ \text{SR}(\text{inc}_2, s) = r \equiv r = \text{ok} \]
\[ \text{SR}(\text{dec}_1, s) = r \equiv r = \text{ok} \]
\[ \text{SR}(\text{dec}_2, s) = r \equiv r = \text{ok} \]
\[ \text{SR}(\text{test}_1, s) = r \equiv p_1(s) = 0 \land r = \text{zero} \lor p_1(s) \neq 0 \land r = \text{nonzero} \]
\[ \text{SR}(\text{test}_2, s) = r \equiv p_2(s) = 0 \land r = \text{zero} \lor p_2(s) \neq 0 \land r = \text{nonzero} \]

• Initial Situation Axioms:

\[ p_1(S_0) \geq 0 \]
\[ p_2(S_0) = 0 \]

• Goal:

\[ p_1 = 0 \land p_2 = 0 \]

Notice that the specification above is almost an instance of a two-dimensional problem with limited incrementability, except that \( p_1 \) is allowed to increment as well. Due to the similarity between the two types of problems, the following conjecture seems plausible in analogy to Theorem 7.

**Conjecture 2.** Given the two-dimensional planning problem \( \langle D, G \rangle \) with unrestricted incrementability as defined above, together with an FSA plan axiomatized by \( D_{\text{fsa}} \), there exists a computable bound \( N \) such that the following holds.

If \( D \cup D_{\text{fsa}} \cup \{ p_1(S_0) \leq N \} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s] \),

then \( D \cup D_{\text{fsa}} \models \exists s. T^*(Q_0, S_0, Q_F, s) \land G[s] \).
In this section, I shall show that unlike Theorem 7, this conjecture is unlikely to be true, since proving it would lead to a simple solution to a long-standing conjecture in number theory.

The argument is based on a reduction from the Collatz conjecture, also known as the “3N + 1 Problem.” One of its many equivalent statements is that the following procedure always terminates for any input integer \( n > 0 \):

\[
\textbf{while } n \neq 1 \\
\{ \\
\text{if } n \text{ is even} \\
\quad n := n/2 \\
\text{else} \\
\quad n := 3n + 1 \\
\}
\]

For example, if the input is \( n = 7 \), throughout the execution of this procedure, \( n \)'s value will change in the following way, and finally terminates at \( n = 1 \):

\[
7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \\
\rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1
\]

Schroeppel has shown that a two-counter machine can be used to encode the \( 3N + 1 \) program above for an arbitrary positive integer as input [Schroeppel, 1972], where a \( k \)-counter machine consists of \( k \) registers, each can be independently incremented and decremented by 1, and tested against 0. Below, I shall use this result, and show that if two dimensional planning problems with unrestricted incrementability are finitely verifiable, then the Collatz conjecture can be proved by a simple exhaustive enumeration.

\textbf{Theorem 9. If FSA plans of a two-dimensional planning problem with unrestricted incrementability are finitely verifiable, then there is a computable number \( N \) that, such that}
the Collatz conjecture is true iff it holds for the first \( N \) instances.

**Proof.** It is not difficult to see that two-counter machines can be modeled by the basic action theory for two-dimensional problems with unlimited incrementability above. As a result, consider the FSA plan in Figure 4.4.

The FSA plan starts by testing whether \( p_1 = 1 \). Since the only test available on \( p_1 \) is whether \( p_1 = 0 \), this is done by first decrementing \( p_1 \) and then testing against 0. If the initial value of \( p_1 \) is indeed 1, the FSA plan terminates in a goal state. Otherwise, it enters loop 1 as marked in the figure. In this loop, \( p_2 \) is incremented by 1 every time...
$p_1$ is decremented by 2. This essentially calculates $\lfloor \frac{p_1}{2} \rfloor$ in $p_2$. Depending on whether $p_1$ was originally odd or even, either loop 2 or loop 3 is entered. Loop 2 handles the even case. In this case, $p_2$ contains exactly $\frac{p_1}{2}$ and $p_1$ has the value 0, so all this loop does is to swap the content of $p_1$ and $p_2$ and goes back to the start of the program. If loop 3 is entered, it indicates that the original value of $p_1$ is odd, and $p_2$ now contains $\frac{p_1 - 1}{2}$. So to make $p_1$ to contain $3p_1 + 1$ of the original value, we simply need to get the value $6p_2 + 4$, which is exactly what loop 3 does.

Given this FSA plan, if the bound $N$ in Conjecture 2 and Theorem 9 exists, then one can simply verify termination of the FSA plan, and thus the Collatz conjecture, to this bound, and if the FSA plan always terminates, it would mean that it always terminates on all initial values of $p_1$, which solves the Collatz conjecture.

This analysis indicates that understanding finite verifiability of two dimensional planning problems with unrestricted increasing effects is an interesting open question, and it is at least as hard as solving the Collatz conjecture.

4.4 Concluding Remarks

This chapter works towards a method to reason about the correctness of FSA plans for planning problems that require loops. In particular, it studies a property of the planning problem called “finite verifiability,” with which plan correctness can be concluded based on its execution on finitely many cases.

The analysis in this chapter draws a rough line in the space of planning problems. On one side of the line lies the finitely verifiable problems, including problems with $k$ decreasing-only parameters and problems with exactly 2 parameters one of which may be increased. On the other side are the vast majority of the planning problems for which finite verifiability is not generally possible, such as problems with both increasing effects and sequence fluents.
Despite the apparent small size of the finitely verifiable fragment, the exact splitting line is difficult to pinpoint. For example, finite verifiability of the striped tower problem and the two-dimensional case with unrestricted incrementability is nontrivial to conclude.

Given the results in this chapter, it is worth noting that finite verifiability studies a property of the planning problems which leads to one possible method for verifying the correctness of their plans. Apart from this approach, there may well be other ways to verify plan correctness, e.g., by using semantic annotation or model checking often used in the research of program correctness. While these are also interesting directions that are yet to be explored, the discussion in this chapter is orthogonal to those directions.

Finally, throughout this chapter, it is assumed that the FSA plan to be verified is given by the user without worrying about the details how these plans are generated. In the next chapter, I shall switch the focus, and present an algorithm to automatically generate FSA plans. As we shall see, the resulting FsaPlanner can solve a large number of interesting planning-with-loops problems that go far beyond the finitely-verifiable classes, including, for example, the striped tower challenge. Of course, for the verifiable classes, FsaPlanner is powerful enough to efficiently generate FSA plans that are provably correct, as shown in [Hu and Levesque, 2010].
Chapter 5

Generating Plans with Loops

Given the usefulness of plans with loops demonstrated in Chapter 1, it is highly desirable to have an efficient mechanism that automatically generates such plans.

In this chapter, I shall present an algorithm that achieves this goal by directly searching in the space of FSA plans. The resulting planner, called FsaPlanner, demonstrates good performance on a number of benchmark problems that require loops, some of which are not solved by other existing automated planners.

In the following, I shall begin with a review of Levesque’s KPlanner [Levesque, 2005], by which the new approach is inspired. This is followed by the main algorithm of this chapter, presented as a nondeterministic procedure, which readily translates to exhaustive search in an actual implementation. With this as a starting point, I explore two directions to make the base planner efficient in practice, namely, by adding heuristics and using random restart in the search process. Reported at the end of either extension are the performance summaries of the resulting variant of the planner, showing the efficiency of FsaPlanner and the diversity of problems that it can solve.
5.1 Review of Levesque’s KPLANNER

Based on the robot program representation introduced in Chapter 3, Levesque developed Kplanner, an automated planner to generate plans with loops for a certain type of planning problems with incomplete knowledge about the initial situation and thus requiring loops [Levesque, 2005].

A challenge faced by generating plans with loops in that and other work is that the plans, e.g., robot programs, are Turing complete in sufficiently expressive dynamic domains [Lin and Levesque, 1998], which means that a sound and complete general computational method to generate them with guaranteed correctness does not exist.

In order to get around this intrinsic difficulty, Kplanner trades the strong notion of correctness for practicality. More specifically, it assumes that the need for loops in the plan comes exclusively from a single unknown and unbounded numerical property, i.e., the planning parameter introduced in Section 4.1.1. Instead of returning a plan that works provably correctly for all planning parameter values, Kplanner generates one that is guaranteed to be correct only for two selected integers, namely, a small generation bound $N_1$ and a larger test bound $N_2$. After a plan is found, its correctness may be further verified by a separate process, e.g., by applying the finite verifiability theorems in Chapter 4, if the problem fall into the right category.

Under this assumption, Kplanner searches for a plan by alternating between the following two phases.

1. The generation phase: A conditional plan that solves the problem when the planning parameter is known to have value $N_1$ is generated. This is done by an exhaustive forward search. When such a conditional plan is found, Kplanner checks if it is a possible unwinding of a robot program with loops. If so, the robot program is forwarded to the test phase; otherwise, another plan is enumerated.

2. The test phase: The robot program obtained from the generation phase is tested
with respect to a world where the planning parameter has value \( N_2 \). If it works correctly, then it is returned as the solution to the planning problem; otherwise, \textsc{Kplanner} resumes the generation phase, and a new robot program is enumerated.

\textsc{Kplanner} is implemented in Prolog. In order to express basic action theories in this language, Levesque designed the following syntax for specifying planning problems.

- \texttt{poss(A,C):} the precondition of action \( A \) is \( C \);
- \texttt{causes(A,R,F,V,C):} on performing action \( A \) and obtaining sensing result \( R \), fluent \( F \) changes its value to \( V \) if condition \( C \) is possibly true;
- \texttt{senses(A,R,F,V,C):} if fluent \( F \) has value \( V \) such that condition \( C \) is true, then performing action \( A \) will obtain sensing result \( R \);\(^1\)
- \texttt{init(F,V):} initially, fluent \( F \) may take value \( V \);
- \texttt{param_fluent(F):} \( F \) is the planning parameter;
- \texttt{init_parm(P,F,V):} in phase \( P \) (either \texttt{generate} or \texttt{test}), the planning parameter \( F \) may take initial value \( V \).

As an example of a problem specification for \textsc{Kplanner}, Figure 5.1 shows the definition of the tree chopping example introduced in Section 3.2. To solve it in \textsc{Kplanner}, the user simply needs to evoke the planner with the goal formula by typing

\[
\text{kplan(and(chops_left=0,axe=stored))}.
\]

\textsc{Kplanner} handles projection queries by regression, so the current situation is always represented by the execution history from the initial situation. Here, a history is a sequence \( \langle A_1, R_1 \rangle \cdots \langle A_n, R_n \rangle \) of performed actions \( A_i \) paired with their sensing results \( R_i \), as proposed in [De Giacomo and Levesque, 1999]. To reason about projection

\(^1\)In the original version of \textsc{Kplanner}, \texttt{senses/5} was realized by two predicates \texttt{settles/5} and \texttt{rejects/5}. Since \texttt{senses/5} is more intuitive and compact, it is adopted in later versions.
prim_action(chop,[ok]). % hit the tree with the axe
prim_action(look,[down,up]). % look if the tree is up or down
prim_action(store,[ok]). % put away the axe

prim_fluent(axe). % stored or out
prim_fluent(chops_left). % number of chops to fell the tree

poss(chop,and(axe=out,chops_left>0)).
poss(look,true).
poss(store,axe=out).

causes(store,axe,stored,true).
causes(chop,chops_left,X,X is chops_left-1).

% looking determines whether the tree is up or down
senses(look,up,chops_left,C,C>0).
senses(look,down,chops_left,C,C=0).

init(axe,out). % the axe is out and available

parm_fluent(chops_left). % chops_left is the parameter
init_parm(generate,chops_left,1). % bound for generating is 1
init_parm(test,chops_left,3). % bound for testing is 3

Figure 5.1: Specification of the treechop problem in KPLANNER syntax.

with respect to a particular history $h$, the following two inductively defined auxiliary abbreviations are often used.

1. The last situation of a history $h$ is

$$
end(h) = \begin{cases} 
S_0 & \text{if } h = \langle \rangle, \\
\text{do}(A, end(h')) & \text{if } h = h' \cdot \langle A, R \rangle,
\end{cases}
$$

where $\langle \rangle$ is the empty history and “.” denotes concatenation.

2. The formula $Sensed(h)$ is defined by

$$
Sensed(h) = \begin{cases} 
\text{TRUE} & \text{if } h = \langle \rangle, \\
Sensed(h') \land \text{sr}(A, end(h')) = R & \text{if } h = h' \cdot \langle A, R \rangle.
\end{cases}
$$
Levesque shows that KPLANNER works efficiently in a dozen domains, including treechop, arith, variegg and bintree, which I will elaborate later in the experiment sections. Moreover, although the planner only guarantees their correctness within the generation and test bounds, the robot programs generated by KPLANNER on these examples are indeed correct in general.

5.2 Basic FSAPLANNER Algorithm

The design of FSAPLANNER is motivated by two limitations of KPLANNER. For one, the unknown and unbounded property must be expressed with a single integer fluent (the planning parameter), which limits the expressiveness of the problem specification language and thus the type of problems that can be represented and solved by the planner. For the other, the separation of conditional planning and loop detection essentially forces the planner to perform two separate search operations in order to find a candidate plan, yet all conditional plans are not guaranteed to be an unwinding of a robot program, so unnecessary computation is often spent on generating some, if not many, unfruitful conditional plans.

In this chapter, I shall report the design and implementation of FSAPLANNER which tries to solve these problems.

To address the first point, instead of using an integer parameter to range over a class of planning problems, we assume that the initial situation is incompletely specified, and among all the possibilities, two special sets of examples are given to the planner, one for generation and the other for verification. We use $\mathcal{D}_g$ and $\mathcal{D}_t$ to represent the action theory for the generation and test problems, respectively. Notice that no assumption is made on how the incomplete knowledge is axiomatized, which gives more freedom to the domain designer. Indeed, the treatment with planning parameters in KPLANNER is a special case of this general setting, since $\mathcal{D}_g$ here simply corresponds to $\mathcal{D} \cup \{p(S_0) = N_1\}$.
1: FsaPlanner(\mathcal{D}_g, \mathcal{D}_t, G) \{ \\
2: \quad P_0 := \langle\{q_0, q_F\}, \{\}, \{\}, q_0, q_F\}; \\
3: \quad \textbf{while}(\text{true}) \{ \\
4: \quad \quad P := \text{generate}(\langle\rangle, q_0, P_0)[\mathcal{D}_g, G]; \\
5: \quad \quad \textbf{if} (P == \text{fail}) \\
6: \quad \quad \quad \text{return} \text{fail}; \\
7: \quad \quad \textbf{if} (\text{test}(\langle\rangle, q_0, P)[\mathcal{D}_t, G]) \\
8: \quad \quad \quad \text{return} P; \\
9: \quad \} \\
10: \} \\

Figure 5.2: Main program of FsaPlanner

in Kplanner, and \mathcal{D}_t to \mathcal{D} \cup \{p(S_0) = N_2\}, where \mathcal{D} is the full basic action theory of the planning problem without mentioning the planning parameter \( p \) in its initial situation axioms. This feature enables FsaPlanner to be compatible with Kplanner, including its regression-based reasoning module, yet offers the flexibility for multiple extensions. Later in this chapter, I will show that a representation of the basic action theory using a list of possible world states not only enables the planner to handle more general problems, but also makes it more efficient by using a progression based reasoner.

