

Solving conformant planning using Chen's determinizing method

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA

BY

Amine Abou-Rjeili

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

April 2008

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of master's thesis by

Amine Abou-Rjeili

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Dr. Charles Hudson Turner

Name of Faculty Adviser

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Acknowledgements

As with any major undertaking, the contributions of others throughout the rugged path towards the goal make things that much easier and deserve more credit than what can be expressed on a page. Nevertheless, I will extend my gratitude in this section. Unfortunately, due to memory limitations and page size, one is bound to omit some of these wonderful people. For those not mentioned here, please accept my apologies and my thanks.

First and most important I would like to thank my parents, Nouhad and Samir, and my brother, Jad, for their endless support, sacrifices, profound dedication, love and belief in me of which without, this thesis would have not been possible. I hope this makes you proud.

As for my dear friend, Roy, whose help made everything that much easier, I am eternally grateful and will never forget. These two sentences do not give justice to the help you have offered to me. I thank you.

The patience, wisdom and support attributed by my advisor, Dr. Hudson Turner, is unforgettable. I cannot stress enough my appreciation and gratitude owed to Dr. Turner for making the completion of this thesis possible. Even though these two sentences do not give justice to the hard work and dedication exhibited by Dr. Turner, I hope you will accept my thanks.

As very well said by Frances Ward Weller,

"A friend can tell you things you don't want to tell yourself".

For this I extend my special thanks to my friends for their support, advice and tolerance. I know I can be difficult at times and yet you were always there.

I would also like to extend my appreciation to the computer science department of the University of Minnesota Duluth for all the help through this endeavor.

Thank you all.

Contents

1	Introduction	2
1.1	Introduction	3
1.1.1	Classical Planning	3
1.1.2	Conformant Planning	6
1.1.3	The Road Ahead	7
2	Planning Framework	10
2.1	Planning Framework	11
2.1.1	Problem Representation Language	11
2.1.2	Transitions and Histories	15
2.1.3	Conformant Planning	16
2.1.4	Classical planning	17
2.1.5	Benchmark Problems	19
3	Determinizing	22
3.1	Determinizing	23
3.2	Copyfree	28

3.3	Determinizing a multiple initial fluent	29
3.4	Determinizing a nondeterministic effect of an action	33
3.5	Termination	34
3.6	Uniformity	35
3.7	Adequacy	36
3.8	Preservation under determinizing	36
3.9	Proof of Theorem 1	37
3.9.1	Proof of lemmas for Theorem 1	41
3.10	Proof of Theorem 2	51
3.10.1	Proof of lemmas for Theorem 2	52
3.11	Proof of Theorem 3	75
3.12	Properties of Affected Sets	77
3.13	Proof of Preservation lemmas	79
3.14	Proof of Termination	80
4	Implementation	83
4.1	Implementation	84
4.1.1	Input parser	88
4.1.2	PDDL output plugin	88
5	Experimental Results	91
5.1	Experimental Results	92
5.1.1	Bomb in the Toilet	92
5.1.2	Ring of Rooms	93

5.1.3	SQUARE	95
5.1.4	CUBE	95
5.1.5	Result tables	95
6	Related Work	107
6.1	Related Work	108
7	Future Work	110
7.1	Future Work	111
A	Appendix A	113
B	Appendix B - Blocks world PDDL description (from AIPS2000)	115
C	Appendix C - BTUC(2)	118
D	Appendix D - RING(2)	120
E	Appendix E - SQUARE(5)	123
F	Appendix F - CUBE(3) [corner]	126
G	Appendix G - SAFE(5)	129

List of Figures

1.1	Pictorial view of BLOCKS(4).	4
1.2	BLOCKS(4) problem description.	5
1.3	BTC(2) problem description	7
1.4	Solution to BTC(2).	8
2.1	An example of a non-operator	15
2.2	An example of a nondeterministic effect	15
2.3	Graphical view of SQUARE(5) problem.	20
2.4	An example of an action that makes use of one of our extensions of the language of Chen.	21
3.1	The two worlds of the BTC(2) problem corresponding to the uncertainty in the initial state with regard to the location of the bomb.	24
3.2	The result of determinizing BTC(2).	25
3.3	An example of a problem with nondeterministic effects that cannot be determinized.	27
3.4	The determinized problem from Figure 3.3.	27
3.5	An example of a conformant planning problem that does not adhere to the <i>copyfree</i> condition.	28
3.6	Determinizing a multiple fluent proposition in a conformant planning problem that is not <i>copyfree</i>	29

3.7	The result of determinizing a conformant problem $Q = \langle I, D, G \rangle$ and any multiple initial fluent d in I	32
3.8	The result of determinizing a conformant problem $Q = \langle I, D, G \rangle$ and any nondeterministic fluent proposition g in W in D	34
4.1	Overview of application framework.	85
5.1	Description of the <i>move-left</i> action from the RING(2) problem	95

Abstract

Planning is the task of generating a series of actions that transform a world from its initial state to one satisfying a predefined goal. In this thesis we will be concerned with conformant planning, that is, planning problems that exhibit uncertainty about the initial state and nondeterministic action effects. In addition, no sensing of the environment is permitted during plan execution. In this context a planning problem is defined in terms of three components:

- a description of the initial state,
- a description of the goal,
- a description of how actions affect the state of the world.

In defining a valid plan in such an environment, we distinguish two concerns:

- executability—the guarantee that no matter which of the possible initial states and no matter what the outcome of nondeterministic actions, the whole plan can be executed, and
- whenever the plan is executed the goal is achieved.

A widely-studied special case of conformant planning known as classical planning is concerned with environments where there is no uncertainty: the initial state is completely known and the effects of actions are deterministic. For this type of problem, defining a valid plan is a more straightforward process due to the lack of uncertainty. Much research has been done on classical planning systems, and so it might be beneficial if one could transform a conformant problem to an equivalent classical problem and solve it using top-of-the-line classical planners. This is the foundation of our approach.

Liu developed a method to transform a deterministic conformant problem, defined as a conformant problem with uncertainty only in the initial state, into a classical problem. Liu's approach is implemented as a slight extension to the Causal Calculator (CCalc), a publicly available system for planning and reasoning about actions. Chen extended this approach to consider nondeterministic action effects. Even in the case of initial uncertainty, Chen's determinizing method provides a more concise alternative to the approach introduced by Liu. Also, Chen proved that the determinizing method is correct for uncertain initial states and demonstrated successful results in solving benchmark problems, such as the Bomb in the Toilet problem and the Ring of Rooms problem. However, a proof of correctness for determinizing nondeterministic action effects was not given.

This thesis is focused on extending the determinizing method of Chen to be more widely applicable, automating the determinizing method, and comparing our runs on standard benchmark problems with top-end conformant planners. To this end, we have implemented a determinizing frontend that converts a conformant problem into a corresponding classical problem and then solves it using any state-of-the-art classical planning system. This allows us to take advantage of advances in classical planning systems. Furthermore, we prove correct the method of action determinizing, thus completing the proof of correctness of the determinizing procedure.

Chapter 1

Introduction

1.1 Introduction

Planning is the task of finding a sequence of actions that guarantee the achievement of a goal. “Conformant planning” [6] is planning under uncertainty without any sensory information. The possible sources of uncertainty are nondeterminism in the effects of actions and incomplete knowledge of the initial state. In this type of planning problem, a plan is a sequence of actions that has to guarantee success no matter which state the world is in initially (among those consistent with what is known about the initial state) and no matter what the outcome of any nondeterministic action that is executed. The more widely-studied *classical planning* is the special case of conformant planning in which there is no uncertainty. That is, the initial state of the world is fully known, and actions have deterministic effects. This type of planning has a lower computational complexity than conformant planning, due to its relative simplicity. When only plans of polynomially-bounded length are considered, classical planning is NP-complete (assuming typical languages for describing the components of the planning problem). By comparison, conformant planning may be \sum_2^p -complete or even \sum_3^p -complete, depending on various assumptions and/or restrictions on the action description language [11].

In this thesis, we extend the work of Chen [4], which is based on the work of Liu [9], in developing a *determinizing* technique used to transform a conformant problem into an equivalent classical problem, which in turn can be solved using state-of-the-art classical planners. Due to the fact that classical planners have evolved to be very efficient as compared to conformant planners, this technique provides a viable alternative for solving conformant planning problems. Initial experiments carried out by Chen on standard benchmark problems indicate that in some cases this approach can outperform top-end planners that solve conformant problems directly (i.e., conformant planners). To complement the determinizing technique, Chen introduced an action representation language based on STRIPS [8] that allows for the representation of conformant planning problems. As will be seen later, this language has inconvenient limitations and we will introduce enhancements to overcome some of these limitations.

1.1.1 Classical Planning

To better illustrate the concept of classical planning, a blocks world problem with four blocks, BLOCKS(4), will be presented. This problem has been taken from the Artificial Intelligence Planning Systems 2000

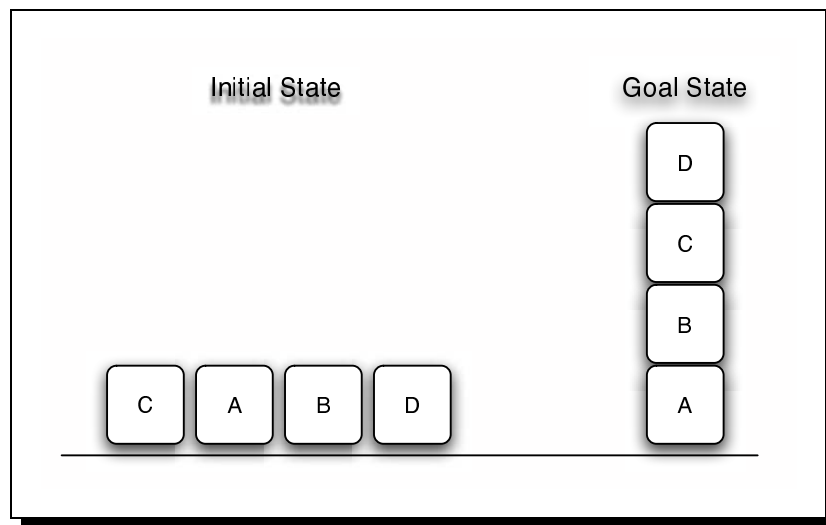


Figure 1.1: Pictorial view of BLOCKS(4).

(AIPS 2000) competition ¹. This problem has four blocks laid out on a table next to each other with the goal being to place them on top of each other in a column as illustrated in Figure 1.1. In this problem, there are four actions: pick-up, put-down, stack, unstack. The effect of each action is self-described by the action's name.

Let us take a look at the problem description, shown in Figure 1.2. In the initial state of the world, blocks *A*, *B*, *C*, *D* are clear, they are on the table and the hand (used to grab the blocks) is empty. The initial state assigns one value for each of the attributes in the world either being positive if present in the description or negative if not. From this convention it follows that the initial state is fully known.

The goal description is interpreted differently from the initial description in that we do not assume that “unmatched” attributes are negative. It tells us the values that certain attributes should have. Also notice that no uncertainty is involved in any of the action effects; every effect is totally deterministic. For example, looking at action *pickup*, we see that the effect is to remove the object from the table and place it in the hand of the robot, thus causing the robot to be holding it. In addition the object is “uncleared” to not allow any other object to be placed on top of it. Since the initial state and action effects are totally deterministic, this is a classical planning problem.