### 5.2.1 Algorithm

The high-level algorithm of FsaPlanner is very similar to that of Kplanner. Figure 5.2 shows the pseudo-code of the main program, where \( P_0 \) is the smallest FSA plan\(^2\) to initiate the search, with one initial state \( q_0 \), one final state \( q_F \), and empty \( \gamma \) and \( \delta \) functions.

The planner switches between a generation phase (lines 4–6) and a test phase (lines 7 and 8). In the generation phase, an FSA plan \( P \) is enumerated, which achieves the goal \( G \) with respect to the generation theory \( \mathcal{D}_g \). If no such plan exists, then the planner

\(^2\)Strictly speaking, \( P_\perp = \langle\{q_F\}, \{\}, \{\}, q_F, q_F\rangle \) is the minimum FSA plan, and could have been used as the starting FSA in the search. It is a solution to trivial planning problems where the goal is always true in the initial state. However, using \( P_0 \) instead of \( P_\perp \) makes the presentation simpler, and \( P_\perp \) can be treated as a special case.
returns failure; otherwise the planner enters the test phase, where the generated plan $P$ is tested with respect to the test theory $D_t$. If $P$ is valid for $D_t$, then it is returned as a solution; otherwise, the planner resumes the generation phase, and a new plan is enumerated.

The main novelty of FSAPlanner lies in the plan generation algorithm, which addresses the second limitation of KPlanner described above. Recall that KPlanner first generates a conditional plan $P'$ that works for $p(S_0) = N_1$, and then tries to wind $P'$ into a loopy plan $P$. Unfortunately, even for a compact plan $P$ containing loops, its unwinding $P'$ may be large, so it may take a very deep search to find the conditional plan. Moreover, not all conditional plans are amenable to winding, so many of the enumerated conditional plans will be rejected, and thus a huge number of search steps could be wasted.

To overcome this problem, we instead look for a plan with loops directly in the space of FSA plans by enumerating them and simulating their execution at the same time. Figure 5.3 shows the pseudo-code for the generation phase of FSAPlanner. Note that due to the existence of nondeterminism in the pseudo code, “return” statements should be understood as backtracking points: when the execution reaches a dead-end or a generated FSA plan is rejected in the test phase, the execution rolls back to the most recent nondeterministic choice point, and a new choice is attempted from there.

The intuition behind this algorithm is to start from the smallest FSA plan in the initial state, and expand it as necessary during the simulation of its execution. Expansions happen when an action association in $\gamma$ or a transition in $\delta$ is missing, in which case a nondeterministic choice for the action or target plan state is made, and the new association or transition is added to the corresponding function, introducing a new plan state if necessary.

The procedure `generate` (lines 1–21) revises the current plan $\langle Q, \gamma, \delta, q_0, q_F \rangle$ by simulating its execution from plan state $q$ and history $H$. It identifies the next action to
Chapter 5. Generating Plans with Loops

85

1: \textbf{generate}(H, q, (Q, \gamma, \delta, q_0, q_F)) \{ \\
 2: \textbf{if } (q == q_F) \{ \\
 3: \textbf{if } (D_g \cup \{\text{Sensed}(H)\}) \models G[\text{end}(H)] \\
 4: \textbf{return } (Q, \gamma, \delta, q_0, q_F); \\
 5: \textbf{else } \\
 6: \textbf{fail}; \\
 7: \} \\
 8: \textbf{else if } (\gamma(q) == \bot) \{ \\
 9: \text{nondeterministically choose action } a \text{ such that } \\
 10: (D_g \cup \{\text{Sensed}(H)\}) \models \text{Poss}(a, \text{end}(H)); \\
 11: \gamma' := \gamma \cup \{q \rightarrow a\}; \\
 12: \textbf{return } \text{tryAct}(H, q, a, (Q, \gamma', \delta, q_0, q_F)); \\
 13: \} \\
 14: \textbf{else} \\
 15: a := \gamma(q); \\
 16: \textbf{if } (D_g \cup \{\text{Sensed}(H)\}) \models \text{Poss}(a, \text{end}(H)) \\
 17: \textbf{return } \text{tryAct}(H, q, a, (Q, \gamma, \delta, q_0, q_F)); \\
 18: \textbf{else} \\
 19: \textbf{fail}; \\
 20: \} \\
 21: \}

22: \textbf{tryAct}(H, q, a, P_0) \{ \\
23: \text{let } r_1, \cdots, r_n \text{ be all possible sensing results for action } a; \\
24: \textbf{for } i = 1, \cdots, n \{ \\
25: P_i := \text{progress}(H, q, a, r_i, P_{i-1}); \\
26: \} \\
27: \textbf{return } P_n; \\
28: \}

29: \textbf{progress}(H, q, a, r, (Q, \gamma, \delta, q_0, q_F)) \{ \\
30: \textbf{if } (D_g \cup \{\text{Sensed}(H \cdot (a, r))\}) \not\models \text{FALSE} \{ \\
31: \textbf{return } (Q, \gamma, \delta, q_0, q_F); \\
32: \} \\
33: \textbf{else if } (\delta(q, r) \neq \bot) \{ \\
34: q' := \delta(q, r); \\
35: \textbf{return } \text{generate}(H \cdot (a, r), q', (Q, \gamma, \delta, q_0, q_F)); \\
36: \} \\
37: \textbf{else} \\
38: \text{nondeterministically} \\
39: \textbf{either} \{ \\
40: \text{choose } q' \in Q; \\
41: \delta' := \delta \cup \{(q, r) \rightarrow q'\}; \\
42: \textbf{return } \text{generate}(H \cdot (a, r), q', (Q, \gamma, \delta', q_0, q_F)); \\
43: \} \\
44: \textbf{or} \{ \\
45: \text{choose } q_{\text{new}} \not\in Q; \\
46: Q' := Q \cup \{q_{\text{new}}\}; \\
47: \delta' := \delta \cup \{(q, r) \rightarrow q_{\text{new}}\}; \\
48: \textbf{return } \text{generate}(H \cdot (a, r), q_{\text{new}}, (Q', \gamma, \delta', q_0, q_F)); \\
49: \} \\
50: \}

51: \}

52: \}

Figure 5.3: The plan generation algorithm
perform in the current plan state $q$. If $q$ is the final state, then the goal condition must be satisfied, in which case the current FSA plan is returned (lines 2–7). If $q$ is not final, then it tries to execute the action $\gamma(q)$ associated with $q$ (lines 14–20), nondeterministically choosing an executable one if nothing is associated yet (lines 8–13). Whenever an action prescribed by the plan is not executable, no extension of the partial FSA plan will be a valid plan, so a failure is returned and another choice will be explored at the most recent nondeterministic choice point (line 19).

To try an action, the planner must find the correct progression for each of the action’s sensing results, and accumulate the transitions, in order to obtain an FSA plan that works for all cases (lines 23–29).

Finally, to progress with respect to an action with a specific sensing result, the planner simply identifies the plan state $q' \in Q$ to transfer to when executing action $a$ in plan state $q$ obtaining sensing result $r$ (lines 31–53). If the transition $\delta(q, r) = q'$ is already in the current FSA plan, then follow this transition, and recursively call the generate procedure from there (lines 35–38). Otherwise, nondeterministically choose a target program state $q' \in Q \cup \{q_{\text{new}}\}$, add the transition to $\delta$ in the current FSA plan, and call generate from this $q'$ (lines 39–52), where $q_{\text{new}} \not\in Q$ is a new plan state. In practice, transfers to existing plan states (lines 41–45) are tried before the one to a new state, since compact plans (those with fewer plan states) are preferred. Naturally, if the sensing result is impossible to obtain, then the current plan is returned directly without further search (lines 32–34).

The algorithm in Figure 5.3 shows the essential components for the generation of plans with loops, and leaves out some details that need to be taken care of in a practical implementation for clarity in presentation. For example, to make the search space finite, a bound on the number of program states needs to be introduced so that any search branch that exceeds this bound fails immediately. Similarly, the number of search steps can be tracked and bounded in an implementation so that infinite branches (e.g., when a loop increments an integer fluent indefinitely) can be avoided. Such mechanisms can
easily be added to the base algorithm when needed.

5.2.2 Implementation and Experiments

Based on the abstract algorithm above, we implemented FsaPlanner in SWI-Prolog, the same language in which Kplanner was written, so as to compare the two planners in their common benchmark problems.

To test the effectiveness of the base version of FsaPlanner, we ran it on all of the example problems of Kplanner version 1.2,\(^3\) including:

- **airport**: A robot wants to board a plane with newspaper and a drink, which can be purchased at different locations in the airport. To board the plane, the robot must be at the correct terminal and gate. This information is unknown initially, but can be acquired with a sensing action `read_screen`.

- **arith**: Given two accumulators, both with initial value 0, two increment actions that add 1 to the respective accumulator, and a sensing action that tests whether the first accumulator is equal to an unknown input \(N\), find a plan that makes the second accumulator to have the value \(2N + 1\).

- **bars**: This is a classical puzzle with 12 identical bars, except one that has different weight from all the others. A balance scale can be used to tell the bars on which side is heavier. The task is to find the odd bar in just 3 weighings, and tell whether it is heavier or lighter than the others.

- **bintree**: Starting from the root node of a binary tree, search for a target node. Available actions include pushing down to the left or right branch, popping up from a child to its parent node, and checking whether the current node is an internal, a leaf or a target node.

\(^3\)Available online at http://www.cs.toronto.edu/~hector/swi-kplanner-1.2.tar.gz.
• **fact**: Given two registers, both with initial value 0, two increment actions that add 1 to the respective register, two multiplication actions that multiplies one register by the other, and a sensing action testing whether the first register is equal to an unknown input $N$, find a plan that makes the second register to contain the factorial of $N$.

• **fixedegg**: The goal is to get a fixed number of good eggs in the bowl from an infinite egg supply in order to make an omelet. Available actions include getting the next egg into the dish, smelling the dish to see whether its content is good or bad, dumping the dish, and moving an egg from the dish to the bowl.

• **safe**: The goal is to get a safe open by opening its door. The door only opens after a certain sequence of buttons has been pushed. This sequence is not known in advance, but is written on a piece of paper. If the paper is picked up, each digit in the sequence can be read in turn.

• **variegg**: The task is the same as in **fixedegg**, except that the number of good eggs needed is not fixed or known. An additional sensing action is provided to tell whether enough eggs have been collected in the bowl.

We compared the running time of **FsaPlanner** with **Kplanner** using all of these example problems in the same setting. This includes the assumption that the optimal depth of search is unknown, so iterative deepening is used on this parameter, and the use of the same hand-written pruning rules to speed up the search. Figure 5.4 shows the run time in seconds of both planners on the benchmark problems, where a dash “-” indicates that no plan is found after 1 hour. All experiments are performed in SWI-Prolog 32-bit version 5.8.0 under Ubuntu Linux 8.04 on an Intel Core2 3.0GHz CPU with 3.2GB memory.

In almost all problems, **FsaPlanner** performs better than **Kplanner**, sometimes even orders of magnitude faster, *e.g.* in **arith**, **fact**, **fixedegg** and **treechop**. Most
Figure 5.4: Comparison between Kplanner and FsaPlanner.

notably, Kplanner was unable to solve fixedegg problem when the number of needed eggs is greater than 6. In contrast, FsaPlanner solves 9 eggs within less than half a second! The last example, stripe, proves to be a difficult problem. As we can see, Kplanner is unable to solve this problem even with strong hints, whereas FsaPlanner solves it (with the same set of hand-written pruning rules) within minutes.

Among these benchmark problems, safe, treechop and variegg are all instances of one-dimensional planning problems, as discussed in Section 4.1.1, so their correctness is guaranteed by FsaPlanner as long as the sufficiently large test bounds are used [Hu and Levesque, 2010].

5.2.3 Discussion

The experimental results suggest that the base algorithm in FsaPlanner has led to a performance gain over Kplanner. However, there are at least three limitations in this
First, the nondeterministic choices in lines 9 and 40 of Figure 5.3 are implemented by a simple uninformed exhaustive search, which means that it is impractical for all but the simplest domains. For problems that require deep search, e.g., bintree, fact, fixedeg, and stripe, both Kplanner and FsaPlanner described so far rely heavily on hand-written pruning rules in the search. Besides, the processing order of sensing results of an action (lines 24-27) also influences the efficiency of plan generation. As a result, how to improve the search speed without human knowledge or intervention is an interesting topic to study.

Second, we have only shown the performance of FsaPlanner on Kplanner examples, which are all encoded with a planning parameter. There are also other benchmark problems in the literature of planning with loops using different representation formalisms [Winner and Veloso, 2007, Srivastava et al., 2008, Bonet et al., 2009, Pralet et al., 2010]. With the more general way to model planning problems, it is also interesting to investigate how FsaPlanner behaves on those problems.

Finally, following Kplanner’s implementation, the projection queries are computed by regression, which may become inefficient when the history becomes long, a typical scenario that happens in solving more difficult problems where deep search is necessary.

To address the latter two problems, the algorithm in Figure 5.3 can be revised to use a list of possible world states to represent the current situation, instead of a history of actions and sensing results. Although it may not be the most compact representation, its generality and flexibility enables users familiar with classical planning to conveniently specify complex planning problems by listing the possible initial states. Moreover, with this explicit representation, progression can easily be used instead of regression for all the projection queries, without worrying about the progressibility [Lin and Reiter, 1997], since each possible world is propositional and fully known. This overcomes the limitations of regression discussed above. To do so, the empty history ⟨⟩ simply corresponds to the
list of all possible states of the initial world for $S_0$, and $H \cdot \langle a, r \rangle$ can be computed by progressing the states for $H$ with respect to action $a$, and collecting all the resulting states whose sensing result matches $r$.

In order to utilize this list-of-states representation for the current situation, we need to make some syntax changes for specifying the initial states. Instead of KPLANNER’s init/2, FSAPlanner now adopts the predicate init/3, where init(W,F,V) asserts that fluent $F$ may initially take value $V$ in the world named $W$. To indicate which are the possible worlds for the generation and test phases, we use the new predicate worlds/2, where worlds(P,L) states that the possible worlds for phase $P$ are those in the list $L$. Besides, the goal condition is given in the problem specification by goal(G) instead of in the command to invoke the planner, so that we only need to type “plan” to start the planning process.

The new specification of treechop, for example, is illustrated in Figure 5.5, which says that in the generation phase, only consider the two possible worlds where chops_left is 0 and 1, respectively, and in the test phase it may be 0, 1, 2 and 3.

This more general representation will be used in the next two sections, where we try to address search efficiency. More specifically, Section 5.3 explores the use of heuristics in the base search algorithm, and Section 5.4 focuses on using random restart techniques. In both cases, the resulting planner proves to perform well on a number of existing and newly-proposed benchmark problems.

5.3 Planning via Heuristic Search

Inspired by the well-known success of heuristic search in classical planning, we designed and adapted a set of heuristics for the choice points in the plan generation algorithm in Figure 5.3, so that the more promising branches are explored first. This makes plan generation much faster than blind search. In this section, I will elaborate these heuristics.
Chapter 5. Generating Plans with Loops

5.3.1 Adding Heuristics to the Basic Algorithm

The two principles behind the design of the heuristics in FsaPLANNER are that the planner should first try the choices which

1. achieve the goal sooner, and

2. use fewer plan states in doing so.

The first point is easy to understand, since success is measured by goal achievement in the first place. The second point, using fewer program states, is proposed, because compact FSA plans are preferable, not just for representational succinctness, but also for their potential generality according to the Occam’s Razor: smaller FSA plans tend
to be more general when applied to similar instances outside the given example initial states in $D_g$ and $D_t$. Since the whole purpose of generating an FSA plan for $D_g$ and $D_t$ is that it hopefully also solves all the infinitely many other cases for the problem, this compactness and generality property is crucial. Another benefit of trying to minimize the number of plan states in the search is that the set of plan states is limited to $Q$, so the fewer states a branch consumes, the more flexibility it leaves to the other unexplored branches, and thus the more likely a plan will be found.

Following these principles, we adapted the following heuristics for selecting actions and state transitions to the basic search algorithm.

**Action Selection**

In Figure 5.3, if the search explores a program state $q$ to which no action is associated yet, we need to select the most promising action to try out (lines 8–13). To resolve the nondeterminism in line 9, given the current history $H$, all candidate actions $A$ satisfying $D_g \cup \{\text{Sensed}(H)\} \models \text{Poss}(A, \text{end}(H))$ are evaluated by simulating the execution of $A$ in $\text{end}(H)$. Since executing $A$ may lead to different sensing results, each such result $R_i$ needs to be considered individually. More specifically, we need to calculate the heuristic estimation $h(H \cdot \langle A, R_i \rangle)$ of each history $H \cdot \langle A, R_i \rangle$.

In our implementation, the uncertainty in the situation $s = \text{end}(H \cdot \langle A, R_i \rangle)$ is represented explicitly by a list of possible worlds $w_1, \ldots, w_n$. With this representation, we define $h(H \cdot \langle A, R_i \rangle) = \max_j h(w_j)$, where $h$ is a heuristic function to evaluate the goal distance for a complete world $w_i$. For this purpose, heuristic functions in classical planning can be used. The value $\max_i h(w_j)$ estimates the worst-case cost to achieve the goal on sensing $R_i$ after performing $A$. To estimate how promising action $A$ is, we use the minimum heuristic value among all its sensing as the combined heuristic value for $A$. After the estimation for all actions is done in situation $\text{end}(H)$, the planner tries the actions (line 9 of Figure 5.3) in the order of increasing heuristic values. During the simulation of
the action, it explores the possible sensing results also in increasing heuristics (line 24). This way, the actions and sensing results that seem closest to the goal are chosen first, which is likely to lead to earlier goal achievement.

Consider the tree chopping example: suppose the uncertainty of the initial situation is represented by four possible worlds \( w_0, \ldots, w_3 \), where the axe is initially out for use in all worlds and the number of chops needed to fell the tree is \( n \in w_n \). Recall that the goal is to have the tree down (\texttt{chops_needed} = 0) and the axe stored. To choose an associated action for the initial plan state, we need to consider actions \texttt{look} and \texttt{store} (\texttt{chop} is not a legal action since its precondition is not met in \( w_0 \)). If we choose \texttt{look}, we will obtain \( w_0 \) for the sensing result \texttt{up} and \( w_1, w_2, w_3 \) for \texttt{down}; if we choose \texttt{store}, we only have one sensing result \texttt{ok} with possible worlds \( w'_0, \ldots, w'_3 \), where \( w'_i \) is the same as \( w_i \) except the axe is stored. Suppose the heuristic function for these complete worlds give us the following values:

\[
\begin{align*}
    h(w_0) &= 1 & h(w'_0) &= 0 \\
    h(w_1) &= 2 & h(w'_1) &= 1 \\
    h(w_2) &= 3 & h(w'_2) &= 2 \\
    h(w_3) &= 4 & h(w'_3) &= 3,
\end{align*}
\]

then we have

\[
\begin{align*}
    h(\langle \texttt{look}, \texttt{up} \rangle) &= \max(h(w_1), h(w_2), h(w_3)) = 4 \\
    h(\langle \texttt{look}, \texttt{down} \rangle) &= \max(h(w_0)) = 1 \\
    h(\langle \texttt{store}, \texttt{ok} \rangle) &= \max(h(w'_0), h(w'_1), h(w'_2), h(w'_3)) = 3.
\end{align*}
\]

As a result, we choose to first try the \texttt{look} action, which has a heuristic value 1, in favor of \texttt{store}, which has 3. During the simulation of \texttt{look}, we will explore the sensing result \texttt{down} before \texttt{up}, due to its smaller pessimistic distance estimate.

For the function \( h \) on complete worlds, we used a variant of the delete-relaxation
heuristics in FF [Hoffmann and Nebel, 2001] adapted to our state space. Notice that classical planning uses a relational representation, whereas our planner relies on a functional encoding. As a result, we compute the relaxation by growing the possible values of each fluent, until either the values satisfying the goal is reached or the values are saturated. In the former case, the heuristic value is the length of a plan that enables the fluents to take the goal-satisfying values; in the latter, the heuristic value is $+\infty$ to indicate that the goal is not reachable.

For example, consider the evaluation of $h(w_2)$, which involves the world where 2 chops are needed to fell the tree and the axe is out. This is illustrated in the rightmost circle in Figure 5.6. If we build the relaxed planning graph from there, in the next layer, the possible values for `chops_needed` will include 1 and 2 due to the `chop` action, and `axe` will include `out` and `stored` due to `store`, as shown in the middle circle. Here, `stored` is a goal satisfying value for `axe`, but the values for `chops_needed` have not satisfied the goal yet. In the next layer, the possible values for `chops_needed` will include 0, 1 and 2, where 0 satisfies the goal, so the expansion of the planning graph stops there. From this graph, we can extract a relaxed plan with one `store` action and two `chop` actions, which gives us $h(w_2) = 3$.

**Transition Selection**

The other nondeterministic choice is which plan state to transition to for a given action and sensing result (lines 39–52). Following our second principle in heuristics design, an effective heuristic is to try $q_F$ first, then all the used plan states, and only when none of
them leads to a plan, one unused plan state in $Q$. This corresponds to reusing used plan states as much as possible, resorting to an unused state only when absolutely necessary, and thus minimizes the number of plan states to use. For choosing the transition to a used, non-initial state, an arbitrary exploration order may be used. For easy implementation, we used a set of integers for the plan states in our planner: always use “1” for representing the initial state $q_0$, “0” for the final state $q_F$, and keep track of the total number $N$ of used states. Then, whenever a transition destination $q'$ is needed, just try the states $0, 1, \cdots, N$ in ascending order, which naturally implements the ordering described above.

### 5.3.2 Experimental Results

We adapted the above-mentioned heuristics to the base version of FsaPlanner. In order to test the effectiveness of the resulting planner, in addition to the benchmark problems of Kplanner, we encoded a dozen new problems, including some found in the recent literature, which we describe briefly here.

- **cornerA** [Bonet et al., 2009]: In an unknown $N \times M$ grid world with actions to move one cell to the four compass directions, and sensing actions to test whether it is beyond the grid boundary in those directions, find a plan that always navigates the robot, initially at an unknown cell, to the north-west corner of of the grid.

- **cornerR** [Bonet et al., 2009]: In an unknown $N \times M$ grid world with actions to move one cell forward, to turn left or right, and a sensing action to tell whether the grid boundary is blocking a forward move, find a plan that always navigates the robot, initially at an unknown cell facing north, to the north-west corner of of the grid.

- **delivery** [Srivastava et al., 2008]: Transport an unknown number of packages, each labelled with its destination, from dock to its destination with a truck of
capacity one. Available actions include moving to a location, loading and unloading a package, and sensing the destination of the currently loaded package.

- **green** [Bonet et al., 2009]: In a blocks world with a single tower consisting of $N$ coloured blocks, collect a green block. Available actions include unstacking a block, dropping and collecting a block in hand. At any time, only the colour of the topmost block in the tower can be sensed.

- **gripper** [Bonet et al., 2009]: A robot with the capacity to carry up to $M$ balls can move between rooms $A$ and $B$, and load or unload a ball if one exists and the capacity permits. Initially the robot is carrying nothing in room $A$, and the goal is to transfer all the $N$ balls in room $B$ to room $A$.

- **hanoi**: There are 3 pegs, $A$, $B$ and $C$, with peg $A$ initially containing a stack of disks in ascending order from top down and $B$ and $C$ empty. The robot has a hand that can hold at most one disk. It can pick up a disk from a peg, and put down a disk onto a peg. The robot senses the failure when trying to pick up a disk from an empty peg or put a larger disk onto a smaller one. The goal is to get all of the disks onto peg C.

- **prizeA** [Bonet et al., 2009]: In the same grid world setting as cornerA, find a plan that navigates the robot, initially at the southeast corner to visit all cells of the grid for a hidden prize.

- **prizeR** [Bonet et al., 2009]: In the same grid world setting as cornerR, navigate the robot, initially facing north at the southeast corner, to visit all the cells.

- **recycle** [Srivastava et al., 2008]: There are an unknown number of recycle bins containing either paper or glass. The robot can move to the next bin, pick the content and collect as either paper or glass. The goal is to process all the bins and collect everything in the right category.


- **sort**: The goal is to sort an unknown array of numbers, with a single pointer initially pointing to the left-most element of the array. Available actions include moving the pointer to the left or right by one element, comparing or swapping the current element with the one to the left or right, sensing whether the pointer is at the left or right boundary, and sensing if the whole array is in order.

- **stripe0** [Srivastava et al., 2008]: Given a tower containing an unknown number of blue blocks on top of the same number of red blocks, arrange the blocks so that the tower is striped. Available actions include picking up a block from the tower or the table, putting down a block onto the tower or the table, sensing whether the tower or the table is empty, and sensing the colour of the block in hand. This version of the striped tower problem motivated our introduction of the more challenging version **stripe**. Notice that this version allows all blocks to be put directly onto the table first, while **stripe** is more constrained, since all the blocks must be in one of the three stacks at all times.

- **transport** [Srivastava et al., 2008]: In a “Y”-shaped transport map with \(L_1, L_2\) and \(L_3\) being the end points and \(D\) the centre. An equal number of monitors and servers are located at \(L_1\) and \(L_2\), respectively, and the goal is to move all monitors and servers to \(L_3\) by a truck of capacity one initially at \(L_1\) and a truck of capacity two initially at \(L_2\), with the constraint that only pairs of monitors and servers can be unloaded at \(L_3\).

- **trash** [Bonet et al., 2009]: A cleaning robot wants to move all trash on the floor into some trash cans. It can wander for the trash and cans when it is far from them, move to them when they are near, and grab or drop the trash when it is at their location. A sensing action tells the robot if there is still trash on the floor.

- **visual-M** [Bonet et al., 2009]: The goal is to place a marker, initially at the lower-left corner of a blocks world scene, on top of a green block. The actions are moving
the marker to the four directions, placing the marker and sensing what is currently
under the marker (table, green block, non-green block, or empty).

We tested the base planner and the heuristic variant of FsaPlanner on all the
summarized benchmarks. Table 5.1 shows the performance of both variants in the same
environment setting.

In the table, $|Q|$ is the number of plan states of the FSA plan. “Base” and “Heuristic”
indicate the base and heuristic versions of the planner, respectively. For either variant,
$N$ is the number of search steps used to find the first FSA plan, and $T$ is the CPU time in
seconds. “-” indicates that no solution is found within 900 seconds. All experiments are
performed in Ubuntu Linux 8.04 on an Intel Core2 3.0GHz CPU with 3.2GB memory.

| Domain   | $|Q|$ | Base $N$ | Base $T$ | Heuristic $N$ | Heuristic $T$ |
|----------|------|----------|----------|---------------|---------------|
| arith    | 5    | 74 0.03  | 9 0.02   |               |               |
| bintree  | 5    | 33 0.04  | 33 0.07  |               |               |
| cornerA  | 5    | 195 0.12 | 531 0.6  |               |               |
| cornerR  | 6    | 126 0.21 | 117 0.23 |               |               |
| delivery | 9    | 667 0.17 | 74 0.14  |               |               |
| factorial| 5    | 188 0.2  | 18 0.03  |               |               |
| green    | 6    | 11 0.01  | 10 0.02  |               |               |
| gripper  | 6    | 19 0.04  | 20 0.22  |               |               |
| gripper+ | 9    | 205 0.28 | 107 0.52 |               |               |
| hanoi    | 7    | 1539 1.53| 383 5.12 |               |               |
| prizeA   | -    | -        | -        |               |               |
| prizeR   | -    | -        | -        |               |               |
| recycle  | 7    | 269 0.16 | 63 0.14  |               |               |
| sort     | 7    | -        | 707 0.55 |               |               |
| stripe0  | 10   | -        | 3836 12.9|               |               |
| stripe   | 15   | -        | 1487209 680.25 |           |               |
| transport| 11   | -        | 1396 159.34 |            |               |
| trash    | 8    | 233 0.19 | 14 0.06  |               |               |
| treechop | 4    | 7 0.01   | 7 0.01   |               |               |
| variegg  | 6    | 55 0.05  | 30 0.04  |               |               |
| visual-M | 7    | -        | 9927 16.14 |              |               |

Table 5.1: Experimental results of base and heuristic FsaPlanner on benchmark problems.
As we can see from the table, the base planner with depth-first search alone can solve many of the easy problems efficiently, but for slightly more difficult problems, the search time quickly jumps from less than 1 second to prohibitively long. In contrast, heuristic search enables the planner to find a plan using much fewer search steps in most cases, and contributes to the solution of 5 unsolvable domains by depth-first search. However, the computation for heuristics results in a small overhead in the search, so for very simple problems where depth-first search and heuristic search explore comparable number of search steps, \textit{e.g., bintree, green} and \textit{treechop}, heuristic search may be slightly slower.

Appendix A contains a list of the FSA plans found by the heuristic planner. (For \texttt{prizeA} and \texttt{prizeR} that were not solved by this planner, the shown FSA plans were generated by the random restart variant discussed in the next section.)

Although the plans are generated by a finite number of generation examples, they prove indeed correct in general for all possible instances of the planning problems.

Among all the benchmark problems, the one for the striped tower example (\texttt{striped}) introduced earlier in Chapter 1 resolves our question there: a plan for this problem indeed exists. It contains two identical subplans starting from the nodes labeled “testA.” The upper copy tries to get a red block in hand and put it onto stack \textit{C}, leaving all the blue blocks on stack \textit{B}, while the lower copy puts a blue block onto \textit{C}, and all red blocks onto \textit{B}, if any. With 15 program states required in the plan, the striped tower puzzle represents one of the largest and most challenging plan to synthesize, and, to the best of our knowledge, is not solved by other existing planners.