¹A tarball of the problems from the competition can be found at <http://www.cs.toronto.edu/aips2000/aips-2000datafiles.tgz>

Initial State: $\{clear(C), clear(A), clear(B), clear(D), ontable(C), ontable(A), ontable(B), ontable(D), handempty\}$

Action domain D:

Operator: $pickup(x)$

Precondition: $clear(x) \wedge ontable(x) \wedge handempty$

Effects: $\{\neg ontable(x), \neg clear(x), \neg handempty, holding(x)\}$

Operator: $putdown(x)$

Precondition: $holding(x)$

Effects: $\{\neg holding(x), clear(x), handempty, ontable(x)\}$

Operator: $stack(x,y)$

Precondition: $holding(x) \wedge clear(y)$

Effects: $\{\neg holding(x), \neg clear(y), clear(x), handempty, on(x,y)\}$

Operator: $unstack(x,y)$

Precondition: $on(x,y) \wedge clear(x) \wedge handempty$

Effects: $\{holding(x), clear(y), \neg clear(x), \neg handempty, \neg on(x,y)\}$

Goal: $on(D,C) \wedge on(C,B) \wedge on(B,A)$

Figure 1.2: BLOCKS(4) problem description.

1.1.2 Conformant Planning

There are many problems that exhibit uncertainty both in the description of the initial state of the world and in the action effects. Furthermore, it can be the case that, due to different factors, information about the environment cannot be known during plan execution. Problems of this nature are known as conformant problems.

To become more comfortable dealing with such problems, we will briefly describe a standard benchmark problem for conformant planners and introduce new terminology to be able to describe such problems.

A well known family of conformant problems is the *Bomb in the Toilet* family of problems. These problems come in many variants, but for this introduction we will be dealing with a simple one. Consider a room with a toilet and several packages of which one of them contains a bomb. A bomb defusing robot has been called upon to defuse the bomb. The only way to do so is to dunk the package with the bomb in the toilet, which in turn clogs the toilet. The problem with a clogged toilet is that a package cannot be dunked in it and so the toilet needs to be flushed to unclog it. To complicate matters further, the robot does not know which package contains the bomb. This type of problem is known as $BTC(n)$, where n is the number of packages. Figure 1.3 shows a description of $BTC(2)$ in which the world is described in terms of the “fluents” in , $clogged$, $damp(P1)$, $damp(P2)$ and $armed$. (Informally, a fluent describes a property of the world.) Fluent in indicates the location of the bomb in terms of the package that contains it, fluent $clogged$ indicates the status of the toilet (either clogged or not), fluents $damp(P1)$ and $damp(P2)$ indicate whether packages $P1$ and $P2$, respectively, are damp or not, and lastly, fluent $armed$ indicates whether the bomb is armed or not. The initial state of the world has uncertainty concerning the location of the bomb. This is reflected in Figure 1.3 in the description of the initial state; the *fluent proposition*

$$in \text{ in } \{P1, P2\}$$

is satisfied no matter whether the bomb is in package $P1$ ($in = P1$) or in package $P2$ ($in = P2$).

So how can the agent solve such a problem without being able to sense the environment? One possible solution would be to dunk package 1 ($dunk(P1)$), then flush the toilet to unclog it ($flush$) and then dunk package 2 ($dunk(P2)$). Let us call this sequence of actions *Plan-1*, for which a graphical view is shown in Figure 1.4. The initial states considered in Figure 1.4 are just those consistent with the description of the initial state. The sequence *Plan-1* of actions leads from each one of those initial states to a state that

satisfies the goal description. To illustrate this consider the initial state in which $in = P1$ (this means that the bomb is in package 1). Applying the actions (in order) $dunk(P1), flush, dunk(P2)$ leads to the state $\{in = P1, clogged, damp(P1), damp(P2), \neg armed\}$ which satisfies the goal description. Notice here that the same sequence of actions also leads to a state that satisfies the goal description in the case where the bomb is initially in package 2. Thus *Plan-1* is indeed a solution to the BTC(2) problem, since regardless of the initial state of the world, a goal state is guaranteed to be reached. Other variations of this problem include uncertainty in the clogging of the toilet and having multiple toilets.

Fluents: in [$domain(in) = \{P1, P2\}$],
 $clogged$ [$domain(clogged) = \{true, false\}$],
 $damp(P1)$ [$domain(damp(P1)) = \{true, false\}$],
 $damp(P2)$ [$domain(damp(P2)) = \{true, false\}$],
 $armed$ [$domain(armed) = \{true, false\}$]

Initial State: $\{in \text{ in } \{P1, P2\}, \neg clogged, \neg damp(P1), \neg damp(P2), armed\}$

Action domain D:

Operator: $dunk(x)$

Precondition: $\neg clogged \wedge \neg damp(x)$

Effects: $\{\neg armed \text{ when } in = x, clogged, damp(x)\}$

Operator: $flush$

Precondition:

Effects: $\{\neg clogged\}$

Goal: $\neg armed$

Figure 1.3: BTC(2) problem description

1.1.3 The Road Ahead

Liu [9] introduced a method, known as *determinizing*, that can sometimes transform a conformant problem into a classical problem. In this approach Liu creates a “copy” of the world for each of the possible initial states and then constructs a classical plan that will solve the problem “simultaneously” in all of these worlds.

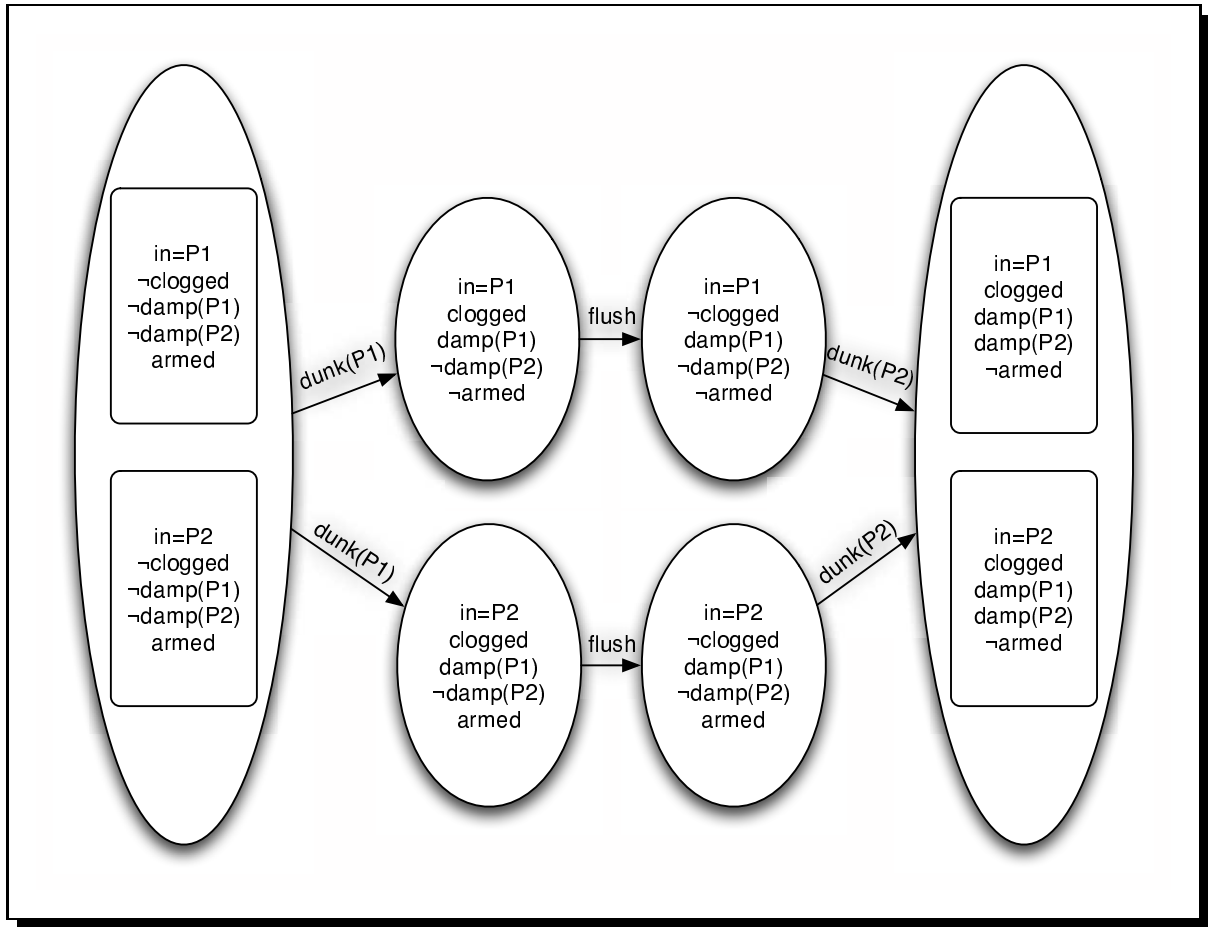


Figure 1.4: Solution to BTC(2).

However, Liu’s approach can handle uncertainty only in the initial state, while the action effects are required to be deterministic. In extending Liu’s work, Chen’s [4] approach can eliminate some duplicate features from the worlds of the initial state to save on space. That is, while Liu’s approach required a full copy of the world for each initial state, Chen showed how it is sometimes possible to get by with only “partial” copies. In addition, Chen’s method can handle some problems with nondeterministic action effects.

In this thesis, we extend Chen’s work by enhancing Chen’s problem description language to be more expressive. To compliment this, we show that Chen’s proof of the correctness of determinizing initial states carries over to the new problem description language. Moreover, we *prove* that the determinizing procedure works for nondeterministic effects. We also automate the determinizing procedure by implementing a frontend application that takes as input a conformant planning problem in the enhanced problem description language, and outputs an equivalent classical planning problem in the standard classical planning language PDDL [7]. The resulting classical planning problem is then fed into a classical planner to produce a plan for the classical problem, which is also a plan for the conformant problem, as we will show in this thesis. Furthermore, we experimentally compare the performance of our approach to state-of-the-art conformant planners.

Chapter 2 describes our planning framework and defines the language used to encode conformant problems. Chapter 3 moves on to define the determinizing procedure and proves that it works on initial states and on nondeterministic action effects. Chapter 4 describes our implementation of a parser that uses the determinizing procedure to transform a conformant problem to a classical problem. Chapters 5 and 6 then present our results and compare them with the results of top-end conformant planners. Due to time constraints, certain items still need to be addressed and these are discussed in Chapter 7 as future work. At the end of the thesis, one finds appendices with descriptions of the different problems used in our experiments.

Chapter 2

Planning Framework

2.1 Planning Framework

In this chapter we will define a problem representation language, based on the language introduced by Chen [4], that will be used to describe conformant planning problems. In contrast to the language of Chen, where action preconditions and the description of the goal were expressed by conjunctions of fluent propositions, our extension will allow these to be expressed using fluent formulas (of which a formal definition will be presented later on). As a result, this will allow for the representation of more complex problems.