Apart from these general observations, we noticed a few interesting phenomena in the experiments. For example, the \texttt{gripper} problem of Bonet \textit{et al.} has a very simple generalized plan with 6 plan states, which, regardless of the capacity of the robot, repeatedly carries just one ball from room B to A at a time, until all the balls are in A. We forced the robot to carry as many balls as possible in each move in a variant called \texttt{gripper+}, and a 9-state solution is found within seconds.
Another interesting problem is sort, which formulates a classic example in an algorithm course declaratively, and then our planner is able to automatically find a correct plan for it, as shown in the appendix. This, together with other programming puzzles like arith and factorial, suggests that our generalized planning techniques may be adapted to such related tasks like automatic program synthesis. As far as we can tell, none of the other existing approaches, except for Kplanner, is able to solve this type of program synthesis tasks involving unbounded integers or arrays.

Notice that our FSA plans are Moore-type controllers, while [Bonet et al., 2009] as well as [Pralet et al., 2010] use a Mealy-type controller representation. This is why our plans use more program states than reported in their result, even though the behavior of the found plans are similar. For the same reason, a direct comparison between these planners is not possible. However, in Chapter 6, we shall see that the FsaPlanner algorithm readily generalizes to other applications, including generalized planning for Mealy plans. As a result, I shall postpone the discussion and comparison of these different approaches until then.

5.4 Planning via Random Restart

Although the heuristic approach works well in many cases, the performance relies heavily on the effectiveness of the heuristics, and an imprecise prediction may lead the search to a dead-end from which recovery takes a long time.

For example, the heuristic planner is unable to solve the grid-world problems prizeA and prizeR within the 900-second time limit. Intuitively, a plan for those domains should follow a zigzag route to sweep all the cells in the grid. However, the heuristic function tends to guide the robot to move in one direction as much as possible. This, together with the observation that $5 \times 5$ is probably the smallest grid that requires this shape for the plan [Bonet et al., 2009], forces the heuristic approach to first explore the impossible
branches in the massive search space, and thus results in the prohibitively long search time.

Inspired by the success of randomized search methods for solving SAT problems [Selman et al., 1992], we explore in this section a different method for dealing with the potentially huge search space in the base algorithm, namely, using randomized search with restarts.

5.4.1 The Random-Restart Algorithm

The main idea is still to implement the base algorithm in Section 5.2.1 using exhaustive search, but whenever an open decision needs to be made in a search step, instead of following a predefined ordering of exploration, a random choice is made. Obviously, the choice may turn out to be a good one or a bad one, so in a lucky trial where the planner always makes a perfect guess for the choices, a plan can be found instantly, while in an unlucky scenario, the full search space needs to be explored before a plan is found. To remedy this, a bound for the search is set in the random exploration, and whenever the bound is exceeded, the search aborts and starts over from scratch again. The hope is, with a properly set bound, the planner can find an FSA plan efficiently.

One thing we noticed from experimenting with our base planner is that if we are to generate an FSA plan based on singleton possible worlds, for some initial states, the planner returns a correct plan almost instantly, but for others, it may take a very long time. As a result, consider the following incremental approach when a list of possible initial states is given: let the planner generate a plan based on the first initial state, then revise it so that it is also correct for the second, and repeat this process until all possible worlds are accommodated. If the sequence of initial states is given in an order such that the “good” initial states precede the “bad” ones, the planner will find a correct plan much faster than in the other orderings. As a result, it is desirable to have the list of possible worlds in a cooperative ordering. Unfortunately, it is difficult to tell a priori
how good an initial state is, so a random re-ordering can be used in the incremental
search algorithm. This constitutes another randomization point in our algorithm.

Following these guidelines, Figure 5.7 shows the adapted algorithm for the random
restart search. Notice that we represent the possible states as a list $S$ directly, instead of
implicitly by the execution history $H$.

A call to the `generate` procedure takes as the argument the list of all possible initial
states (lines 1–12). The `counter` records the number of search steps in the random
exploration, and is set to 0 each time before the random search starts (line 4). For each
trial, the list of initial states is first randomly permuted (line 5), and then the current
plan, starting from an empty one, is incrementally updated for each state in the list
(lines 6–9) by calling `tryState`. In the search process, the partial plan may turn out to be
invalid, or the search bound is exceeded. In both cases, the search fails and the body of
the DO-WHILE loops is executed again.

The function `tryState` determines the action for the current world and program state,
and `tryAct` simulates the execution of the action for the state, and determines the program
transition based on the sensing result. They are similar to the corresponding functions in
Figure 5.3, except that the nondeterministic choice points are implemented by randomly picking one of the candidates (lines 25 and 46), and the state is always fully known and
thus exactly one sensing result can be generated by any action (line 39). Notice that the
step counter is incremented before each search step (line 15), and the search continues
only if the number of search steps conducted so far has not exceeded the given bound yet (lines 16–17).

\footnote{Here, the choices for actions are made completely randomly without considering the heuristic func-
tions introduced in the previous section, but it is also an interesting idea yet to be explored to combine
heuristic search and random restart at this choice point.}
Figure 5.7: The random restart algorithm for plan generation
5.4.2 Experimental Results

We implemented the algorithm in Figure 5.7 in SWI Prolog 32-bit version 5.8.0, and ran the resulting planner on the Intel Core2 3.0GHz CPU with 3.2GB memory under Ubuntu Linux 8.04. For each benchmark problem, we ran the planner 10 times, and collected the minimum, maximum and average times in the runs, which, together with the timing statistics for the heuristic variant taken from Table 5.1 for comparison, are shown in the last four columns of Table 5.2, respectively. The number of program states in the FSA plans and the step bound are also shown in the table.

As we can see, the planner is able to solve all benchmark problems within a reasonable amount of time. This includes the prizeA and prizeR examples which require a zigzag path towards the goal and proved difficult for the heuristic approach in Section 5.3. On the other hand, when the problem’s solution just involves a straightforward progressing towards the goal, like in delivery, factorial and transport, the heuristic approach tends to offer better guidance in the search, and leads to a faster solution. For most of the problems, however, the time is comparable between the two approaches.

5.5 Concluding Remarks

This chapter is concerned with automatic and efficient generation of plans with loops. The main contribution is the FsaPlanner algorithm presented in Section 5.2.1, which models planning with loops as a simulate and search problem in the space of FSA plans. To the best of our knowledge, this is the first direct search algorithm in this area. The benefits of abstracting the whole problem as one single uniform search problem include a clearer understanding of the problem structure, and the potential to improve planning efficiency using existing search techniques. Indeed, Sections 5.3 and 5.4 explores two such

\footnote{Unlike in the heuristic search variant, we did not enforce the 900-second time bound in the experiments here, in order to obtain the average statistics. However, as we can see from the table, only some experiments for prizeR and transport exceeded 900 seconds.}
Problem | Plan Size | Step Bound | Min (s) | Max (s) | Average (s) | Heuristic (s)
---|---|---|---|---|---|---
arith | 5 | 1000 | 0.00 | 0.31 | 0.064 | 0.02
bintree | 5 | 1000 | 0.00 | 0.02 | 0.005 | 0.07
cornerA | 5 | 10000 | 0.02 | 0.28 | 0.112 | 0.6
cornerR | 6 | 15000 | 0.12 | 3.25 | 0.646 | 0.23
delivery | 9 | 10000 | 0.03 | 0.61 | 0.213 | 0.14
factorial | 5 | 50000 | 0.06 | 9.04 | 2.528 | 0.03
green | 6 | 1000 | 0.00 | 0.03 | 0.015 | 0.02
gripper | 6 | 2000 | 0.01 | 0.16 | 0.064 | 0.22
gripper+ | 9 | 50000 | 0.14 | 2.71 | 2.171 | 0.52
hanoi | 7 | 10000 | 0.03 | 1.27 | 0.294 | 5.12
prizeA | 7 | 50000 | 0.08 | 12.24 | 4.785 | -
prizeR | 11 | 500000 | 71.99 | 2460.96 | 1120.1 | -
recycle | 7 | 1000 | 0.02 | 0.2 | 0.108 | 0.14
sort | 7 | 20000 | 0.13 | 6.6 | 2.192 | 0.55
stripe | 15 | 800000 | 8.45 | 652.16 | 347.62 | 680.25
transport | 11 | 1000000 | 46.64 | 992.16 | 375.321 | 159.34
trash | 8 | 1000 | 0.05 | 0.55 | 0.258 | 0.06
treechop | 4 | 1000 | 0.00 | 0.01 | 0.002 | 0.01
variegg | 6 | 1000 | 0.00 | 0.02 | 0.007 | 0.04
visual-M | 8 | 100000 | 0.38 | 78.43 | 26.043 | 16.11

Table 5.2: Experimental results of random restart FsaPlanner.

extensions to the base planner, demonstrating enhanced performance in a number of interesting and challenging benchmark problems. Notably, some of the difficult benchmark problems, e.g., *stripe*, *sort* and *hanoi* are not solved by any other existing approach in the literature.

Given the positive results on modelling planning with loops as a direct search problem, it is interesting to ask whether the technique carries over to other controller synthesis problems, e.g., other generalized planning problems in the literature, and even more fundamentally, what their “generalized planning” is. In the next chapter, I shall apply and extend the results obtained so far in this thesis, and try to answer these questions.
Chapter 6

Generalizations and Applications

In the previous three Chapters, I have presented some theoretical and algorithmic results on planning with loops in the situation calculus framework. Throughout the discussion, a few assumptions were implicitly made, including:

1. The planning problem has a fixed dynamic domain axiomatized by the basic action theory, and the need for loops in the resulting plan comes from the incomplete knowledge about the initial world. This means that, among other things, the set of fluents and actions is fixed.

2. The goal is expressed as a condition on the final state, i.e., only reachability goals are considered.

Although the framework conveniently models a number of interesting problems, as shown in the respective chapters, the assumptions make it insufficient to study other problems that go beyond those restrictions.

As an example against the first assumption, consider the following problem in Figure 6.1(a), which is a simplified version of the problem proposed by [Bonet et al., 2009]:

A robot is initially at the leftmost cell A of some linear grid, can move left and right within the cells, and can observe whether it is at the rightmost cell
B. The goal is to arrive at B.

A simple solution is shown in Figure 6.1(b), which says “repeatedly move right, until atB is observed.” Bonet et al. model this problem by introducing one proposition for each cell. This means that for grids of different sizes, the number of propositions will be different, yet they intuitively belong to the same generalized planning problem. Due to the heterogeneity of the domains, in particular, the variable number of propositions (fluents) in the domain, it cannot easily be characterized using Definition 1, although it is possible to reformulate a variant in terms of an unknown integer under that definition.

As for the second assumption, consider many of the maintenance tasks where the goal is not (only) to make a certain condition to finally become true, but instead (also) to maintain a property for some non-terminating task, e.g., a home service robot that keeps the room clean and delivers coffee upon request in its life time. This cannot be modeled as a simple situation-suppressed formula in the situation calculus as in Definition 1, but instead needs a more general form of goal representation, yet plans with loops can often be used to solve such problems.

This raises the question as to what planning is in this general setting, and how we solve these generalized planning problems. In this chapter, I shall go beyond the assumptions made in the previous chapters, and extend the results to generalized planning where less is assumed on the planning problems and their plans.

In Section 6.1, I shall give a definition of this type of planning as synthesizing a general plan that works for multiple environments, and show that not only existing
forms of planning (standard planning and planning with loops) can be formalized in this new framework, but also the theoretical results on them carry over.

Next, in Section 6.2, I will show that the plan generation algorithm in Chapter 5 can be adapted to the general setting too, and show its application in some recently proposed generalized planning tasks, including controller synthesis [Bonet et al., 2009], service composition [Calvanese et al., 2008] and planning programs [De Giacomo et al., 2010].

6.1 A Unifying Definition of Generalized Planning

Intuitively, the task of generalized planning is to synthesize a program-like plan that works for multiple, possibly infinitely many, cases. To formalize this intuition and define what generalized planning is precisely, let us start with identifying the two main factors of planning, namely, the agent that executes the plan, and the environment in which the agent’s plan is executed. Usually, these factors, together with the goal, are combined in a so-called planning problem, but to better understand their roles and interactions, let us keep them separated for now.

**Definition 16.** An agent $A$ is a tuple $A = \langle Acts, Obs \rangle$, where

- $Acts$ is a set of actions the agent can perform, and

- $Obs$ is a set of observations the agent can make.

Given an agent, its behavior is determined by its plan, which prescribes the action to perform given the agent’s knowledge about the environment so far.

**Definition 17.** The plan for an agent $A = \langle Acts, Obs \rangle$ is a prefix-closed partial function $p : Obs^* \rightarrow Acts \cup \{\text{stop}\}$, where $\text{stop}$ stands for plan termination. A partial function is prefix-closed if $p(o_1, \ldots, o_i)$ is defined for all $i < n$ whenever $p(o_1, \ldots, o_n)$ is defined.

This is a very general notion of plan that completely abstracts from syntactic or structural characterization of a plan representation, which is also often used in auto-
mated process synthesis \cite{PnueliRosner1989} as well as in POMDP-based planning \cite{Ghallab2004}.

For different types of planning, we usually assume the plans have a certain restricted form, \textit{e.g.}, a sequence of actions in classical planning and a finite-state controller in planning with loops, but as a definition, we do not make such assumptions and leave it as general as possible.

To see how this definition accommodates finite-state plans, we say that plan $p : \text{Obs}^* \rightarrow \text{Acts} \cup \{\text{stop}\}$ has a state form, if there is a set $Q$ of plan states $\{q_0, q_1, \cdots\}$, a transition function $\sigma : Q \times \text{Obs} \rightarrow Q$ and an action function $\alpha : Q \times \text{Obs} \rightarrow \text{Acts}$, such that

\[
p(\text{obs}(s_0), \cdots, \text{obs}(s_n)) = \alpha(q(\text{obs}(s_0), \cdots, \text{obs}(s_n)), \text{obs}(s_n))
\]

\[
q(\text{obs}(s_0), \cdots, \text{obs}(s_n)) = \begin{cases} 
q_0 & \text{if } n = 0 \\
\sigma(q(\text{obs}(s_0), \cdots, \text{obs}(s_{n-1})), \text{obs}(s_n)) & \text{if } n > 0 
\end{cases}
\]

By noting that we can use one plan state for each possible observation history, it is immediate that every generalized plan has a state form.

When we restrict the set $Q$ to be finite, we get the notable case of finite-state plans, which is exactly the form of generalized plan most used in the literature. Since the finite-state plan is uniquely identified by the transition function $\sigma$ and action function $\alpha$, such a plan can be represented as a Mealy controller like the one in Figure 6.1(b). Intuitively, the nodes in the graphical plan representation can be seen as the plan states, and the directed edges and their labels represent the $\sigma$ and $\alpha$ functions, respectively.

Notice that in order to define a plan, we only need the specification of the agent. In particular, we do not need any knowledge about the environment the agent acts in. This makes it possible for the plan of an agent to be executed in multiple environments, which we define next.
**Definition 18.** An environment is a tuple $E = \langle \text{Events}, S, s_0, \Delta \rangle$, where

- $\text{Events}$ is the set of all events in the environment;
- $S$ is the internal state space of the environment;
- $s_0 \in S$ is its initial (internal) state;
- $\Delta \subseteq S \times \text{Events} \times S$ is the transition relation.

The evolution of an environment $E$ is characterized by a trace, which is a (finite or infinite) sequence of states that originates from the initial state $s_0$ and caused by the happening of events in the environment.