2.1.1 Problem Representation Language

The basic building blocks of a problem description are a finite set of symbols called *actions* and a finite set of symbols called *fluents*, disjoint from the actions. Associated with each fluent is a nonempty finite set of symbols called its *domain*. Intuitively, fluents characterize the state of the world; each fluent takes exactly one value at a time from its domain. Actions define how the world moves from one state to another.

A planning problem is described as a triple $\langle I, D, G \rangle$ where I describes the initial state, D describes the actions (known as an action domain) and G describes the goal to be achieved.

We will denote the domain of a fluent f by $domain(f)$. For example, from the BTC(2) problem, as presented in Section 1.1, the constant *in* is a fluent with $domain(in) = \{P1, P2\}$. A fluent with the Boolean domain $\{true, false\}$ is called a *Boolean fluent*. Here, we need to note a syntactic restriction on the symbols of a domain, in that they cannot contain the “.” character due to the special use of this character in the determining procedure. In addition, the fluents of the problem domain need to conform to a restriction known as “*copyfree*” (these restrictions will become clear in the next chapter).

A *fluent proposition* is an expression of the form

$$f \text{ in } V$$

where f is a fluent and V a nonempty subset of $domain(f)$. Intuitively this means that the value of f is one of the elements of V . If $|V| = 1$ then the fluent proposition is called *deterministic* and otherwise it is called *nondeterministic*.

As a notational convenience, one can write $f = v$ for $f \text{ in } \{v\}$. Furthermore, for the case where f is a

Boolean fluent, we can write f to stand for $f = true$ and $\neg f$ to stand for $f = false$.

Fluent Formulas and Partial State Descriptions

The symbols

$$\wedge, \vee, \supset, \equiv, \top, \perp, \neg$$

are *propositional connectives*. \perp (false) and \top (true) are 0-place connectives, \neg (negation) is a 1-place or unary connective, and \wedge (conjunction), \vee (disjunction), \supset (implication), and \equiv , (equivalence) are 2-place or binary connectives.

A *fluent formula* is a formula of propositional logic whose atoms are fluent propositions. Alternatively it can be defined recursively as follows.

- Every fluent proposition is a fluent formula.
- Both \top and \perp are fluent formulas.
- If ϕ is a fluent formula then so is $\neg\phi$.
- If ϕ, γ are fluent formulas then $(\phi \odot \gamma)$ is a fluent formula, where \odot is a binary connective.

A fluent formula is used to describe the attributes of the world at a particular moment in time.

It is sometimes convenient to omit some parentheses from a fluent formula. For example, a fluent formula of the form $(F \odot G)$ can be written as $F \odot G$. These abbreviations will become clear when used in context.

A *partial state description* (PSD) is a set of fluent propositions in which each fluent appears at most once.

A PSD corresponds to the fluent formula obtained by conjoining its elements.

Satisfaction of Fluent Formulas and Partial State Descriptions

A *state* is a function that maps each fluent f to a value from $domain(f)$. It can also be thought of as an alternative representation for a special kind of a partial state description where

- each fluent appears exactly once, and
- for each fluent proposition f in V , $|V| = 1$.

By definition, a state s assigns a value to a fluent f , and we write $s(f)$ to denote the value assigned by s to f .

For any state s and any fluent formula ϕ , we define the *satisfaction of ϕ by s* , denoted $s \models \phi$, in a similar manner to how an interpretation satisfies a formula in propositional logic. A recursive definition of this is as follows.

- $s \models f$ in V iff $s(f) \in V$
- $s \models \top$, $s \not\models \perp$
- For any fluent formula ϕ , $s \models \neg\phi$ iff $s \not\models \phi$
- For any fluent formulas ϕ, γ ,
 - $s \models \phi \wedge \gamma$ iff $s \models \phi$ and $s \models \gamma$
 - $s \models \phi \vee \gamma$ iff $s \models \phi$ or $s \models \gamma$
 - $s \models \phi \supset \gamma$ iff if $s \models \phi$ then $s \models \gamma$
 - $s \models \phi \equiv \gamma$ iff $s \models \phi \supset \gamma$ and $s \models \gamma \supset \phi$

Similarly, a state s *satisfies* a partial state description P if it satisfies every fluent proposition in P , and we write $s \models P$. Notice that every *PSD* is satisfied by at least one state.

Operators

An *effect proposition* is an expression of the form

$$f \text{ in } V \text{ when } \phi,$$

where f in V is a fluent proposition and ϕ a fluent formula. ϕ is called the *antecedent* and f in V the *consequent*. If $|V| = 1$ then the effect proposition is *deterministic*. (This is equivalent to an effect proposition

as defined in Chen with the difference being that for Chen the antecedent of an effect proposition is a PSD instead of a fluent formula as defined here.) As a convenience, for any effect proposition C , we sometime write $Ant(C)$ to refer to the antecedent of C and $Con(C)$ to refer to the consequent of C .

An *operator* describes the effects of an action and has the following syntax

Operator: a

Precondition: Pre

Effects: E

where a is an action, Pre a fluent formula and E a set of effect propositions satisfying the following restriction:

Restriction 2.1.1. *For any distinct effect propositions f in V when ϕ and f in V' when ψ in E , for any state s ,*

if s satisfies Pre then s satisfies at most one of ϕ and ψ .

This restriction will guarantee that in any state that satisfies the precondition of a , the result of executing a is well defined.

In the operator effects, as an abbreviation, we will use the expression f in V to stand for f in V when \top . Thus, for example, the effect proposition

$\neg clogged$ when \top

can be written as

$\neg clogged$.

To gain a better understanding of the syntax of an operator, let us look at the operator $dunk(x)$ from the BTC(2) problem (described in Section 1.1), as shown in Figure 1.3. (Here it is important to note that operator $dunk(x)$ is a schematic representation of operators $dunk(P1)$ and $dunk(P2)$, with x a metavariable ranging over $P1, P2$.) Notice that $dunk(x)$ has the precondition $\neg clogged \wedge \neg damp(x)$ and the effect $\{\neg armed$ when $in = x, clogged, damp(x)\}$. Intuitively, this means that the $dunk(x)$ action can occur only if the toilet is not clogged and package x is not damp. The effect of this action is to disarm the bomb if it is in package x , to clog the toilet, and to make package x damp.

Operator: $dunk(P)$
 Precondition: $\neg clogged \wedge \neg damp(P)$
 Effects: $\{\neg armed \mathbf{when} in(P), clogged, \neg clogged, damp(P)\}$

Figure 2.1: An example of a non-operator

An example of a non-operator is shown in Figure 2.1. The problem here is that in the effect we have the effect propositions

1. $clogged$
2. $\neg clogged$

which violate Restriction 2.1.1, since any state satisfies the antecedent \top . (Recall $clogged$ is a shorthand notation for $clogged \mathbf{when} \top$ and $\neg clogged$ a shorthand notation for $\neg clogged \mathbf{when} \top$).

Operator: $dunk(P)$
 Precondition: $\neg clogged \wedge \neg damp(P)$
 Effects: $\{\neg armed \mathbf{when} in(P), clogged \mathbf{in} \{true, false\}, damp(P)\}$

Figure 2.2: An example of a nondeterministic effect

Up to this point we have dealt with actions whose effects are deterministic. However, conformant problems allow for action effects to be nondeterministic and so to get an idea of what such an action looks like, we show one in Figure 2.2. Nondeterminism here stems from the effect

$clogged \mathbf{in} \{true, false\}$

where it is not known if dunking the package will clog the toilet or not.

An *action domain* is a set of operators, exactly one for each action. An action domain is said to be *nondeterministic* if it contains at least one operator with a nondeterministic effect. Otherwise it is *deterministic*.

2.1.2 Transitions and Histories

An operator is *executable* in a state s if $s \models Pre$, where Pre is the precondition of the operator.

The world *transitions* from state s to state s' using action a , denoted $tr(s, a, s')$, if

1. a is executable in s ,

2. for each effect proposition f **in** V **when** ϕ in the effects of a , if $s \models \phi$ then $s' \models f$ **in** V ,
3. for all fluents f , if there is no effect proposition f **in** V **when** ϕ in the effects of a s.t. $s \models \phi$, then $s'(f) = s(f)$.

Condition 3 captures the “commonsense law of inertia”: things don’t change unless they’re made to.

From the definition of transition, it follows that for any state s and any action a that is executable in s , there is at least one state that s transitions to: that is, there is at least one state s' such that $tr(s, a, s')$. This is due to Restriction 2.1.1.

A *history* is a sequence $\langle s_0, a_0, s_1, \dots, s_{n-1}, a_{n-1}, s_n \rangle$ of states and actions such that $tr(s_i, a_i, s_{i+1})$ for $0 \leq i \leq n - 1$. We say that n is the *length* of the history.

2.1.3 Conformant Planning

A *conformant planning problem* is a triple $\langle I, D, G \rangle$ where I is a partial state description describing the initial state, D an action domain, and G a fluent formula describing the goal to be achieved.

We say a set S of states *satisfies* a fluent formula ϕ if for all $s \in S$, $s \models \phi$.

An action is *executable* in a set S of states if $S \models Pre$.

For sets of states S, S' we say the world transitions from S to S' using action a , denoted $tr(S, a, S')$, if a is executable in S and $S' = \{s' | tr(s, a, s'), s \in S\}$.

An *epistemic history* is a sequence $\langle S_0, a_0, S_1, \dots, S_{n-1}, a_{n-1}, S_n \rangle$ of sets of states and actions such that $tr(S_i, a_i, S_{i+1})$ for $0 \leq i \leq n - 1$, where n is the *length* of the history.

A *plan* for a conformant planning problem is a sequence $\langle a_0, \dots, a_{n-1} \rangle$ of actions with a corresponding epistemic history $\langle S_0, a_0, S_1, \dots, S_{n-1}, a_{n-1}, S_n \rangle$ where $S_0 = \{s | s \models I\}$ and $S_n \models G$. Intuitively this means that the plan guarantees the achievement of the goal starting from any state consistent with I .

It is important to note here that S_0 is guaranteed to be nonempty because I is a PSD and, as noted earlier, for every PSD there is at least one state that satisfies it.

2.1.4 Classical planning

A *classical planning problem* is a triple $\langle I, D, G \rangle$ where I is a state (the initial state), D a deterministic action domain, and G a fluent formula describing the goal state. Classical planning is the special case of conformant planning in which the actions can have only deterministic effects, and I is a state. (As mentioned earlier, a state can be understood as a special kind of a PSD.)

A *plan* for a classical planning problem is a sequence $\langle a_0, a_1, \dots, a_{n-1} \rangle$ of actions with a corresponding history $\langle s_0, a_0, \dots, a_{n-1}, s_n \rangle$ such that $s_0 \models I$ and $s_n \models G$. As can be seen, this is essentially a special case of a conformant plan: here we have a sequence of states and actions, whereas in the conformant case we have a sequence of sets of states and actions.

Next is an example of a classical planning problem. A blocks world with two blocks, *block(2)*, will be discussed. A description of the problem is as follows:

Set of fluents: $\{holding, onTableA, onTableB, clearA, clearB, BonA, AonB\}$

$domain(holding) = \{A, B, NOTHING\}$

$domain(onTableA) = \{true, false\}$

$domain(onTableB) = \{true, false\}$

$domain(clearA) = \{true, false\}$

$domain(clearB) = \{true, false\}$

$domain(BonA) = \{true, false\}$

$domain(AonB) = \{true, false\}$

Set of actions: $\{pickupA, pickupB, putBonA, putAonB, takeoffBFromA, takeoffAFromB\}$

Initial State I : $\{holding = NOTHING, onTableA, onTableB, clearA, clearB, \neg BonA, \neg AonB\}$

Action domain D :

Operator: *pickupA*

Precondition: $holding = NOTHING \wedge clearA \wedge onTableA$

Effects: $\{holding = A, \neg clearA, \neg onTableA\}$

Operator: pickupB

Precondition: holding = NOTHING \wedge clearB \wedge onTableB

Effects: {holding = B, \neg clearB, \neg onTableB}

Operator: putdownA

Precondition: holding = A

Effects: {holding = NOTHING, onTableA, clearA}

Operator: putdownB

Precondition: holding = B

Effects: {holding = NOTHING, onTableB, clearB}

Operator: putBonA

Precondition: clearA \wedge holding = B

Effects: {BonA, \neg clearA, clearB, holding = NOTHING}

Operator: putAonB

Precondition: clearB \wedge holding = A

Effects: {AonB, \neg clearB, clearA, holding = NOTHING}

Operator: takeoffBFromA

Precondition: BonA \wedge clearB \wedge holding = NOTHING

Effects: {holding = B, clearA, \neg clearB, \neg BonA}

Operator: takeoffAFromB

Precondition: AonB \wedge clearA \wedge holding = NOTHING

Effects: {holding = A, clearB, \neg clearA, \neg BonA}

Goal G : BonA

It is easy to see that this problem is a classical planning problem since all action effects are deterministic and the initial description is a state.

A plan of minimal length for this problem is $\langle \text{pickupB}, \text{putBonA} \rangle$.

2.1.5 Benchmark Problems

In this section, we will present informal descriptions of the conformant problems used for our experimental purposes. For full descriptions of specific instances of these problems, see Appendix C for the *BTUC*(2) problem, Appendix D for the *RING*(2) problem, Appendix E for *SQUARE*(5) problem, Appendix F for *CUBE*(3) and Appendix G for *SAFE*(5). The first problem is the bomb in toilet family of problems (see Section 1.1 for a general description). In this thesis, we will be dealing with four different instances of this problem, as follows.

- *BTC*(n) - BT problem with n packages and uncertainty in the initial state; it is not known which of the n packages the bomb is in.
- *BTUC*(n) - *BTC*(n) problem with the addition that in the effects of action *dunk* it is not known whether the toilet will clog or not. Figure 2.2 shows the nondeterministic operator *dunk*, used in the problem.
- *BMTC*(n, t) - *BTC*(n) problem with the addition of having t toilets.
- *BMTUC*(n, t) - *BTUC*(n) problem with the addition of having t toilets.

The bomb in the toilet problems are a classic benchmark for conformant solvers and were used by ([4], [9], etc.).

The next problem used in our experiments is the Ring of Rooms, denoted *RING*(n), where n is the number of rooms. It was introduced by Cimatti et al. in [5]. A general description is as follows.

There is a ring of rooms each consisting of a window that can be open, closed or locked. A robot can move to each room either clockwise or counterclockwise and can close a window if it is open and lock it if it is closed. The goal is to lock the windows in all the rooms. The uncertainty here is in the initial state in that it is not known which room the robot is in or the status of the windows. In this problem instance, the windows obey the law of inertia in that a window cannot change its state unless caused by some action.

There is another variation of this class of problems, discussed in [5], in which a window can become open if it is unlocked and there is no action acting upon it. For example, if the robot is closing the window in room 1, then the window in room 2 may change its state to become open from closed, for example. This class of problems is referred to as the $URING(n)$ where n is the number of rooms. However, we did not consider this type of problem due to reasons that will be discussed in Section 5.1.

The third problem that we used for our experiments is the SQUARE problem, denoted $SQUARE(n)$, where n is the number of subsquares per row or column of the square. For example, $SQUARE(5)$ has a square with 5 units on each side for a total of 25 subsquares. This is illustrated in Figure 2.3. This problem was used in

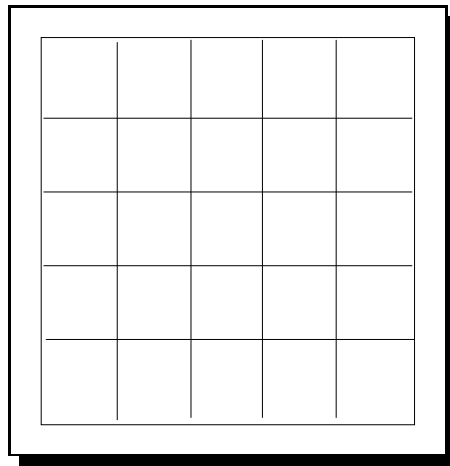


Figure 2.3: Graphical view of SQUARE(5) problem.

[2] to test the GPT planner. This is a navigation problem where initially the robot is at an unknown position and the goal is to reach a predefined position. There are four actions in this problem— one action to move in each direction (up, down, left, right). Note here that in the case where it is not possible to move further in a given direction, the action of “moving” in the direction leaves the robot’s position unchanged. There are two variations to this problem; *corner* and *center*. The difference is in the destination position of the robot. In *corner*, the robot must reach a corner position, whereas in *center* the robot must reach the center of the square. In this thesis, we will deal with *corner* since this is the instance that was used by [2]. A solution to this problem is for the robot to move $n - 1$ positions in each dimension of the square.

Next we considered the CUBE problem, denoted $CUBE(n)$, where n is the number of subcubes in each dimension. This is the $SQUARE(n)$ problem extended to 3 dimensions. This problem was used by [2] and

[3] in their experiments. This is very similar to $SQUARE(n)$ with the only difference being that the robot is in a cube instead of a square. Again there are two variations to this problem— *center* and *corner*. In our experiments we consider both variations. In this problem there are six actions to allow for the robot to move in each direction of the cube.

The last problem considered is the *Safe* problem, denoted $SAFE(n)$ where n is the total number of combinations available. This problem was used by [3] to evaluate the performance of Conformant-FF. In this problem, there is a safe where there is one combination from a pool of n combinations that opens it, with the correct combination being unknown. The objective is to open the safe. A solution to this is to try all different combinations. This will guarantee that the safe will end up open, no matter which is the correct combination.

The planning problems presented thus far do not benefit from our extensions to the planning framework. However, since fluent formulas are more expressive than partial state descriptions (recall that PSDs are a special case of fluent formulas), our language allows for expressing more complex planning problems. For a simple example consider a variant of the bomb in the toilet problem where the dunk action can occur if the toilet is not clogged and the package is not damp or if the package contains the bomb. This can be easily expressed, using our extensions, as shown in Figure 2.4.

Operator: $dunk(x)$
Precondition: $(\neg clogged \wedge \neg damp(x)) \vee in = x$
Effects: $\{\neg armed \textbf{ when } in = x, clogged, damp(x)\}$

Figure 2.4: An example of an action that makes use of one of our extensions of the language of Chen.

Chapter 3

Determinizing

3.1 Determinizing

Determinizing is a technique used to transform a conformant planning problem into a classical planning problem by eliminating uncertainty in the initial state and in the action effects. This technique was first developed by Liu [9] to handle uncertainty only in the initial state. Roughly speaking, Liu introduced a “copy” of the world corresponding to each possible initial state and simultaneously solved the corresponding classical planning problem for each of these worlds. In further developing this technique, Chen [4] showed that uncertainty in the initial state can be eliminated more efficiently by, roughly speaking, removing duplicate attributes from the different worlds. Furthermore, Chen then extended the procedure to handle uncertainty in the action effects, given certain assumptions about the problem domain. As stated by Chen, the procedure can always remove all uncertainty from the initial state but with a worst-case exponential cost. However, there is no guarantee that all uncertainty can be removed from the action effects, and in cases where this is possible there will still be a worst-case exponential cost [4].

In this section we will extend Chen’s determinizing procedure slightly. (Chen’s description language required that an action precondition be a PSD, instead of a fluent formula, and similarly that the antecedent of an effect proposition be a PSD, again instead of a fluent formula, as we allow.) We will then prove that this technique produces an equivalent problem with all uncertainty removed from the initial state and, where applicable, from the action effects. That is we will show that, for any conformant problem $\langle I, D, G \rangle$, if $\langle I', D', G' \rangle$ results from determinizing $\langle I, D, G \rangle$, then a plan is a solution to $\langle I, D, G \rangle$ iff it is a solution to $\langle I', D', G' \rangle$.

It is important to note here that in each phase of determinizing, we determinize one “multiple fluent” at a time until there are no more multiple fluents. Informally a multiple fluent is a fluent whose value might be unknown either initially or after execution of some action. A formal definition will be provided later in this chapter. One here should also consider the order of determinizing multiple fluents, since different orders may produce different sized problems. Due to time limitations we will not study this problem in depth in this thesis.

High-level pseudocode for the determinizing procedure is given in Algorithm 1.

To help in understanding these concepts, one can think of a conformant problem as being composed of multiple problems with no uncertainty. For example, consider the BTC(2) problem as described earlier. The

Algorithm 1 $\text{Determinize}(\langle I, D, G \rangle)$

Determine all multiple fluents in the initial state

Determine all multiple fluents in the action effects

return $\langle I', D', G' \rangle$

World 1	World 2
$in = P1$	$in = P2$
$\neg clogged$	$\neg clogged$
$armed$	$armed$
$\neg damp(P1)$	$\neg damp(P1)$
$\neg damp(P2)$	$\neg damp(P2)$

Figure 3.1: The two worlds of the BTC(2) problem corresponding to the uncertainty in the initial state with regard to the location of the bomb.

only uncertainty is in the initial state, where it is not known which package the bomb is in. This can be decomposed into two worlds, where in the first world the bomb is in package 1 and in the second world the bomb is in package 2, as shown in Figure 3.1. Figure 3.2 shows a representation of this new problem in our language. It is easy to check that a sequence of actions that will solve the problem in both worlds, such as the one in Figure 1.4, will solve the original conformant problem (regardless of which package contains the bomb). The main idea of the determinizing method is to create a description of a planning problem that encapsulates all the possible worlds, while eliminating uncertainty both in the initial state and, if possible, in the action effects. Notice that the description in Figure 3.2 manages to encapsulate the two possible worlds without requiring two copies of every fluent: while there are two copies of in and $armed$, one copy each of $clogged$, $damp(P1)$ and $damp(P2)$ is sufficient.

Here, we should note that, for convenience, we will often refer to the states of D' (in the determinized problem) as *extended states* in order to distinguish them from the states of D (in the conformant problem), which we will sometimes refer to as *original states*.