**Definition 19.** A trace on $E$ is a partial function $\tau : \mathbb{N} \rightarrow S$ satisfying

- $\tau(0) = s_0$;
- if $\tau(n)$ is defined, then for all $0 < i \leq n$, $\langle \tau(i - 1), e_i, \tau(i) \rangle \in \Delta$ for some event $e_i \in \text{Events}$.

We call a trace $\tau$ finite, if for some $n \in \mathbb{N}$, $\tau(n)$ is defined but not $\tau(n + 1)$. In this case, we sometimes denote $\tau$ as the explicit sequence $s_0 s_1 \cdots s_n$ where $s_i = \tau(i)$, and define $\text{last}(\tau) = s_n$ as the last state of the trace.

Among all the possible evolutions of the environment, some are the desired behavior that the agent is required to achieve in a planning task, whereas others are undesirable that it should avoid. Thus the traces can be used to formulate the goal of a planning problem.

**Definition 20.** A goal for an environment $E$ is the set $\text{Tr}$ of desired traces in $E$.

Like for plans, this definition is also general without restricting the actual representation of the goals. For example, classical reachability goals can be considered as a set $\text{Tr}$ of finite traces such that $\text{last}(\tau)$ satisfies the goal condition for all $\tau \in$
Tr. Since traces can also be infinite, i.e., $\tau(n)$ is defined for all $n \in \mathbb{N}$, this definition is general enough to also capture other types of goals in the planning literature, including temporally-extended and long-running ones [Bacchus and Kabanza, 1998, De Giacomo and Vardi, 1999, Bertoli et al., 2003, Edelkamp, 2006].

Note that in order to specify goals, only knowledge about the environment is needed, and nothing on the agent acting in the environment is assumed.

Finally, to characterize the execution of an agent’s plan in an environment, we need to know how the agent observations are related to the environment states, and the agent actions to the events happening in the environment. In particular, we need the following two connecting functions between the agent and the environment.

1. An observation function $\text{obs} : S \to \text{Obs}$, which determines how much of the environment the agent can observe for the purpose of plan execution, i.e., when selecting an action to perform, the states $s_1$ and $s_2$ cannot be distinguished by the agent if $\text{obs}(s_1) = \text{obs}(s_2)$;

2. An execution function $\text{exec} : \text{Acts} \to \text{Events}$, which determines the events in the environment that the agent causes by doing its actions. This function enables the separation between what the agent can do and what changes the environment may have.

Given the observation and execution functions, we can determine the execution of an agent $A$’s plan $p$ in the environment $E$.

**Definition 21.** A plan $p$ for agent $A$ in environment $E$ with observation function $\text{obs}$ and execution function $\text{exec}$ is legal if for all $n \geq 0$, states $s_1, \ldots, s_n \in S$ and actions $a_1, \ldots, a_n \in \text{Acts}$,

$$
\text{if } a_i = p(\text{obs}(s_0), \ldots, \text{obs}(s_{i-1})), \text{ and } (s_{i-1}, \text{exec}(a_i), s_i) \in \Delta \text{ for all } 0 < i \leq n,
$$

then one of the following is true for $a = p(\text{obs}(s_0), \ldots, \text{obs}(s_n))$
• \( a = \text{stop} \),

• there exists \( s \in S \) such that \( \langle s_n, \text{exec}(a), s \rangle \in \Delta \).

This definition essentially says that for a plan to be legal, it must ensure that either the it can legally stop or the action it prescribes is possible at any point in its execution.

When a plan is legal, we can talk about its runs.

**Definition 22.** A trace \( \tau \) is a run of plan \( p \) for an agent \( A \) in environment \( E \) with observation function \( \text{obs} \) and execution function \( \text{exec} \), if there are actions \( a_1, a_2, \cdots \in \text{Acts} \) satisfying

- \( a_i = p(\text{obs}(\tau(0)), \ldots, \text{obs}(\tau(i-1))) \), and

- \( \langle \tau(i-1), \text{exec}(a_i), \tau(i) \rangle \in \Delta \) if \( a_i \neq \text{stop} \) and \( \tau(i) = \bot \) if \( a_i = \text{stop} \).

Although the plan \( p \) is deterministic, due to the nondeterminism of the environment, its execution may lead to multiple traces. We use \( \text{trace}(p) \) to denote the set of all runs of plan \( p \).

When a plan \( p \) has a finite state form, its execution can also be characterized by a history of pairs of plan states and world states.

**Definition 23.** Given an agent \( A \) in environment \( E = \langle \text{Events, } S, s_0, \Delta \rangle \) with observation function \( \text{obs} \) and execution function \( \text{exec} \), and a finite-state plan \( p \) with states \( Q \), transition function \( \sigma \) and action function \( \alpha \), an execution history is the sequence

\[
\langle q_0, s_0 \rangle \cdot \langle q_1, s_1 \rangle \cdot \cdots \cdot \langle q_{n-1}, s_{n-1} \rangle \cdot \langle q_n, s_n \rangle,
\]

where

- \( q_0 \in Q \) is the initial controller state,

- \( q_{i+1} = \sigma(q_i, \text{obs}(s_i)) \), and
• \( (s_i, \text{exec}(\alpha(q_i, \text{obs}(s_i))), s_{i+1}) \in \Delta \) for all \( 0 \leq i < n \).

Each \( (q_i, s_i) \) is called a configuration in the history.

With this definition of execution history, we can get an alternative definition to Definition 22 for the run of a finite-state plan: A trace \( \tau \) is a run of a finite-state plan \( p \) if for all \( n \) such that \( \tau(n) \) is defined, there are \( q_0, q_1, \cdots, q_n \in Q \) such that \( (q_0, \tau(0)) \cdot (q_1, \tau(1)) \cdots (q_n, \tau(n)) \) is an execution history of \( p \), and if \( \tau(n + 1) \) is undefined, then \( \alpha(q_n, \text{obs}(\tau(n))) = \text{stop} \).

For easier understanding, I will sometimes include the actions between the configurations in an execution history. For example,

\[
\langle q_0, s_0 \rangle \xrightarrow{a_1} \langle q_1, s_1 \rangle \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} \langle q_{n-1}, s_{n-1} \rangle \xrightarrow{a_n} \langle q_n, s_n \rangle,
\]

where \( a_i = \alpha(q_{i-1}, \text{obs}(s_{i-1})) \). When the intermediate configurations are irrelevant, I also abbreviate the execution history as \( \langle q_0, s_0 \rangle \xrightarrow{a_1, \cdots, a_n} \langle q_n, s_n \rangle \) or simply \( \langle q_0, s_0 \rangle \star \rightarrow \langle q_n, s_n \rangle \).

With the definitions above, we are now ready to define planning by merging the agent \( A = \langle \text{Acts}, \text{Obs} \rangle \), the environment \( E = \langle \text{Events}, S, s_0, \Delta \rangle \) and the desired traces \( \text{Tr} \) using the connecting functions \( \text{obs} \) and \( \text{exec} \).

**Definition 24.** A basic planning problem is a tuple

\[
P = \langle \text{Acts}, \text{Obs}, \text{Events}, S, s_0, \Delta, \text{Tr}, \text{obs}, \text{exec} \rangle,
\]

where \( \langle \text{Events}, S, s_0, \Delta \rangle \) is an environment, \( \langle \text{Acts}, \text{Obs} \rangle \) is an agent, \( \text{obs} \) is an observation function, \( \text{exec} \) is an execution function, and \( \text{Tr} \) is the set of goal-satisfying traces. A solution to a basic planning problem \( P \) is a plan \( p \) such that \( p \) is legal and \( \text{trace}(p) \subseteq \text{Tr} \).

If a set of basic planning problems share the same agent, i.e., \( \text{Acts} \) and \( \text{Obs} \) are kept fixed, then it may be possible to have a single plan that achieves the goals for all of the
problems. This is precisely the idea of generalized planning introduced at the beginning of this section.

**Definition 25.** A generalized planning problem $\mathcal{P} = \{P_1, P_2, \ldots\}$ is a (finite or infinite) set of basic planning problems

$$P_i = \langle Acts, Obs, Events_i, S_i, s_{i0}, \Delta_i, Tr_i, obs_i, exec_i \rangle$$

which share the same $Acts$ and $Obs$.

A plan $p$ is a solution to a generalized planning problem $\mathcal{P}$, if $p$ is a solution to every $P_i \in \mathcal{P}$.

Intuitively, we require that the plan $p$ for a fixed agent $A = \langle Acts, Obs \rangle$ always generate goal-satisfying traces in $Tr_i$ on all of the environments $E_i$, i.e., $\text{trace}(p) \subseteq Tr_i$ for all environments $E_i$. Notice again that no restriction exists on the form of $p$ in these definitions, although we usually assume that it is finite-state in practice.

With the definitions above, we can now formalize the grid example in Figure 6.1.

**Example 1.** Following [Bonet et al., 2009], we can use one proposition for each cell in the grid, and define $\mathcal{P} = \{P_1, P_2, \cdots\}$ with

$$P_i = \langle Acts, Obs, Events_i, S_i, s_{i0}, \Delta_i, Tr_i, obs_i, exec_i \rangle,$$

where

- $Acts = Events_i = \{\text{left}, \text{right}\}$, $exec_i$ is identity,
- $Obs = \{\text{at}B, \text{at}B^c\}$,
- $S_i = \{p_0, \overline{p_0}\} \times \{p_1, \overline{p_1}\} \times \cdots \times \{p_i, \overline{p_i}\}$,
- $s_{i0} = \langle p_0, \overline{p_1}, \cdots, \overline{p_{i-1}}, p_i \rangle$, 
• $\Delta_i(\langle \overline{p_0}, \ldots, \overline{p_{l-1}}, p_l, \ldots, \overline{p_i} \rangle, \text{left}, \langle \overline{p_0}, \ldots, p_{l-1}, \overline{p_l}, \ldots, \overline{p_i} \rangle)$, and
$\Delta_i(\langle \overline{p_0}, \ldots, p_{l-1}, \overline{p_l}, \ldots, \overline{p_i} \rangle, \text{right}, \langle \overline{p_0}, \ldots, p_l-1, \overline{p_l}, \ldots, \overline{p_i} \rangle)$
for all $l = 1, \ldots, i$,

• $Tr_i = \{ \tau \mid \text{last}(\tau) = \langle \overline{p_0}, \overline{p_l}, \ldots, \overline{p_{l-1}}, p_i \rangle \}$,

• $obs_i(s) = \begin{cases} 
  atB & \text{if } s = \langle \overline{p_0}, \overline{p_l}, \ldots, \overline{p_{l-1}}, p_i \rangle, \\
  \overline{atB} & \text{otherwise.}
\end{cases}$

Notice that this formalization allows for variable number of fluents for grids of different sizes. Alternatively, we can appeal to an integer planning parameter to denote the current location like in Chapter 4, and get $P = \{ P_1, P_2, \cdots \}$ with

$$P_i = \langle Acts, Obs, Events_i, S_i, s_{i0}, \Delta_i, Tr_i, obs_i, exec_i \rangle,$$

where

• $Acts = Events_i = \{ \text{left, right} \}$, $exec_i$ is identity,

• $Obs = \{ atB, \overline{atB} \}$,

• $S_i = \{ 0, 1, \ldots, i \}$,

• $s_{i0} = i$,

• $\Delta_i(n-1, \text{left}, n)$, and $\Delta_i(n, \text{right}, n-1)$ for all $n = 1, \ldots, i$,

• $Tr_i = \{ \tau \mid \text{last}(\tau) = 0 \}$,

• $obs_i(s) = \begin{cases} 
  atB & \text{if } s = 0, \\
  \overline{atB} & \text{otherwise.}
\end{cases}$


6.1.1 Relationship to Existing Forms of Planning

This definition of generalized planning is able to accommodate most existing forms of planning in the literature, including standard planning, planning with temporally-extended goals, and planning with loops.

**Standard Planning**

It is not hard to see that standard forms of planning studied in the literature [Ghallab et al., 2004], including classical, conformant and conditional planning, are all special cases of generalized planning with reachability goals.

- **Classical Planning**
  
  Classical planning is generalized planning for singleton deterministic environment sets. Indeed, classical planning can be formulated as a generalized planning problem \( P = \{P\} \), where

  \[
  P = \langle Acts, Obs, Events, S, s_0, \Delta, Tr, obs, exec \rangle.
  \]

  \( S \) here is the finite state space, and the transition relation \( \Delta \) is deterministic, \( i.e., \) for all \( s, s', s'' \) and \( e \), if \( \Delta(s,e,s') \) and \( \Delta(s,e,s') \), then \( s' = s'' \). \( Acts = Events \) is the finite set of actions with \( exec \) being the identity function. Since no observability exists in classical planning, \( Obs = \{nil\} \), and \( obs(s) = nil \) for all \( s \in S \). For this reason, plans for classical planning problems are unable to differentiate the history of observations except for its size. As a result, the plan can simply be represented as a sequence of actions.

- **Conditional Planning**
  
  Conditional planning with deterministic actions and unknown initial states can be
formalized as a generalized planning problem $\mathcal{P} = \{P_1, \cdots, P_N\}$, where

$$P_i = \langle Acts, Obs, Events, S, s_{i0}, \Delta, Tr, obs, exec \rangle$$

are identical except for the initial state $s_{i0}$. In this case, $Acts = Events$ are the ground actions in the planning problem, and $exec$ is the identity function. Notice that $N$ can be at most the size of $S$, which happens when we know nothing about the initial state, so all the states in $S$ are possible initial states.

- **Conformant Planning**

Conformant planning can be considered as conditional planning with no observability, i.e., $Obs = \{nil\}$ and $obs(s) = nil$ for all $s \in S$. Similar to classical planning, this forces the resulting plan to have a sequential representation.

### Planning with Temporally-Extended Goals

Much work on planning with temporally-extended goals in the literature deals with deterministic, fully known domains with no observability, just like classical planning, except that the goal is to satisfy a linear temporal logic (LTL) formula [Baier and McIlraith, 2006]. Since the semantics of all LTL modal operators are defined over traces of states, such temporally-extended goals fall into our definition naturally. As a result, this type of planning is also a special case of generalized planning for singleton environment sets $\mathcal{P} = \{P\}$, where

$$P = \langle Acts, Obs, Events, S, s_0, \Delta, Tr, obs, exec \rangle.$$ 

This is exactly the form for classical planning except for the assumptions on $Tr$.

### Planning with Loops

Next I move on to planning with loops, and study the relationship between this definition of generalized planning and three existing approaches to planning with loops, including
Chapter 6. Generalizations and Applications

[Bonet et al., 2009] and [Srivastava et al., 2011], and our situation calculus-based work presented in the previous chapters. All these approaches aim at finding or characterizing generalized plans with a finite-state representation to solve classes of deterministic problems containing possibly infinitely many instances. As we shall see, although they adopt different formalisms, they all fit into Definition 25.

For the situation calculus-based formalism in Section 3.2, recall that a planning problem there is defined as a basic action theory $D$ paired with a situation-suppressed goal formula $G$. It can be understood as a generalized planning problem $P = \{P_i, \cdots\}$, with

$$P_i = \langle \text{Acts, Obs, Events, } S, s_{i0}, \Delta, \text{Tr, obs, exec} \rangle.$$  

Like in standard and temporally-extended planning, all $P_i$ share the same elements except for the initial state $s_{i0}$. Among them, $\text{Acts}$ and $\text{Events}$ are the set of all ground actions in $D$, with $\text{exec}$ being the identity function, as usual. $S$ is the state space, which we define as the set of all possible world states modeled by the basic action theory, i.e., each state $s \in S$ is a mapping from all ground fluents to their respective values. For notational convenience, let us assume that given any situation object $t$ in a model $M$ of the basic action theory, the function $\zeta_M$ gives us the corresponding state, i.e., $f^M(t) = o$ iff $\zeta_M[t](f) = o$ for all fluent $f$ and domain object $o$. Then $\Delta = \{\langle \zeta_M(t), a, \zeta_M(t') \rangle \mid t' = \text{do}^M(a^M, t) \text{ and } \text{Poss}^M(a^M, t) = \text{TRUE}\}$. $\text{Obs}$ is the set of all sensing results, with $\text{obs}$ mapping from the states to these sensing results.$^1$ $\text{Tr}$ is the set of traces where the goal $G$ is satisfied in the final state, i.e., $\text{Tr} = \{\tau \mid \text{last}(\tau) \models G\}$ where $s \models \phi$ means $M, \mu_{s(f)}^x \models \phi^f_x$ for all models $M$ of $D$ and variable assignment $\mu$ and $x$ is a variable not mentioned in $\phi$. The only element that differs across the basic planning problems $P_i$ is

---

$^1$Strictly speaking, this translation cannot model the fact that sensing actions are not always executable and sensing results are obtained only after the corresponding sensing actions are executed. This can be remedied by adding a “last-action” fluent before the translation in order to remember the previously executed action in the states. Not being a key issue here, I will not elaborate the details of the exact translation.
the initial state $s_{i0}$, each of which captures the initial situation in one model $\mathcal{M}_i$ the basic action theory $\mathcal{D}$ by $s_{i0} = \zeta_{\mathcal{M}_i}(s_0^{\mathcal{M}_i})$. The second formalization of the linear grid problem in Example 1 shows a generalized planning problem that can be obtained by following this translation.

Another interesting case is the controller synthesis problem of [Bonet et al., 2009]. Although they obtain a finite-state controller based on one single propositional planning problem, they implicitly expect the controller to work for other “similar” problems too, but this notion of generalization across problems is not formally defined in their work. Our effort on giving a precise formalization of generalized planning was largely motivated by closing this gap. With the definitions in this Chapter, we can now understand their planning task as finding a controller that solves a generalized planning problem $\mathcal{P} = \{P_i, \ldots\}$ where

$$P_i = \langle \text{Acts}, \text{Obs}, \text{Events}, S_i, s_{i0}, \Delta_i, Tr_i, obs_i, \text{exec} \rangle$$

share only Acts, Obs, Events, exec, and all the other elements can be completely different. Here, we require Acts = Events and exec be the identity function, since all the set of actions are all the same in the “similar” problems.

A similar characterization can be given to [Srivastava et al., 2011] which learns a generalized plan for a relational PDDL domain from example solutions to selected problems. Due to varying number of objects, the problem instances in the domain may have a different number ground actions, but the plan may only have finitely many fixed actions. This distinguishes their planning problems from those of [Bonet et al., 2009], where a finite number of ground actions is assumed. For this reason, “abstract actions” are used in the plan in [Srivastava et al., 2011], i.e., actions whose arguments are not concrete objects in the domain, but instead what roles those objects must be in. At execution time, any tuple of objects that meet the roles of the abstract action can be used to instantiate the concrete action to perform. To model their planning problems, Acts is the set of all abstract actions, which is shared across all problems, and Events are the
actual concrete actions that take place in the environment. An abstract action may be mapped to different concrete events in different instances, and exec captures the actual mapping. As a result, the generalized planning problem for [Srivastava et al., 2011] is \( \mathcal{P} = \{P_i, \ldots\} \) where

\[
P_i = \langle Acts, Obs, Events_i, S, s_{i0}, \Delta_i, Tr_i, obs_i, exec_i \rangle
\]

share only the abstract actions Acts and observations Obs.

This confirms our claim that our definition of generalized planning is indeed general enough to capture many, if not most, existing forms of planning. Next, I shall show that the new framework not only provides a framework for representing generalized planning problems, but also helps to bring more insight into the structure of the planning problems themselves. More specifically, I shall show that we can reconstruct and extend much of the finite verifiability result of one-dimension planning problems in Section 4.1.1 by using this new framework.

### 6.1.2 One-Dimensional Planning Problems Revisited

For a cleaner presentation, I will adopt a slightly simplified version of the one-dimensional planning problems, namely, I assume for now that all the finite fluents are binary, and there is no sequence function in the domain. The results can be extended to deal with the original version, but it will require further elaboration.

**Definition 26.** A problem \( \mathcal{P} = \{P_0, P_1, \ldots\} \) is one-dimensional simplified, or 1ds, of size \( m \), if there is \( \vec{b}_0 \in \langle \text{TRUE}, \text{FALSE} \rangle^m \) such that every \( P_i \in \mathcal{P} \) is of the form

\[
\langle Acts, Obs, Events_i, S_i, s_{i0}, \Delta_i, Tr_i, obs_i, exec_i \rangle,
\]

where
1. \( S_i = \{ \langle n, \vec{b} \rangle \mid n \in \{0, \cdots, i\}, \vec{b} \in \{\text{True, False}\}^m \} \);

2. \( s_0 = \langle i, \vec{b}_0 \rangle \);

3. if \( \Delta_i(\langle n, \vec{b} \rangle, e, \langle n - d, \vec{b}' \rangle) \) for \( n > 0 \), then \( d \in \{0,1\} \); furthermore, for all \( n' > 0 \), \( \Delta_i(\langle n', \vec{b} \rangle, e, \langle n' - d, \vec{b}' \rangle) \);

4. \( Tr_i = \{ \tau \mid \text{last}(\tau) \in \{0\} \times \{\text{True, False}\} \} \);

5. \( \text{obs}_i(\langle n_1, \vec{b} \rangle) = \text{obs}_i(\langle n_2, \vec{b} \rangle) \) for all \( n_1, n_2 > 0 \) and \( \vec{b} \in \{\text{True, False}\}^m \).

For any state \( \langle n, \vec{b} \rangle \) in the state space \( S \), \( n \) is the value of the planning parameter denoting the number of unprocessed entities, and \( \vec{b} = \langle b_1, \cdots, b_m \rangle \) are the values of all \( m \) finite (boolean) fluents. Condition 5 above requires that states only differing in their non-zero integer part are observationally indistinguishable; Condition 3 enforces that the objects can only be processed one-by-one and independently; Condition 2 says that the initial states are similar except for the number of objects; finally, Condition 4 indicates that the goal is achieved only when all objects are processed with certain properties satisfied.

The key result for one-dimensional problems was Theorem 4. With the new definition, we have the following analogous, but notationally much simpler theorem.

**Theorem 10.** Given a 1ds problem \( \mathcal{P} = \{P_0, P_1, \cdots\} \) of size \( m \), and a finite-state plan \( p \) with \( l \) program states, if \( p \) is a plan for \( \{P_0, P_1, \cdots, P_N\} \), where \( N = l \cdot 2^m \), then \( p \) is a plan for \( \mathcal{P} \).

**Proof.** The proof of this theorem is very similar to that of Theorem 4, but without the model-theoretic reasoning in the situation calculus.

For the sake of contradiction, assume that \( p \) is not a plan for \( \mathcal{P} \), and in particular, \( p \) is not a plan for \( P_{N'}, \in \mathcal{P} \) but solves \( P_i \) for all \( i < N' \). Obviously, \( N' > N \).
Consider the execution of $p$ on $P_N$: it must be of the form

$$\langle N, \vec{b}_0 \rangle \xrightarrow{\sigma} \langle 1, \vec{b} \rangle \xrightarrow{a} \langle 0, \vec{b}' \rangle \xrightarrow{*} \text{Success},$$

where $\sigma$ is a sequence of actions, and $a$ is a single action that decreases the planning parameter from 1 to 0.

Due to conditions 3 and 5 in Definition 26, the execution of $p$ on $P_{N'}$ must follow

$$\langle N', \vec{b}_0 \rangle \xrightarrow{\sigma} \langle N' - (N - 1), \vec{b} \rangle \xrightarrow{a} \langle N' - N, \vec{b}' \rangle \xrightarrow{*} \text{Failure}.$$

Due to condition 3, each action can decrease the integer parameter by at most 1, and the action sequence $\sigma \cdot a$ decreases it from $N'$ to $N' - N$, so there are at least $N + 1 = l \cdot 2^m + 1$ states with distinct integer elements. Among all of these states, there are at least $l + 1$ states that have identical Boolean values, and for at least two of them $p$ is in identical program states, due to the pigeon hole principle, i.e., there exist action sequences $\alpha$, $\beta$, and $\gamma$ with $\alpha \cdot \beta \cdot \gamma = \sigma \cdot a$, such that

$$\langle N', \vec{b}_0 \rangle \xrightarrow{\alpha} \langle n_1, \vec{b}' \rangle \xrightarrow{\beta} \langle n_2, \vec{b}' \rangle \xrightarrow{\gamma} \langle N' - N, \vec{b}' \rangle \xrightarrow{*} \text{Failure},$$

where $n_1 > n_2$, and $p$ is at the same program state $q^*$ after the action sequences $\alpha$ and $\alpha \cdot \beta$.

Now consider the execution of $p$ on $P_{N' - (n_1 - n_2)}$: due to conditions 3 and 5 in Definition 26, it must be

$$\langle N' - (n_1 - n_2), \vec{b}_0 \rangle \xrightarrow{\alpha} \langle n_2, \vec{b}' \rangle \xrightarrow{\gamma} \langle N' - N, \vec{b}' \rangle \xrightarrow{*} \text{Failure}.$$

This contradicts the assumption that $p$ solves $P_i$ for all $i < N'$.

As a result, $p$ must be a plan for $P$. \qed
Figure 6.2: Plan construction for a 1ds generalized planning problem from a plan for its finite subset.

Like Theorem 4, this theorem assumes that the plan $p$ is given, and the proof uses state repetition due to limited possible configurations, similar in essence to the pumping lemma in automata theory [Hopcroft et al., 2007]. Interestingly, this idea can be further used to show that 1ds problems can be not only finitely verified, but also finitely solved.

**Theorem 11.** A 1ds generalized problem $\mathcal{P} = \{P_0, P_1, \cdots\}$ of size $m$ has a plan $p$, if and only if the finite set $\mathcal{P}' = \{P_0, P_1, \cdots, P_N\}$ has a plan $p'$ where $N = 2^m$.

**Proof.** The “only if” direction is trivial, so we focus on the “if” direction.

Suppose we are given a plan $p'$ that solves $\mathcal{P}'$. Without loss of generality, we can assume that $p'$ is a tree-like finite-state plan [Hu and De Giacomo, 2011a], as shown in Figure 6.2(a). Now consider the execution of $p'$ on the basic problem $P_N$: if $p'$ decreases $n$ from $N$ to 0, then there must be at least $N + 1$ states with different integers in the run, but there are at most $N = 2^m$ boolean combinations, so at least two states in the run have identical boolean states. Let them be $\langle N - u, \vec{b}^r \rangle$ and $\langle N - u - v, \vec{b}^r \rangle$, respectively, and the corresponding plan states be $q_u$ and $q_v$, as shown in the figure by the bold dots.

Then, we redirect the incoming edge to $q_v$ into $q_u$, and discard the sub-tree starting from $q_v$, as shown in Figure 6.2(b). This gives us a finite-state plan $p$ as shown in Figure 6.2(c), which is a generalized plan for $\mathcal{P}$.

To see this, for any basic planning problem $P_k$ where $k > N$, the execution of $p$ will enter $q_u$ in state $\langle k - u, \vec{b}^r \rangle$. If $k - u > v$, the execution of $p$ will follow the loop and enter $q_u$ again in state $\langle k - u - v, \vec{b}^r \rangle$. This repeats until $\langle v', \vec{b}^r \rangle$ is reached at $q_u$ for some
\( v' \leq v \). From there, the execution of \( p \) is guaranteed to terminate and achieve the goal, in the same way \( p' \) runs on \( P_{v'+u} \in \mathcal{P}' \). This is due to the fact that \( v' + u \leq N \), and that \( q_u \) is reached in the very state \( \langle v', b' \rangle \) when running \( p' \on P_{v'+u} \), from which \( p' \) terminates and achieves the goal without ever reaching \( q_v \).

Theorem 11 gives us a sound and complete algorithm to solve 1ds problems: given problem \( \mathcal{P} = \{ P_0, P_1, \cdots \} \), we use any existing approach, e.g., reduction to classical planning introduced in [Hu and De Giacomo, 2011a], to find a conditional plan for the finite problem set \( \{ P_0, P_1, \cdots, P_N \} \). If a plan is found, we can construct a finite-state plan that works for \( \mathcal{P} \) using the construction sketched above; otherwise, no plan exists for \( \mathcal{P} \).

**Corollary 2.** If a 1ds generalized planning problem has a plan, then it has a finite-state plan.

This justifies the use of finite-state plans as a plan representation for 1ds problems, and indicates that the FsaPlanner algorithm in Chapter 5, which enumerates and verifies all finite-state plans with \( 1, 2, 3, \cdots \) plan states, is guaranteed to terminate with a valid plan on 1ds problems, as long as one exists.

Finally, we stress that these results are obtained on 1ds problems, which are a fragment of the one-dimensional problems in Chapter 4 without sequence fluents. Similar results may be obtained for the full one-dimensional class, but that requires further elaboration.

### 6.2 A Generic Algorithm for Controller Synthesis

The FsaPlanner algorithm in Chapter 5 focuses on achieving final-state goals using FSA plans. Given its positive results on those problems and the broader scope of generalized planning defined in the previous section, it is interesting to ask whether the algorithm can be extended to solve all generalized planning problems.
Unfortunately, this is not possible in the most general setting, since the representation of the problems itself can be infinite. However, as we shall see in this section, under a reasonable finiteness assumption, a general algorithm indeed exists. In the following, I shall propose such an algorithm, which is a natural generalization of the FsaPLANNER algorithm in Figure 5.3, and show that three recently proposed application domains, namely, controller synthesis [Bonet et al., 2009], service composition [De Giacomo et al., 2009] and planning programs [De Giacomo et al., 2010], can all be modeled and solved in this framework, and the results are competitive with their tailored approaches.

To understand the diversity of the problems and applicability of our solver, let us start with a brief overview of some example problems in these three domains.