It is important to understand that, since I is a PSD, there might be fluents not mentioned in I , in which case

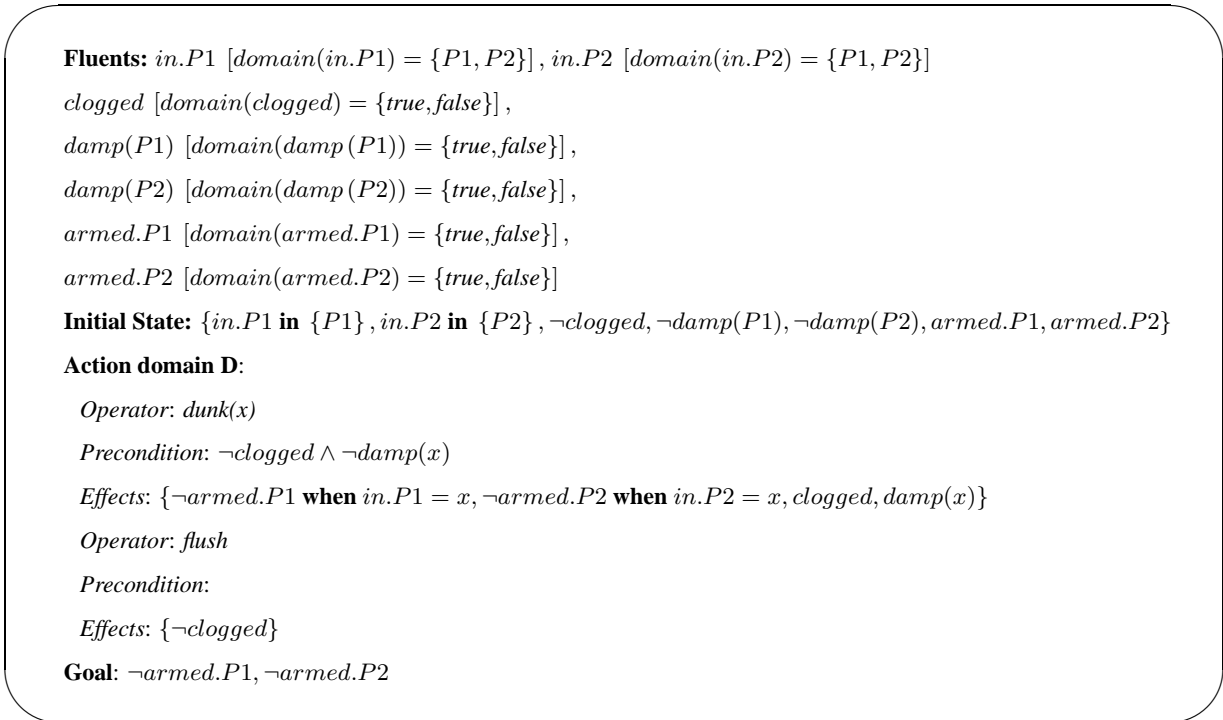


Figure 3.2: The result of determinizing BTC(2).

for any such fluent h we add

$$h \text{ in } domain(h)$$

to I . One can easily check that this does not affect the meaning of I in any way - which is to say that it does not affect the states that satisfy I .

At this point we present theorems for the determinizing procedure. (These theorems employ some concepts that will be defined a little later, but appear here to help the reader anticipate what will follow.)

Theorem 1. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree, if we obtain $\langle I', D', G' \rangle$ by determinizing a multiple initial fluent, then A^* is a solution for $\langle I, D, G \rangle$ iff A^* is a solution for $\langle I', D', G' \rangle$.*

Theorem 2. *For any conformant planning problem $\langle I, D, G \rangle$ in which D is copyfree and uniformity preserving, if $\langle I', D', G' \rangle$ is obtained by determinizing a nondeterministic fluent proposition and I' is uniform, then A^* is a solution to $\langle I, D, G \rangle$ iff A^* is a solution to $\langle I', D', G' \rangle$.*

Theorem 3. *For any conformant problem $\langle I, D, G \rangle$, if D is adequate then D is uniformity preserving.*

There are several differences between our Theorem 1 and Theorem 2 and the corresponding theorems presented by Chen. Here we explicitly specify the condition that D is *copyfree*. Furthermore, in our definitions it is implicit that G , and all fluent preconditions and operator preconditions in D are formulas (as defined in Section 2.1) whereas Chen defines them as partial state descriptions (PSDs).

To prove the correctness of the determinizing procedure we need to prove Theorem 1, Theorem 2, and Theorem 3. (Here, it should be noted that it is easy to check for uniformity of I' and so a proof will not be presented.) Furthermore, we need to prove that *copyfree*, uniformity preserving, adequacy and uniformity are preserved under determinizing. This is required where D contains more than one nondeterministic fluent proposition (or multiple initial fluent) and thus multiple passes of the determinizing procedure on D are needed (see Algorithm 1). (Section 3.8 presents the appropriate lemmas for which the proofs can be found in Section 3.13.)

Notice that when determinizing a nondeterministic fluent we have enforced certain requirements on I' and D for the determinizing procedure to work (i.e., for Theorem 2 to hold). To illustrate why this is needed consider the conformant planning problem $\langle I, D, G \rangle$ shown in Figure 3.3. D is not uniformity preserving (as will be easy to check once we have presented the definition). As a result, in this example, the determinized problem has a solution that is not a solution for the original problem. Figure 3.4 shows the corresponding determinized problem $\langle I', D', G' \rangle$. Let $A^* = \langle foo, foo \rangle$ be a sequence of actions. This produces the epistemic history $\langle S_0, foo, S_1, foo, S_2 \rangle$ in $\langle I, D, G \rangle$ where

$$S_0 = \{ \{a = 0, \neg b\} \},$$

$$S_1 = \{ \{a = 1, \neg b\}, \{a = 2, \neg b\} \},$$

$$S_2 = \{ \{a = 1, b\}, \{a = 2, b\}, \{a = 1, \neg b\}, \{a = 2, \neg b\} \}.$$

Notice that $S_0 = Mod(I)$ and $S_2 \not\models G$ and therefore A^* is not a solution for $\langle I, D, G \rangle$. On the other hand, A^* produces the history $\langle s'_0, foo, s'_1, foo, s'_2 \rangle$ in $\langle I', D', G' \rangle$ where

$$s'_0 = \{ a.1 = 0, a.2 = 0, \neg b.1, \neg b.2 \},$$

$$s'_1 = \{ a.1 = 1, a.2 = 2, \neg b.1, \neg b.2 \},$$

$$s'_2 = \{ a.1 = 1, a.2 = 2, b.1, \neg b.2 \}.$$

In this case, $s'_0 \models I'$ and $s'_2 \models G'$ and thus A^* is a solution for $\langle I', D', G' \rangle$. The problem here arises because state s'_1 satisfies the fluent precondition $a.1 = 1$ but not the fluent precondition $a.2 = 1$ (and so D' does not preserve uniformity, which is one of the extra conditions required in Theorem 2). Notice here that the fluent preconditions $a.1 = 1$ and $a.2 = 1$ in D' are “derived” from the same fluent precondition ($a = 1$) in D , one for each world of the conformant problem. However, s'_1 does not satisfy this fluent precondition in both worlds and so s'_1 (in the determinized world) is not consistent with S' (in the original world), resulting in different histories for the same sequence of actions. To guarantee that this situation does not occur, we introduce the concept of uniformity. Furthermore, we need to guarantee that uniformity is preserved as we transition from one state to another, which leads to the concept of uniformity preserving. The importance of these concepts will become clear as we present the proof in more detail.

$$\begin{aligned}
 I &= \{a = 0, \neg b\} \\
 D: \\
 \text{Operator: } &foo \\
 \text{Precondition: } &\top \\
 \text{Effects: } &\{a \text{ in } \{1, 2\} \text{ when } \top, b \text{ when } a = 1\} \\
 G &= (a = 2 \vee b)
 \end{aligned}$$

Figure 3.3: An example of a problem with nondeterministic effects that cannot be determinized.

$$\begin{aligned}
 I' &= \{a.1 = 0, a.2 = 0, \neg b.1, \neg b.2\} \\
 D': \\
 \text{Operator: } &foo \\
 \text{Precondition: } &\top \\
 \text{Effects: } &\{a.1 = 1 \text{ when } \top, a.2 = 2 \text{ when } \top, b.1 \text{ when } a.1 = 1, b.2 \text{ when } a.2 = 1\} \\
 G' &= (a.1 = 2 \vee b.1) \wedge (a.2 = 2 \vee b.2)
 \end{aligned}$$

Figure 3.4: The determinized problem from Figure 3.3.

$$\begin{aligned}
I &= \{a \text{ in } \{1, 2\}, a.2 \text{ in } \{3\}, b\} \\
D: & \\
\text{Operator: } &foo(x) \\
\text{Precondition: } &\top \\
\text{Effects: } &\{a = 2 \text{ when } a.2 = 3\} \\
G &= (a = 2)
\end{aligned}$$

Figure 3.5: An example of a conformant planning problem that does not adhere to the *copyfree* condition.

3.2 Copyfree

Notice that the framework defined in Section 2.1 does not enforce any restrictions on the naming of fluents and thus it is possible that fluents in the original problem (i.e., conformant problem) conflict with determinized fluents in the determinized problem. This might lead to situations where a conformant problem cannot be determinized as illustrated in the following examples.

Consider the conformant problem shown in Figure 3.5. Now, consider determinizing the multiple initial fluent a in I . As part of this process, we include the fluent proposition $a.2 = 2$ in I . But $a.2$ already appears in I and thus we cannot include the new fluent proposition (since I is a PSD, each fluent appears at most once in I). As a result, this problem cannot be determinized.

Next we present a similar example where we determinize a nondeterministic fluent proposition. To this end, consider the conformant planning problem shown in Figure 3.6. Consider determinizing nondeterministic fluent proposition $a \text{ in } \{2, 4\}$, which produces $a.2 = 2 \text{ when } b$ (in addition to the other copies of $a \text{ in } \{2, 4\} \text{ when } b$). But, foo contains the effect proposition $a.2 = 1 \text{ when } b$ and so $a.2 = 2 \text{ when } b$ cannot be added to the effects of foo (since Restriction 2.1.1 will be violated). Thus this problem cannot be determinized.

To overcome this difficulty we introduce the concept of *copyfree*.

Definition 3.2.1. *The set of copies of f is defined as follows:*

1. f is a copy of f

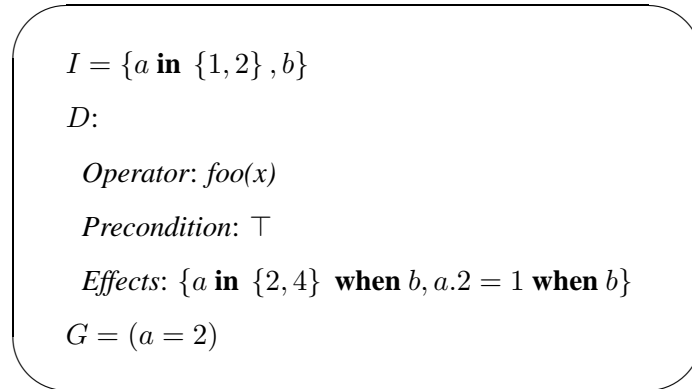


Figure 3.6: Determinizing a multiple fluent proposition in a conformant planning problem that is not *copyfree*.

2. if f' is a copy of f , then so is $f'.w$, for any $w \in domain(f'')$, for any fluent f'' .

Definition 3.2.2. A domain D is *copyfree* if, for all fluents f in D , the only copy of f is f .

Referring back to the examples, we see that in Figure 3.5, I contains fluents a and $a.2$, where $a.2$ is a copy of a (thus the problem is not *copyfree*). In Figure 3.6, D contains fluents a and $a.2$, where $a.2$ is a copy of a (thus the problem is not *copyfree*). Note that this is a technical issue due to the convention used to represent “extended” fluents. One can see that the importance of *copyfree* is pervasive and so it is assumed implicitly throughout the thesis.

3.3 Determinizing a multiple initial fluent

Given a conformant planning problem $\langle I, D, G \rangle$, a fluent f is a *multiple initial fluent* if I contains a fluent proposition $f \text{ in } V$ with $|V| > 1$. For example, in the *BTC(2)* problem, fluent *in* is a multiple initial fluent in I because $in \text{ in } \{P1, P2\}$ belongs to I . Notice here that for any multiple initial fluent f , there is exactly one $f \text{ in } V$ in I (since I is a PSD, f occurs at most once in I and for any f that is not included in I we add $f \text{ in } domain(f)$ to I).

Let d be a multiple initial fluent such that $d \text{ in } W \in I$. We now describe the transformation of $\langle I, D, G \rangle$ that determinizes d .

We say a fluent f *affects* a fluent f' (in D) if there is an operator in D with an effect $f' \text{ in } V \text{ when } \psi$ where

3.3. DETERMINIZING A MULTIPLE INITIAL FLUENT

f appears in ψ . For fluent d , we define its *affected set* A_d^D (in D) as the least set of fluents such that

1. $d \in A_d^D$,
2. for all fluents f, f' , if $f \in A_d^D$ and f affects f' (in D), then $f' \in A_d^D$.

We will use the notation A_d as a shorthand for A_d^D when it is clear from the context what D is.

When determinizing multiple initial fluent d in domain description D , we obtain a domain description D' where the fluents are:

- the original fluents of D that do not belong to A_d (that is, the fluents not “affected” by d), and
- new fluents $f.w$ for each $f \in A_d$ and $w \in W$.

(In Theorem 1 and Theorem 2, the condition that D is *copyfree* simply guarantees that all fluents $f.w$ in item 2 are indeed new.)

For any fluent f and all $w \in W$, define

$$\text{extend}(f, w) = \begin{cases} f.w & , \text{ if } f \in A_d, \\ f & , \text{ otherwise.} \end{cases} \quad (3.1)$$

Here it is important to note that for any fluent f , and any $w \in W$,

$$\text{domain}(\text{extend}(f, w)) = \text{domain}(f).$$

For any fluent proposition f **in** V , define

$$\text{extend}(f \text{ **in** } V, w) = \text{extend}(f, w) \text{ **in** } V. \quad (3.2)$$

For any partial state description P , define

$$\text{extend}(P, w) = \{\text{extend}(f \text{ **in** } V, w) : f \text{ **in** } V \in P\}. \quad (3.3)$$

extend is defined on fluent formulas as follows. The base case where the formula is a fluent proposition, is already defined.

$$\text{extend}(\perp, w) = \perp \quad (3.4)$$

3.3. DETERMINIZING A MULTIPLE INITIAL FLUENT

$$extend(\top, w) = \top \quad (3.5)$$

$$extend(\neg\phi, w) = \neg extend(\phi, w) \quad (3.6)$$

$$extend(\phi \odot \psi, w) = extend(\phi, w) \odot extend(\psi, w) \quad (3.7)$$

for any binary operator \odot

For any effect proposition f in V when ϕ , define

$$extend(f \text{ in } V \text{ when } \phi, w) = extend(f \text{ in } V, w) \text{ when } extend(\phi, w) \quad (3.8)$$

For a set of effect propositions S , define

$$extend(S, w) = \{extend(f \text{ in } V \text{ when } \phi, w) : f \text{ in } V \text{ when } \phi \in S\} \quad (3.9)$$

For W define

$$extend(f, W) = \{extend(f, w) : w \in W\} \quad (3.10)$$

$$extend(f \text{ in } V, W) = \{extend(f \text{ in } V, w) : w \in W\} \quad (3.11)$$

$$extend(P, W) = \bigcup_{w \in W} extend(P, w) \quad (3.12)$$

$$extend(\phi, W) = \bigwedge_{w \in W} extend(\phi, w) \quad (3.13)$$

$$extend(f \text{ in } V \text{ when } \phi, W) = \{extend(f \text{ in } V \text{ when } \phi, w) : w \in W\} \quad (3.14)$$

$$extend(S, W) = \bigcup_{w \in W} extend(S, w) \quad (3.15)$$

Figure 3.7 shows the result of determinizing a conformant planning problem with respect to a multiple initial fluent.

For any state s' in D' and any $w \in W$, define $s' \cdot w$ such that for any fluent f in the language of D

$$s' \cdot w(f) = s'(extend(f, w)). \quad (3.16)$$

So $s' \cdot w$ is the original state in D such that, roughly speaking, each fluent gets the value assigned to it by extended state s' (in D') with respect to the world w .

To illustrate the determinizing procedure, we will apply the procedure to the *BTC(2)* problem, defined in Figure 1.3, the result of which is shown in Figure 3.2. Notice that in the problem description we have

For any conformant problem $Q = \langle I, D, G \rangle$ and any multiple initial fluent d in I , the result of determinizing d is $Q' = \langle I', D', G' \rangle$, where

$$I' = \{d.w = w : w \in W\} \cup \text{extend}(I - \{d \text{ in } W\}, W),$$

$$G' = \text{extend}(G, W), \text{ and}$$

for each operator a in D of the form

Operator: a

Precondition: Pre

Effects: E

we include in D' the corresponding operator

Operator: a

Precondition: $\text{extend}(Pre, W)$

Effects: $\text{extend}(E, W)$

Figure 3.7: The result of determinizing a conformant problem $Q = \langle I, D, G \rangle$ and any multiple initial fluent d in I .

in in $\{P1, P2\}$ in I , making in a multiple initial fluent. Furthermore, it is the only multiple initial fluent. (In addition, D is deterministic.) Next we need to create the affected set of in (i.e., A_{in}). By definition, $in \in A_{in}$. Since in affects $armed$ (since in operator $dunk(x)$ we have $\neg armed$ **when** $in = x$ with x being a schematic variable ranging over $\{P1, P2\}$), we have $armed \in A_{in}$. So, we have that

$$A_{in} = \{in, armed\} .$$

Figure 3.2 shows the result of determinizing multiple initial fluent in .

Here it is important to note that when there is more than one multiple initial fluent, the repeated determinizing of multiple initial fluents is guaranteed to terminate. A detailed discussion can be found in Section 3.5.

3.4 Determinizing a nondeterministic effect of an action

After all initial multiple fluents have been determinized, we move to determinize action effects in D . We will start this discussion by defining terms and functions used in the determinizing process.

Given a conformant planning problem $\langle I, D, G \rangle$, a fluent proposition f **in** V is said to be a *nondeterministic fluent proposition* if in domain D there occurs an effect proposition

$$f \text{ in } V \text{ when } \phi$$

such that $|V| > 1$.

For a nondeterministic fluent proposition f **in** V in D , f is said to be a *nondeterministic fluent* in D .

Here it is important to note that it can be the case that for some nondeterministic fluent g , there exist nondeterministic fluent propositions g **in** W and g **in** W' such that $W \neq W'$. This is the reason why we determinize nondeterministic fluent propositions, in contrast to the case of determinizing the initial state, where we determinize multiple initial fluents. (Recall that there can be only one fluent proposition d **in** W in I for a given multiple initial fluent d .)

Let g **in** W be a nondeterministic fluent proposition in D .

For any fluent f and $w \in W$, define

$$\text{extend}'(f \text{ in } V, w) = \begin{cases} f.w = w & , \text{ if } f = g \text{ and } V = W \\ \text{extend}(f \text{ in } V, w) & , \text{ otherwise} \end{cases} \quad (3.17)$$

$$\text{extend}'(f \text{ in } V \text{ when } \phi, w) = \text{extend}'(f \text{ in } V, w) \text{ when } \text{extend}(\phi, w) \quad (3.18)$$

$$\text{extend}'(E, w) = \{ \text{extend}'(f \text{ in } V \text{ when } \phi, w) : f \text{ in } V \text{ when } \phi \in E \} \quad (3.19)$$

For W define

$$\text{extend}'(f \text{ in } V \text{ when } \phi, W) = \{ \text{extend}'(f \text{ in } V \text{ when } \phi, w) : w \in W \} \quad (3.20)$$

$$\text{extend}'(E, W) = \bigcup_{w \in W} \text{extend}'(E, w) \quad (3.21)$$

For any conformant problem $Q = \langle I, D, G \rangle$ and any nondeterministic fluent proposition g **in** W in D , the result of determinizing g **in** W is $Q' = \langle I', D', G' \rangle$, defined as follows. The set of fluents in Q' is the set of

fluents in Q that are not in A_g (defined as in Section 3.3) along with $|W|$ copies of each fluent in A_g . More precisely, the fluents of Q' are

- the fluents of D not in A_g
- and the fluents $f.w$ such that $f \in A_g$ and $w \in W$.

As before, the domain of a new fluent $f.w$ in D' is the same as the domain of f in D . In addition, I' , D' and G' are obtained as shown in Figure 3.8. Here one should notice the similarities with determinizing a multiple initial fluent in I (as shown in Figure 3.7).

For any conformant problem $Q = \langle I, D, G \rangle$ and any nondeterministic fluent proposition g in W in D , the result of determinizing g in W is $Q' = \langle I', D', G' \rangle$, where

$$I' = \text{extend}(I, W)$$

$$G' = \text{extend}(G, W), \text{ and}$$

for each operator a in D of the form

Operator: a

Precondition: Pre

Effects: E

we include in D' the corresponding operator

Operator: a

Precondition: extend(Pre, W)

Effects: extend'(E, W)

Figure 3.8: The result of determinizing a conformant problem $Q = \langle I, D, G \rangle$ and any nondeterministic fluent proposition g in W in D .

3.5 Termination

In this section we will provide lemmas that show termination of the determinizing procedure.

Lemma 3.5.1. *For a conformant problem $\langle I, D, G \rangle$ where D is copyfree, the procedure of determinizing all multiple initial fluents in I , one at a time, terminates.*

Lemma 3.5.2. *For a conformant problem $\langle I, D, G \rangle$ where D is copyfree, the procedure of determinizing all nondeterministic fluent propositions in D , one at a time, terminates.*

For the interested reader the proof of Lemma 3.5.1 is found in Section 3.14. The proof of Lemma 3.5.2 is left as future work.

3.6 Uniformity

In this section we present formal definitions of uniformity and related concepts. These concepts are required when determinizing nondeterministic fluent propositions (as stated in Theorem 2).

Definition 3.6.1. *An extended state s' is uniform on a fluent proposition f in V in D if for every $w, w' \in W$*

$$s' \cdot w \models f \text{ in } V \text{ iff } s' \cdot w' \models f \text{ in } V .$$

Definition 3.6.2. *An extended state s' is uniform on ϕ in D if s' is uniform on every fluent proposition f in V that occurs in ϕ .*

Stated otherwise, s' is uniform on a fluent formula ϕ iff all the original states encoded in s' agree on the truth or falsity of ϕ and all of its subformulas.