**Controller synthesis:** In a $1 \times 5$ grid world shown in Figure 6.3(a), the robot is initially in one of the leftmost two cells. The goal is to visit cell $B$, and finally be in cell $A$. The robot can perform left and right moves within the grid, and can observe whether its current location is $A$, $B$ or neither of them. The synthesis task is to generate a controller that solves not only this problem instance, but also all $1 \times N$ grids in Figure 6.3(b). This problem is very similar to the problems for FsaPLANNER in Chapter 5, although the robot here can obtain observations of the states directly without the sensing actions. As a result, the solution to this type of problem is usually a Mealy-type controller, instead of a Moore controller like an FSA plan. Figure 6.3(c) shows a plan to the $1 \times 5$ grid problem. As reported in Sections 5.3.2 and 5.4.2, FsaPLANNER is able to solve most of these controller synthesis problems reformulated with sensing actions. In this section, we shall show that the generalized version of the planner is able to solve the problems in their original formulation with competitive efficiency.

**Service composition:** Imagine a service composition task where the goal is to provide a target service $M_T$ in Figure 6.4(a) from a set of available services $M_1$ and $M_2$ in Figure 6.4(b). Initially, both the target and the available services are in their initial states.
At any time, the target service may request an action from its current state, and an orchestrator should select one of the available services to perform this requested action. Upon action completion, the target and the chosen services will update their control states according to the transitions, but the orchestrator only has partial observability as which states the services are in (e.g. $o_0$ and $o_1$). It is the orchestrator’s responsibility to guarantee that all legal requests of the target can be satisfied at any time, and when the target is in its final states, so must be all the available services. Figure 6.4(c) shows one possible orchestrator for our example problem.

**Planning programs:** As a middle ground between automated planning [Ghallab et al., 2004] and high-level action languages like Golog [Levesque et al., 1997], planning programs enable the user to declaratively specify a goal network involving maintenance and achievement goals in a dynamic domain, and the planning problem is to find a strategy such that all goal requests can be accommodated for the long-lived agent.

Consider the researcher’s world involving walking, driving and bus-riding between
her home, department, the parking lot and the pub, shown in Figure 6.5(a). The goal network in Figure 6.5(b) specifies the everyday commuting needs of the researcher. For example, if the current goal node is $t_1$ in Figure 6.5(b), the next goal may be either a transition to $t_0$, requesting to “be home with the car parked at home while maintaining a non-empty fuel tank,” or a transition to $t_2$, requesting to “be in the pub while maintaining a non-empty tank.” In either case, the researcher must behave in a way that not only achieves the current goals, but also ensures that all possible future requests can still be satisfied after her actions. Figure 6.5(c) shows a control policy for this problem.

Figure 6.5: Planning program example.
6.2.1 Problem Representation

It is not difficult to see that all the three problems can be formalized as a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_m\}$ where

$$P_i = (\text{Acts}, \text{Obs}, \text{Events}, S, s_{i0}, \Delta, \text{Tr}, \text{obs}, \text{exec}),$$

and their plans are all finite-state plans characterized by a finite set $Q$, a transition function $\sigma$ and an action function $\alpha$. I shall show the formal details for the examples in Section 6.2.3.

In order to directly use the problem definition above as the input to the generalized planner, the representation must be finite. This means, in addition to restricting $S$ to be finite, the goal-satisfying traces $\text{Tr}$ must have a finite representation, even though each individual trace $\tau \in \text{Tr}$ may be infinite. For this reason, we restrict that the traces in $\text{Tr}$ must have the following property: given any finite-state plan, if an execution history on the problem contains repeating configurations, then there is no need to consider any extension of it, in the sense that the validity of any history that the current execution may generate can be reduced to the validity of a shorter history where the configurations between the repetition are removed.

For example, in controller synthesis, as the desired traces should directly satisfy the reachability goal, no execution history of a finite-state plan should visit the same configuration twice, as that would indicate either a shorter history, and thus a shorter trace, exists, or the execution will be trapped in a dead loop. This is an example where repeating configurations make an invalid history.

As another example, in service composition, two identical configurations in an execution history indicate that all the requests between the two are successfully fulfilled, and the same process could happen indefinitely for future requests of the same pattern, which is a desired behavior. So in this case, repeating configurations make a valid history, and
no extension needs to be further explored.

To capture this intuition formally, given a basic planning problem

\[ P = \langle \text{Acts}, \text{Obs}, \text{Events}, S, s_0, \Delta, \text{Tr}, \text{obs}, \text{exec} \rangle, \]

we require that for any finite-state plan \( p \) and its trace \( \tau \) where \( \tau(k) = s_k \), if

\[ (q_0, s_0) \cdot (q_1, s_1) \cdot \cdots \cdot (q_{n-1}, s_{n-1}) \cdot (q_n, s_n) \cdot \cdots \]

has two identical configurations \( q_i = q_j \) and \( s_i = s_j \) for \( 0 \leq i < j \leq n \), then \( \tau \in \text{Tr} \) if and only if \( \tau' \in \text{Tr} \), where

\[
\tau'(k) = \begin{cases} 
\tau(k) & \text{if } k \leq i, \\
\tau(k + (j - i)) & \text{otherwise.}
\end{cases}
\]

In the actual implementation of our generalized planner, we enforce the finiteness constraint by representing the goal-satisfying traces \( \text{Tr} \) for any basic and generalized planning problem via a behavior specification function \( \beta : (Q \times S)^* \rightarrow \{\text{True}, \text{False}, \text{Unknown}\} \), which has the property that for all history \( h' = (q_0, s_0) \cdot \cdots \cdot (q_n, s_n) \) where \( \langle q_i, s_i \rangle = \langle q_j, s_j \rangle \) for some \( 0 \leq i < j \leq n \), there exists a prefix \( h \preceq h' \) such that \( \beta(h) \in \{\text{True, False}\} \). Since both \( Q \) and \( S \) are finite, this assumption ensures that only histories with at most \( |Q| \cdot |S| \) need to be considered, and thus the goal can be finitely represented.

Intuitively, a True value for \( \beta(h) \) means that \( h \) is valid and conclusive, i.e., there is no need to further extend it; False means it is invalid, i.e., \( h \) should never be generated by the finite-state plan; Unknown means validity cannot be concluded yet, so all one-step execution need to be examined.

We say that a finite-state plan for a planning problem satisfies the behavior specification \( \beta \) iff for all history \( h \) that it can possibly generate, \( \beta(h) \neq \text{False} \) and there exists
an extension $h'$ such that $\beta(h') = \text{True}$. (In fact after a certain number of steps all its extension become \text{True}, due to the required condition above.)

With this definition, we can check whether any finite-state plan satisfies a behavior specification. More interestingly, if we do not have the plan yet, we can use the behavior specification function to actually synthesize one. This is what we study next.

### 6.2.2 Generic Solver

Generalizing the algorithm in Figure 5.3 of Chapter 5, I now propose a generic solver that solves all generalized planning problems with the finitely represented goals as described above, by systematically searching the space of bounded finite-state plans and traversing the AND-OR execution tree of the incremental partial plans. Here, the OR nodes are the choice points for the plan’s actions and transitions, while the AND nodes handle all possible environment feedback. Each node of the search tree keeps a current partial plan, along with its current program state, the world states and the execution history so far.

Given a generalized planning problem $P = \{P_1, \ldots, P_m\}$ where

$$P_i = \langle \text{Acts}, \text{Obs}, \text{Events}, S, s_i, \Delta, Tr, obs, exec \rangle,$$

and a bound $N$ on the number of program states in $Q = \{1, \ldots, N\}$, assuming the goal-satisfying traces is represented by a behavior specification function $\beta$, the algorithm in Figure 6.6 generates a finite-state plan with transition function $\sigma$ and action function $\alpha$ by a call to $\text{synthesize}_N(I)$, where $I = \{s_{i0}, \ldots, s_{m0}\}$ is the set of initial states in $P$ (line 1). This creates the root of the search tree, which is an AND node containing an initial plan with empty transition and action functions $\sigma = \alpha = \emptyset$. The initial control state is 1, the set of initial world states is $I$, and the execution history starts with the empty sequence $\langle \rangle$.

The function $\text{AND}\_\text{step}$ (lines 4–7) represents AND nodes in the search tree that
1:  synthesize$_N$(I)
2:  return AND_step$_N$((∅, ∅), 1, I, ⟨⟩);
3:
4:  AND_step$_N$((σ, α), q, S, h)
5:    for each s ∈ S
6:        (σ, α) := OR_step$_N$((σ, α), q, s, h · ⟨q, s⟩);
7:  return (σ, α);
8:
9:  OR_step$_N$((σ, α), q, s, h)
10:    if β(h) = True return (σ, α);
11:    else if β(h) = False fail;
12:    else if σ(q, obs(s)) = q’ and α(q, obs(s)) = a
13:        S’ := {s’ | ⟨s, exec(a), s’⟩ ∈ ∆};
14:        return AND_step$_N$((σ, α), q’, S’, h);
15:    else
16:        NON-DETERMINISTICALLY CHOOSE
17:            a ∈ Acts and q’ ∈ {1, · · · , N};
18:        S’ := {s’ | ⟨s, exec(a), s’⟩ ∈ ∆};
19:        σ’ := σ ∪ {⟨q, obs(s)⟩ → q’};
20:        α’ := α ∪ {⟨q, obs(s)⟩ → a};
21:        return AND_step$_N$((σ’, α’), q’, S’, h);

Figure 6.6: A generic algorithm for generalized planning.

handle all contingencies in the world states. For this, the function OR_step is called for each possible state s ∈ S, with the history h augmented with the current control and world states ⟨q, s⟩. Note that σ and α are updated after each call to OR_step (line 6), so that the resulting plan is incrementally implemented to handle all states in S.

The function OR_step (lines 9–21) simulates a one-step execution of the current partial plan with transition function σ and action function α for a given program state q and world state s with execution history h. Four different cases may arise during this simulation:

1. If the behavior specification function β(h) returns True, it means the current partial plan has generated a valid and conclusive history, so no further extension is necessary. In this case, ⟨σ, α⟩ is returned as a good partial plan (line 10).

2. If β(h) returns False, it indicates that h is illegal, so no extension of the current
partial plan can be a valid plan. In this case, the current search branch fails, and the algorithm backtracks to the most recent non-deterministic choice point (see below), from where alternative choices are explored (line 11).

3. Otherwise, \( \beta(h) \) must have returned unknown, indicating a legal but non-conclusive history, so further simulation of the plan is needed. If the transition function \( \sigma \) and action function \( \alpha \) are defined for the current program state \( q \) and observation \( obs(s) \), then we simply follow it by recursively calling AND_step with the current partial plan and history \( h \), but successor control state \( q' = \sigma(q, obs(s)) \) and the set \( S' \) of all possible successor world states obtained by executing action \( a = \alpha(q, obs(s)) \) (lines 12–14).

4. If, on the other hand, the transition and action functions are undefined for \( q \) and \( obs(s) \), then the algorithm makes a non-deterministic choice for \( q' \) and \( a \) (lines 16–17). We recursively call AND_step to handle all successor states in the same way as in the previous case, except that the new mappings appended to the functions before calling (lines 19–20).

Like the FsaPlanner algorithm, this algorithm is also formulated using nondeterministic choices to reveal the compactness of our solution. It is not hard to see that the resulting search actually strategically enumerates all valid plans with up to \( N \) states, e.g., avoiding plans with unreachable states.

In the discussion below, I always assume that the bound \( N \) on the number of control states is given. If not, an iterative deepening search over \( N \) could be used, which is guaranteed to terminate, since the environment, the plan and required behavior tests are all finite.

In practice, for the generic solver to work well, the non-deterministic choice at line 16 must be resolved wisely, making use of the structure of the target problem. For example, in most application domains, compact plans are preferable, so one should try
to reuse control states as much as possible; for planning tasks like generalized planning and planning programs, one could make use of domain-independent heuristics developed in the state-of-the-art planners [Hoffmann and Nebel, 2001, Richter and Westphal, 2010] for effective action selection, like what we did in Section 5.3.

6.2.3 Examples

In the following, I will show how the generic algorithm can be instantiated to efficiently synthesize plans in the three different applications illustrated in the introduction of this section.

Controller Synthesis

Bonet et al. formalize the controller synthesis problem as $C = \langle F, I, A, G, R, O, D \rangle$ [Bonet et al., 2009], where

- $F$ is a set of (primitive) fluents,
- $I$ is a set of $F$-clauses representing the initial situation,
- $A$ is a set of actions with conditional effects,
- $G$ is a set of literals representing the goal situation,
- $R$ is a set of non-primitive fluents,
- $O$ is the set of observable fluents, $O \subseteq R$, and
- $D$ is the set of axioms defining the fluents in R.

Without going into details, e.g., the evaluation of non-primitive fluents in $R$ using the axioms in $D$, we note that this problem can be modeled as a set of basic planning problems $P_i = \langle Acts, Obs, Events, S, s_0, \Delta, Tr, obs, exec \rangle$, where

- $Acts = Events = A$ and $exec$ is the identity function,
• $\text{Obs} = d(O)$ and $S = d(F)$, where $d(V)$ is the cross product of the domains for all variables $v \in V$,

• $s_{i0}$ are all the states in $S$ satisfying the initial condition $I$,

• $(s, a, s') \in \Delta$ iff action $a$ changes state $s$ to $s'$ in $C$,

• $\text{obs}(s) = o$ iff $o$ is observed in state $s$ in $P$.

In addition, $Tr$ is represented by the behavior specification accepts all legal execution histories leading to a state satisfying the goal, and rejects those that contain repeated configurations (indicating an unnecessary/infinite loop) and that cannot be extended (indicating dead end). Formally,

$$\beta((q_0, s_0), \ldots, (q_k, s_k)) =$$

\[
\begin{cases}
\text{TRUE} & \text{if } s_k \models G; \\
\text{FALSE} & \text{if } (s_k, a, s') \notin \Delta \text{ for all } a \in \text{Events}, s' \in S, \text{ or} \\
& (q_k, s_k) = (q_i, s_i) \text{ for some } 0 \leq i < k; \\
\text{UNKNOWN} & \text{otherwise.}
\end{cases}
\]

The structure of the generalized planning problems is very close to that of our generic problem, so the adaptation of the algorithm is straightforward. I implemented the adapted planner in SWI-Prolog, ran it on the set of benchmark problems in the original work of [Bonet et al., 2009]. The performance of our planner is compared with the compilation approach of Bonet, Palacios and Geffner (BPG) [Bonet et al., 2009] and the constraint programming based planner (Dyncode) by Pralet [Pralet et al., 2010]. Since their planners are not publicly available, we did not rerun their experiments, but instead used the data in the respective paper directly, so this comparison only serves as a feasibility check for our approach, due to the different experiment settings explained in Table 6.1.
In the table, column $N$ shows the number of control states required for the smallest plan, and for each planner, “Solve” is the solution time (in seconds) with $N$ states, and “Prove” is the time (in seconds) needed for the planner to prove that no plan exists with less than $N$ states. Surprisingly, the simple adaptation of our generic solver, which essentially performs depth-first search, achieves comparable performance to Dyncode, and works much faster than BPG in some cases. We believe that rewriting our planner in a compiled language like C and including effective heuristics will further improve the performance of our planner. Especially the latter may play a big role when we are faced with more difficult problems.