Definition 3.6.3. *An extended state s' is uniform on domain D if s' is uniform on every fluent precondition in D .*

Uniformity on D allows for consistency in the effects of actions between the states of the epistemic history of the original world and the epistemic history of the determinized world.

For convenience we will say a state s' is uniform meaning that s' is uniform on D . This will be clear in context.

Definition 3.6.4. *An action domain D is uniformity preserving if for every extended state s' that is uniform on D and action a in D' , if $tr(s', a, s'_a)$ then s'_a is uniform on D .*

For convenience we will say that a domain D preserves uniformity to mean that D is uniformity preserving.

Definition 3.6.5. I' is uniform if s' is uniform for all $s' \in Mod(I')$.

Having these definitions at hand makes our theory “usable”.

3.7 Adequacy

Currently it is not clear how to determine if a domain D preserves uniformity in general and so we introduce a syntactic condition, called “adequacy”, on D that guarantees it, as defined by Definition 3.7.1. In addition, the relationship between adequacy and “uniformity preserving” is presented as Theorem 3. Finally, to complete this discussion we show that the determinizing procedure preserves adequacy (Section 3.8).

Definition 3.7.1. An action domain D is adequate if for every nondeterministic fluent proposition g in W , for every effect proposition f in V when ϕ in D , if $f \in A_g$, then

for any fluent precondition ψ in which f in V' occurs, $V \subseteq V'$ or $V \cap V' = \emptyset$.

For the interested reader, the proof of Theorem 3, showing that adequate action domains preserve uniformity, can be found in Section 3.11.

To see adequacy in action, consider the conformant problem described in Figure 3.3, which cannot be determinized. We will show that D is not adequate. To this end, take nondeterministic fluent proposition a in $\{1, 2\}$. By definition of A_a , $a \in A_a$. But, $a = 1$ occurs in the fluent precondition of the effect proposition b when $a = 1$ and neither $\{1, 2\} \subseteq \{1\}$ nor $\{1, 2\} \cap \{1\} = \emptyset$. Thus we can conclude that D is not adequate.

3.8 Preservation under determinizing

It can be the case that a conformant problem contains more than one multiple initial fluent or nondeterministic fluent proposition, in which case our algorithm dictates the application of the determinizing procedure multiple times. For this reason, we need to make sure that *copyfree*, adequacy, uniformity preserving, and

uniformity are preserved under determinizing, as stated by the following lemmas. However, due to time constraints the formulation and proof of the lemma of preservation of uniformity is left as future work.

Lemma 3.8.1. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree and adequate, if $\langle I', D', G' \rangle$ is obtained by determinizing a nondeterministic fluent proposition g in W , then D' is adequate.*

Lemma 3.8.2. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree and adequate, if $\langle I', D', G' \rangle$ is obtained by determinizing a multiple initial fluent d , then D' is adequate.*

Conjecture 3.8.1. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree and uniformity preserving, if $\langle I', D', G' \rangle$ is obtained by determinizing a nondeterministic fluent proposition g in W , then D' is uniformity preserving.*

Lemma 3.8.3. *For any conformant problem $\langle I, D, G \rangle$ where D is copyfree and d a multiple initial fluent in I , if we obtain $\langle I', D', G' \rangle$ by determinizing d , then D' is copyfree.*

Lemma 3.8.4. *For any conformant problem $\langle I, D, G \rangle$ where D is copyfree and g in W is a nondeterministic fluent proposition in D , if we obtain $\langle I', D', G' \rangle$ by determinizing g in W , then D' is copyfree.*

For the interested reader, the proofs of the above lemmas can be found in Section 3.13.

Correctness of the determinizing procedure is guaranteed as follows. For any conformant planning problem $\langle I, D, G \rangle$,

- Check that D is copyfree
- Check adequacy of D
- Determinize all multiple initial fluents, one at a time
- Determinize all nondeterministic fluent propositions, one at a time, while checking uniformity of I' at each step

3.9 Proof of Theorem 1

The proof in this section is modelled after Chen's corresponding proof.

First we define several functions that will aid in the proof. Let s be a state and E a set of effect propositions.

$$\text{change}(s, E) = \{f \text{ in } V : f \text{ in } V \text{ when } \phi \in E, s \models \phi\} \quad (3.22)$$

If E is the effect of an action that is executable in s , then $\text{change}(s, E)$ is a partial state description. (This follows easily from the definition of an action, in light of Restriction 2.1.1.)

Next define

$$\text{affected}(s, E) = \{f : f \text{ in } V \in \text{change}(s, E)\} \quad (3.23)$$

$$\text{result}(s, E) = \{f = s(f) : f \notin \text{affected}(s, E)\} \cup \text{change}(s, E) \quad (3.24)$$

When an action a with effect E is executable in a state s , $\text{result}(s, E)$ is a partial state description. (This follows easily from the definitions of a partial state description and of an operator.) In addition, if E is deterministic, then $\text{result}(s, E)$ is a state.

Fact 3.9.1. *For any states s, s' and any action a with effect E that is executable in s ,*

$$\text{tr}(s, a, s') \text{ iff } s' \models \text{result}(s, E).$$

Proof. Left-to-right direction. Assume that $\text{tr}(s, a, s')$ for states s, s' of D and action a of D . We need to show $s' \models \text{result}(s, E)$, where E is the effect of a . By definition of $\text{tr}(s, a, s')$, for each fluent f ,

- if there is an effect proposition $f \text{ in } V \text{ when } \phi \in E$ s.t. $s \models \phi$,
then $s' \models f \text{ in } V$, and
- if there is no effect proposition $f \text{ in } V \text{ when } \phi \in E$ s.t. $s \models \phi$,
then $s'(f) = s(f)$.

This is equivalent to the requiring that

$$s' \models \text{change}(s, E) \text{ and } s' \models \{f = s(f) : f \notin \text{affected}(s, E)\},$$

which is in turn equivalent to requiring that $s' \models \text{result}(s, E)$.

Right-to-left direction. Similar to above case.

□

We say that ψ is a *fluent precondition* in D if it is the antecedent of some effect in D .

For a partial state description P , define $Mod(P)$ as the set of all the states that satisfy P . That is,

Definition 3.9.1. $Mod(P) = \{s : s \models P\}$.

Next, we present several helpful lemmas, of which the proofs are found in Section 3.9.1.

Lemma 3.9.1. For any state s in D' , $w \in W$ and any fluent formula ϕ ,
 $s \models extend(\phi, w)$ iff $s \cdot w \models \phi$.

This lemma carries over to the case of partial state descriptions in addition to fluent formulas, since a partial state description can be thought of as a fluent formula, specifically, the conjunction of the fluent propositions in the partial state description.

Next we will define the notion of *correspondence*. We say a state s in D' *corresponds* to the set of states $\{s \cdot w : w \in W\}$ in D and write

$$s \sim \{s \cdot w : w \in W\} .$$

From the definition of correspondence we obtain Lemma 3.9.2 as follows:

Lemma 3.9.2. If $s \sim S$ then $s \models extend(\phi, W)$ iff $S \models \phi$.

Again, this result holds for partial state descriptions in addition to fluent formulas.

For any set of states S in D' , we say S *corresponds* to the set of states

$$\{s \cdot w : s \in S, w \in W\}$$

in D and write

$$S \sim \{s \cdot w : s \in S, w \in W\} .$$

Lemma 3.9.3. If $S' \sim S$ then $S' \models extend(\phi, W)$ iff $S \models \phi$

Again, this result also holds for partial state descriptions.

Lemma 3.9.4. $Mod(extend(P, W)) \sim Mod(P)$

Up until now in the proof we have defined the functions *change*, *affected*, and *result*. We have shown that if $tr(s, a, s')$ then $s' \models result(s, E)$. Furthermore, we defined the notion of correspondance and proved several lemmas that provide a link between a state in the determinized world and its corresponding state in the original world. These concepts will help us in defining and proving several lemmas that will be used to prove Theorem 1. This will be done next.

To this end, first we need to prove that after determinizing an initial multiple fluent, we still get operators in D' . That is, we need to verify that the operators in D' obey Restriction 2.1.1 (defined in chapter 2) and presented here as a reminder.

Restriction. *For any f in V when ϕ and f in V' when ψ in E such that $V \neq V'$, for any state s , if s satisfies Pre then s satisfies at most one of ϕ and ψ .*

We will show this by contradiction. Let $\langle I', D', G' \rangle$ be the result of determinizing $\langle I, D, G \rangle$ with respect to a multiple initial fluent. Let a be a “non-operator” in D' with effect propositions

$$extend(f \text{ in } V, w) \text{ when } extend(\phi, w)$$

and

$$extend(f \text{ in } V', w) \text{ when } extend(\psi, w)$$

with $V \neq V'$ such that $s' \models extend(\phi, w)$ and $s' \models extend(\psi, w)$ for some s' of D' . By Lemma 3.9.1, $s' \cdot w \models \phi$ and $s' \cdot w \models \psi$. By the determinizing procedure, there is an operator a in D with $f \text{ in } V \text{ when } \phi$ and $f \text{ in } V \text{ when } \psi$ in E . Thus it follows that a is a non-operator in D since it does not obey Restriction 2.1.1 and thus $\langle I, D, G \rangle$ is not a well-formed conformant problem. Hence we have our contradiction. We can conclude that the result of determinizing an initial fluent is indeed a well-formed conformant problem.

Now that we have the above lemmas, we will proceed to prove Theorem 1. We will do this by showing the following:

1. $Mod(I') \sim Mod(I)$. That is we show that the set of states that satisfy the extended initial state description correspond to the set of states that satisfy the original initial state description.
2. This correspondance is maintained by actions. That is for any action a , set of states S in D , and set of states S' in D' such that $S' \sim S$, then a is executable in S' iff it is executable in S . Furthermore, if we have $tr(S', a, S'_a)$ in D' and $tr(S, a, S_a)$ in D , then $S'_a \sim S_a$.

3. At the last action we will have that $S'_{n+1} \sim S_{n+1}$ and so

$$S'_{n+1} \models \text{extend}(G, W) \text{ iff } S_{n+1} \models G.$$

To this end, several lemmas will be presented of which the proofs are shown in Section 3.9.1. Then, these lemmas will be used to prove Theorem 1.

Lemma 3.9.5. $Mod(I') \sim Mod(I)$.

Lemma 3.9.6. If $S' \sim S$, then action a is executable in S' iff a is executable in S .

Lemma 3.9.7. If $S' \sim S$, $tr(S', a, S'_a)$ and $tr(S, a, S_a)$ then $S'_a \sim S_a$.

Now that we have these lemmas in hand, we will proceed with the proof.