**Service Composition**

De Giacomo *et al.* defines each service $M_i$ as a tuple $\langle A, O_i, X_i, x_{i0}, F_i, \delta_i, a_i \rangle$ [De Giacomo et al., 2009], where
• $A$ is the (shared) set of actions;

• $O_i$ is a set of observations;

• $X_i$ is a set of service states;

• $x_{i0} \in X_i$ is the initial service state;

• $F_i \subseteq X_i$ is a set of final states, where the service is allowed to terminate;

• $\delta_i \subseteq X_i \times A \times X_i$ is the transition relation of the service;

• $o_i : X_i \to O_i$ is the observation function.

To formalize their definition as a generalized planning problem, we model the joint
behavior of the services as a dynamic environment, and let each basic planning problem
$P_i$ be of the form

$$P_i = \langle Acts, Obs, Events, S, s_{i0}, \Delta, Tr, obs, exec \rangle,$$

where

• $Acts = Events = \{1, \ldots, n\}$ are the index of the services the orchestrator may
  choose from at each step,

• $Obs = O_1 \times \cdots \times O_n \times A$ are the observations of the service states and next action
  request,

• $S = X_T \times X_1 \times \cdots \times X_n \times A$ is the joint state space,

• $s_{i0} = \langle x_{T0}, x_{10}, \ldots, x_{n0}, a_i \rangle$ where $a_i$ are all the actions that the target service may
  possibly request in the initial state, i.e., $\langle x_{T0}, a_i, \cdot \rangle \in \delta_T$,

• $\Delta(\langle x_T, x_1, \ldots, x_n, a \rangle), k, (t'_T, t'_1, \ldots, t'_n, a') \rangle$ iff
Chapter 6. Generalizations and Applications

1. \( \delta_T(x_T, a, t'_T) \),

2. \( \delta_k(x_k, a, t'_k) \),

3. \( x_i = t'_i \) for all \( i \notin \{T, k\} \), and

4. \( \delta_T(t'_T, a', t) \) for some \( t \in X_T \).

\[ \text{obs}(\langle x_T, x_1, \cdots, x_n, a \rangle) = \langle o_1(x_1), \cdots, o_n(x_n), a \rangle. \]

Notice that the state of the target service \( M_T \) is not observed, but the orchestrator can keep track of it using its internal states, since the target service is deterministic.

For the goal-satisfying traces \( Tr \), the behavior specification needs to consider the following factors

1. at any time, the requested action (stored as the last element of a state tuple) must be executable by at least one service;

2. (final state constraint:) whenever the target service is in a final state, so must be all composing services;

3. (state repetition:) if we reach a configuration (control state and world state combined) that has been visited already in the execution history, then all future executions from the current configuration would be handled in the same way as from its previous occurrence, so there is no need to consider further extensions. the current history.
Formally, we define the behavior specification as

\[
\beta((q_0, s_0), \ldots, (q_k, s_k)) =
\begin{cases}
\text{True} & \text{if } q_i = q_k \text{ and } s_i = s_k \text{ for some } 0 \leq i < k; \\
\text{False} & \text{if } (s_k, a, s) \notin \Delta \text{ for all } a \in \text{Events}, s \in S, \text{ or} \\
& s_k = (x_T, x_1, \ldots, x_n, a) \text{ where } x_T \in F_T \\
& \text{but } x_i \notin F_i \text{ for some } i \in \{1, \ldots, n\}; \\
\text{UNKNOWN} & \text{otherwise.}
\end{cases}
\]

We instantiated the generic algorithm for service composition, and implemented the planner in SWI-Prolog. During the instantiation, we noticed that, although the non-determinism of the environment is specified as a single transition relation \( \Delta \) in our formalization above, the non-determinism actually comes from two different sources, namely, the uncertainty as to which action the target may request, and the nondeterministic effect of each composing service.

As a result, we took this special structure of the problem into account, and used two AND steps in the Prolog implementation, one for each source of non-determinism above. This dramatically reduces the observation cases in the search nodes, since only the state of the chosen service changes and thus needs to be observed, and observations on all other services can be safely ignored. Exploiting this structure makes the branching factor of the AND nodes exponentially smaller, and thus contributes to a much more efficient solver.

We experimented with our planner on 18 benchmark problems on an Intel Core2 3.0GHz CPU with 3.0GB memory. The problems requires the composition of target services of 2–4 states from 2–5 available services ranging from 2 to 10 states. Twelve of the smaller problem instances used in the experiments are shown in Figure 6.7, and the full list of problems is available online at http://www.cs.toronto.edu/~yuxiao/compose.tgz.
All problems are either solved or proved unsolvable within less than 0.01 second. This matches the results obtained by model checking for problems with full observability, and is much faster, finding smaller orchestrators, for cases with partial observability [De Giacomo et al., 2009]. The obtained results are not definitive since the existing benchmarks are quite small, but certainly they show the effectiveness of the proposed approach in practice.

**Planning Programs**

Next, I move on to planning programs, like the researcher’s example in the introduction, recently proposed by De Giacomo et al. [De Giacomo et al., 2010].

According to them, a *domain* is a tuple $D = \langle P, A, X_0, \rho \rangle$, where $P$ is a set of propositions, $A$ is a set of actions, $X_0 \in 2^P$ is the initial state, and $\rho \subseteq 2^P \times A \times 2^P$ is the set of transitions. A *planning program* for $D$ is a tuple $T = \langle T, G, t_0, \delta \rangle$, where

- $T = \{t_0, \ldots, t_q\}$ is the finite set of *program states*;
- $G$ is a finite set of goals of the form “achieve $\phi$ while maintaining $\psi$,“ denoted by pairs $g = \langle \psi, \phi \rangle$, where $\psi$ and $\phi$ are propositional formulas over $P$;
- $t_0 \in T$ is the *program initial state*;
- $\delta \subseteq T \times G \times T$ is the *program transition relation*.

This synthesis problem for planning programs can be modeled in our framework as a generalized planning problems $\mathcal{P} = \{P_i \cdots\}$ with

$$P_i = \langle Acts, Obs, Events, S, s_{i0}, \Delta, Tr, obs, exec\rangle,$$

where
Figure 6.7: Example composition problems used in our experiments. Labels of sensing results are omitted when the service is fully observable.
• \( Acts = Events = A \) is their set of available actions, and \( exec \) is the identity function,

• \( Obs = S = 2^P \times T \times G \times T \), and \( obs \) is the identity function.

• \( s_i^0 \) are all the tuples \( \langle X_0, t_0, g, t \rangle \) where \( \langle t_0, g, t \rangle \in \delta \);

• \( \Delta(\langle x, t, \langle \psi, \phi \rangle, t' \rangle, a, \langle x', t, \langle \psi, \phi \rangle, t' \rangle) \) iff
  1. \( \langle x, a, x' \rangle \in \rho \),
  2. \( x \models \psi \) and \( x' \models \psi \), and
  3. \( x' \not\models \phi \);

• \( \Delta(\langle x, t, \langle \psi, \phi \rangle, t' \rangle, a, \langle x', t', g, t'' \rangle) \) iff
  1. \( \langle x, a, x' \rangle \in \rho \),
  2. \( x \models \psi \) and \( x' \models \psi \),
  3. \( x' \models \phi \),
  4. \( \langle t', g, t'' \rangle \in \delta \);

The goal-satisfying traces \( Tr \) is characterized by the behavior specification that has the following properties:

1. (desired state repetition:) if the system returns to a configuration after some other goal transitions, then it indicates that all the goals have been satisfied between the two repeating configurations, and the same can happen for all future goal requests of the same pattern, so the history is valid and needs no further exploration;

2. (undesired state repetition:) if the system returns to a configuration while resolving the same goal transition, then it indicates the existence of a dead loop, since the system will repeat the configurations indefinitely without achieving the goal request, so this type of state repetition should never occur in a valid history;
3. as usual, violation of maintenance goals and dead ends should be avoided.

Let \( s_i = \langle x_i, t_i, \langle \psi_i, \phi_i \rangle, t'_i \rangle \), the following behavior specification function \( \beta \) encodes these constraints.

\[
\beta(\langle q_0, s_0 \rangle, \cdots, \langle q_k, s_k \rangle) =
\begin{cases}
\text{TRUE} & \text{if } s_i = s_k \text{ for some } 0 \leq i < k \text{ and } t_k \neq t_{k-1}; \\
\text{FALSE} & \text{if } s_i = s_k \text{ for some } 0 \leq i < k \text{ and } t_k = t_{k-1}, \text{ or } \\
& s_k \not\models \psi_k, \text{ or } \langle s_k, a, s \rangle \notin \Delta \text{ for all } a, s; \\
\text{UNKNOWN} & \text{otherwise.}
\end{cases}
\]

Adapting the generic algorithm in Figure 6.6, we obtain a solver for planning programs in SWI-Prolog. Like in service composition, a plan for a planning program is faced with two sources of nondeterminism in each cycle, namely, the uncertainty about which goal may be requested, and the nondeterministic effects of the actions. This special structure is also utilized in the implementation by introducing two AND steps. In the OR steps, for effective action selection, the additive heuristics [Bonet and Geffner, 2001b] in classical planning is used.

We ran the resulting planner on two benchmark problems, including the researcher’s world and the more realistic smart home application in [De Giacomo et al., 2010]. On the Intel Core2 3.0Hz CPU with 3GB RAM, our planner is able to solve them within 0.04 and 0.39 seconds respectively, while the model checking approach in [De Giacomo et al., 2010] requires several minutes for the first problem, and times out after 24 hours for the second! Notice again that we only considered relatively simple cases, so the results are not definitive but very promising.
6.3 Concluding Remarks

This chapter explores the application and generalization of the results obtained in the previous chapters.

The first contribution (Section 6.1) of this chapter lies in the definition of generalized planning, which offers, possibly for the first time, a common ground for existing work in the planning with loops literature. An advantage of the new definition is that the flat representation is independent of any formalism, and arguably makes it easier to gain insight into the problem structures. For example, a considerable fraction of the theoretical result on one-dimensional planning problems is reconstructed and extended under this definition.

The other contribution is the generic algorithm for generalized planning problems (Section 6.2) obtained by generalizing the FsaPlanner algorithm in Chapter 5. In principle, it solves all generalized planning problems with a finite state space and behavior specification bounded by state repetition. The adaptation of this algorithm to the three diverse application domains shows competitive to superior performance compared with the original tailored approaches in the literature.

Compared to the previous chapters, the results in this chapter are relatively preliminary. It shows a generic framework and solver for generalized planning, which leads to promising results, but many extensions and more analysis are yet to be explored. We leave them for future work.
Chapter 7

Conclusions

In this thesis, I studied AI planning problems where plans take the form of finite-state controllers. The technical contributions of this thesis include the following aspects.

1. We presented a formal definition of planning for plans in the form of Moore-type finite-state controllers in the situation calculus framework, based on a previous formalization using robot programs [Levesque, 1996]. We showed that the new definition is a more compact and general alternative, and it served as the foundation for the rest of the thesis.

2. We studied a property of the planning problems called “finite verifiability,” with which the general correctness of a plan on possibly infinitely many cases can be reduced to its correctness on a finite subset of those cases. We identified several classes of problems with and without this property. We showed that if the problem can be characterized by \( k \) decreasing integer parameters, or by exactly 2 parameters one of which may also be increased, then its plans are finitely verifiable. On the other hand, if the problem contains sequence fluents indexed by an integer that can be both increased and decreased, then it is powerful enough to encode the Halting problem, so finite verifiability cannot be achieved. This study still leaves open the finite verifiability of many problems between the two classes, e.g., the two-
dimensional problems with unrestricted incrementability, whose finite verifiability would lead to a proof for the long-standing Collatz conjecture.

3. We proposed a generic algorithm for automatically generating plans with loops by directly searching in the space of finite-state plans and their execution trees. We showed that a direct implementation of the base algorithm outperforms Levesque’s Kplanner [Levesque, 2005] with the same settings. We further explored two ways to enhance algorithm’s efficiency, by adding heuristics and randomizing the search, respectively, and showed that the resulting planners can solve a number of interesting and challenging benchmark problems, some of which are not solved by other existing approaches.

4. We explored a generalization of our results to accommodate other existing forms of planning with loops, including the Mealy-type plan representation and non-terminating plans for temporal goals. For this purpose, we presented a definition of generalized planning that is independent of existing formalisms. We showed that this definition captures many existing forms of planning, and simplifies our earlier finite-verifiability proof for an important fraction of the one-dimensional planning problems. We also generalized the planning algorithm in this more general setting, and showed that simple instantiations of the resulting generic algorithm lead to efficient planners for three diverse application domains including controller synthesis, service composition and planning programs.

This thesis also opens up a promising direction for future research on planning with loops. Among all the possibilities, here are a few interesting immediate next steps.

1. This thesis only investigated the finite verifiability for problems expressed in terms of integer planning parameters. Apart from exploring more deeply the splitting line between the verifiable and non-verifiable problems in this class, another interesting direction is to investigate problems where the incomplete knowledge about the
infinite property is represented by other means, *e.g.*, a complete axiomatization of the blocks world if the problem involves all the possibilities of stacks of blocks [Cook and Liu, 2003].

2. The generic search algorithm in Chapter 5 reveals the structure of planning with loops problems, and more insight can be obtained to efficiently solve such problems. For example, the current heuristics described in Section 5.3 handles action selection and transition selection separately, which sometimes lead to suboptimal estimations. As a result, a better heuristics that simultaneously considers the actions and transitions is highly desirable and deserves more investigation.

3. The application of our algorithm in the existing application domains proved promising and competitive with the tailored approaches in the literature. These results lead to the solution of problems which are previously deemed too difficult (*e.g.*, the smart home application), and thus encourage more extensive study of those problems, including the development of more challenging benchmark problems and domain-dependent techniques to enhance the solver instantiation.
Appendix A

Example Plans

This appendix contains an example plan for each of the benchmark problems automatically generated by FsAPLANNER as described in Chapter 5.

- arith
• bintree

- STOP
- check_node
  - target
  - leaf
  - internal
- pop_from
- push_to(left)
- push_to(right)
- left
- right

• cornerA

- wall(up)
- wall(left)
  - yes
  - move(up)
  - no
  - move(left)
    - yes
    - wall(up)
    - no
    - move(up)

- wall(ahead)
  - yes
  - move_forward
  - turn_left
    - yes
    - wall(ahead)
    - no
    - STOP
  - no
  - move_forward

• cornerR
• delivery

• factorial
• **green**

![Diagram for green]

• **gripper+**

![Diagram for gripper+]
• gripper

\[
\begin{align*}
\text{checkB} & \quad \text{empty} \quad \text{nonempty} \\
& \quad \text{STOP} \quad \text{load} \\
& \quad \text{move} \\
& \quad \text{unload} \\
& \quad \text{move} \\
\end{align*}
\]

• hanoi

\[
\begin{align*}
\text{pick(c)} & \quad \text{put(b)} \quad \text{ok} \\
\text{put(a)} & \quad \text{fail} \\
\text{pick(b)} & \quad \text{ok} \quad \text{fail} \\
\text{put(c)} & \quad \text{fail} \\
\text{put(b)} & \quad \text{fail} \\
\text{pick(c)} & \quad \text{ok} \quad \text{ok} \\
\text{pick(a)} & \quad \text{fail} \\
\end{align*}
\]
- prizeA
• prizeR
• recycle

• sort
• striped
• transport
• trash

• treechop
• variegg

• visual-M