Proof of Theorem 1. Left-to-Right Direction. Consider a solution for $\langle I', D', G' \rangle$ of size n ,

$A^* = \langle a_1, a_2, \dots, a_n \rangle$. This corresponds to an epistemic history $\langle S'_1, a_1, S'_2, \dots, a_n, S'_{n+1} \rangle$ of size $n + 1$ in D' such that $S'_1 = Mod(I')$ and $S'_{n+1} \models G'$. It is enough to show that there is an epistemic history in D , $\langle S_1, a_1, S_2, \dots, a_n, S_{n+1} \rangle$ such that $S_1 = Mod(I)$ and $S_{n+1} \models G$. By Lemma 3.9.5, $Mod(I') \sim Mod(I)$ and thus $S'_1 \sim S_1$. Since $S'_1 \sim S_1$ and a_1 is executable in S'_1 then, by Lemma 3.9.6, a_1 is executable in S_1 . Take S_2 such that $tr(S_1, a_1, S_2)$. Then by Lemma 3.9.7 $S'_2 \sim S_2$. This construction continues until the end where we get that $S'_{n+1} \sim S_{n+1}$. Since, $S'_{n+1} \models G'$ and $S'_{n+1} \sim S_{n+1}$, then, by Lemma 3.9.1, $S_{n+1} \models G$ which means that A^* is a solution for $\langle I, D, G \rangle$.

Right-to-Left Direction. Similar to above case. □

This completes the proof of Theorem 1.

3.9.1 Proof of lemmas for Theorem 1

Lemma 3.9.1. For any state s' in D' , $w \in W$ and any fluent formula ϕ ,

$$s' \models \text{extend}(\phi, w) \text{ iff } s' \cdot w \models \phi$$

Proof. Using structural induction on the definition of a fluent formula.

Let $P(d)$ stand for: $s' \models \text{extend}(d, w)$ iff $s' \cdot w \models d$.

- case 1: $P(f \text{ in } V)$.
 - $s' \models \text{extend}(f \text{ in } V, w)$
 - iff $s' \models \text{extend}(f, w) \text{ in } V$ [by def. of extend]
 - iff $s'(\text{extend}(f, w)) \in V$ [by def. of satisfaction]
 - iff $s' \cdot w(f) \in V$
 - iff $s' \cdot w \models f \text{ in } V$ [by def. of satisfaction]
- case 2: $P(\top)$.
 - $s' \models \text{extend}(\top, w)$
 - iff $s' \models \top$ [def. extend]
 - iff $s' \cdot w \models \top$.
- case 3: $P(\perp)$.

Similar to $P(\top)$.
- case 4: ϕ is a fluent formula
 - IH: $P(\phi)$
 - Need to show $P(\neg\phi)$
 - $s' \models \text{extend}(\neg\phi, w)$
 - iff $s' \models \neg\text{extend}(\phi, w)$ [def. extend]
 - iff $s' \not\models \text{extend}(\phi, w)$
 - iff $s' \cdot w \not\models \phi$ [IH]
 - iff $s' \cdot w \models \neg\phi$
- case 5: ϕ, ψ are fluent formulas
 - IH: $P(\phi), P(\psi)$
 - Need to show $P(\phi \odot \psi)$
 - $P(\phi \wedge \psi)$
 - $s' \models \text{extend}(\phi \wedge \psi, w)$
 - iff $s' \models \text{extend}(\phi, w) \wedge \text{extend}(\psi, w)$ [def. extend]
 - iff $s' \models \text{extend}(\phi, w)$ and $s' \models \text{extend}(\psi, w)$
 - iff $s' \cdot w \models \phi$ and
 - $s' \cdot w \models \psi$ [IH]
 - iff $s' \cdot w \models \phi \wedge \psi$

– $P(\phi \vee \psi)$

Similar to $P(\phi \wedge \psi)$.

– $P(\phi \supset \psi)$

Similar to $P(\phi \wedge \psi)$.

– $P(\phi \equiv \psi)$

Similar to $P(\phi \wedge \psi)$.

□

Lemma 3.9.2. *If $s' \sim S$ then $s' \models \text{extend}(\phi, W)$ iff $S \models \phi$*

Proof. $s' \models \text{extend}(\phi, W)$

iff $s' \models \bigwedge_{w \in W} \text{extend}(\phi, w)$

iff for all $w \in W$, $s' \models \text{extend}(\phi, w)$

iff for all $w \in W$, $s' \cdot w \models \phi$ [Lemma 3.9.1]

iff $\{s' \cdot w : w \in W\} \models \phi$

iff $S \models \phi$

□

Lemma 3.9.3. *If $S' \sim S$ then $S' \models \text{extend}(\phi, W)$ iff $S \models \phi$*

Proof. $S' \models \text{extend}(\phi, W)$

iff for all $s' \in S'$, $s' \models \text{extend}(\phi, W)$

iff for all $s' \in S'$, $\{s' \cdot w : w \in W\} \models \phi$ [Lemma 3.9.2]

iff $\{s' \cdot w : s' \in S', w \in W\} \models \phi$

iff $S \models \phi$

□

Lemma 3.9.4. $\text{Mod}(\text{extend}(P, W)) \sim \text{Mod}(P)$

Proof. Need to show that

$$\text{Mod}(P) = \{s' \cdot w : s' \in \text{Mod}(\text{extend}(P, W)), w \in W\}.$$

Left-to-Right direction. Take $s \in Mod(P)$. We need to show that there is a $s' \in Mod(extend(P, W))$ and $w \in W$ such that $s = s' \cdot w$. Consider s' such that

$$s'(extend(f, w)) = s(f) \text{ for all } f \text{ and } w \in W.$$

Then, $s = s' \cdot w$ for all $w \in W$. Since $s \models P$, then by Lemma 3.9.1, we get that

$$s' \models extend(P, w) \text{ for all } w \in W.$$

By definition of *extend*, it follows that $s' \models extend(P, W)$

and so $s' \in Mod(extend(P, W))$.

Right-to-Left direction. Take any $s' \in Mod(extend(P, W))$ and $w \in W$. We need to show that $s' \cdot w \models P$. Since $s' \models extend(P, W)$, then $s' \models extend(P, w)$. By Lemma 3.9.1, $s' \cdot w \models P$. □

Lemma 3.9.5. $Mod(I') \sim Mod(I)$

Proof. We need to show that

$$Mod(I) = \{s' \cdot w : s' \in Mod(I'), w \in W\}.$$

By Lemma 3.9.4, $Mod(extend(I, W)) \sim Mod(I)$, which is to say that

$$Mod(I) = \{s' \cdot w : s' \in Mod(extend(I, W)), w \in W\}.$$

So it is sufficient to show that

$$\{s' \cdot w : s' \in Mod(extend(I, W)), w \in W\} = \{s' \cdot w : s' \in Mod(I'), w \in W\}.$$

Since $Mod(I') \subseteq Mod(extend(I, W))$, it is enough to show that

$$\{s' \cdot w : s' \in Mod(extend(I, W)), w \in W\} \subseteq \{s' \cdot w : s' \in Mod(I'), w \in W\}.$$

In particular, we need to show that, for any $s' \in Mod(extend(I, W)) - Mod(I')$ and any $w \in W$, there is an $s'' \in Mod(I')$ and a $w' \in W$ such that $s'' \cdot w' = s' \cdot w$. So take such an s' and w . Then take s'' agrees with s' everywhere except for the fluents $d.w''$: for these take $s''(d.w'') = w''$. To complete the proof we need to show that

$$s'' \in Mod(I') \text{ and } s'' \cdot w' = s' \cdot w.$$

First let us show that

$$s'' \in Mod(I').$$

To this end we need to show that

$$s'' \models I'.$$

Take any $extend(f \text{ in } V, w_I) \in I'$ for some $w_I \in W$.

Consider 2 cases:

1. $f = d$. Then, by definition of I' ,

$$extend(f \text{ in } V, w_I) = f.w_I = w_I.$$

But, by construction of s'' ,

$$s''(d.w_I) = w_I.$$

Therefore, $s'' \models extend(f \text{ in } V, w_I)$.

2. Otherwise. Then $extend(f \text{ in } V, w_I) \in extend(I - \{d \text{ in } W\}, W)$.

But $s' \in mod(extend(I, W))$ and so

$$s' \models extend(f \text{ in } V, w_I).$$

This implies that

$$s'(extend(f, w_I)) \in V,$$

and so, by construction of s'' ,

$$s''(extend(f, w_I)) \in V.$$

Since $extend(f \text{ in } V, w_I)$ is an arbitrary fluent proposition in I' , it follows that $s'' \models I'$, thus $s'' \in Mod(I')$.

Next we need to show that

$$s'' \cdot w' = s' \cdot w.$$

Let w' be such that

$$s'(extend(d, w)) = w'.$$

Take any fluent f in D . We need to show that

$$s'' \cdot w'(f) = s' \cdot w.$$

Consider 2 cases:

1. $f = d$. Then by construction of s'' ,

$$s''(\text{extend}(d, w')) = w'.$$

That is

$$s'' \cdot w'(d) = w'.$$

But, $s'(\text{extend}(d, w)) = w'$ and so

$$s' \cdot w(d) = w'.$$

2. Otherwise. Then, by construction of s'' ,

$$s''(\text{extend}(f, w')) = s'(\text{extend}(f, w)).$$

It follows that

$$s'' \cdot w'(f) = s' \cdot w(f).$$

Since f is an arbitrary fluent, we conclude that

$$s'' \cdot w' = s' \cdot w.$$

This completes the proof. □

Lemma 3.9.6. *If $S' \sim S$, then action a is executable in S' iff a is executable in S .*

Proof. Assume $S' \sim S$.

Let Pre be the action precondition of a in D . Then $\text{extend}(Pre, W)$ is the action precondition of a in D' .

By Lemma 3.9.3, $S' \models \text{extend}(Pre, W)$ iff $S \models Pre$. □

Lemma 3.9.7. *For all states s' of D' , all effects E of D , all $w \in W$, and all fluent propositions f in V in the language of D ,*

$$\text{extend}(f \text{ in } V, w) \in \text{change}(s', \text{extend}(E, W))$$

iff

$$f \text{ in } V \in \text{change}(s' \cdot w, E).$$

Proof. Left-to-right direction. Assume that

$$\text{extend}(f \text{ in } V, w) \in \text{change}(s', \text{extend}(E, W))$$

for some state s' of D' , effect E of D , $w \in W$ and fluent proposition $f \text{ in } V$ in the language of D . We need to show that

$$f \text{ in } V \in \text{change}(s' \cdot w, E).$$

By definition of *change* and our assumption, there is an effect proposition

$$\text{extend}(f \text{ in } V, w) \text{ when } \text{extend}(\phi, w) \in \text{extend}(E, W)$$

such that $s' \models \text{extend}(\phi, w)$. Note that

$$\text{extend}(f \text{ in } V, w) \text{ when } \text{extend}(\phi, w)$$

can be written as

$$\text{extend}(f \text{ in } V \text{ when } \phi, w).$$

Since this effect proposition belongs to $\text{extend}(E, W)$, we know that $f \text{ in } V \text{ when } \phi$ belongs to E . And since $s' \models \text{extend}(\phi, w)$, we know by Lemma 3.9.1 that $s' \cdot w \models \phi$. We can conclude that $f \text{ in } V \in \text{change}(s' \cdot w, E)$, which is what needed to be shown.

Right-to-left direction. Similar to above case. □

Lemma 3.9.8. For all states s' in D' , all effects E of D , all $w \in W$, and all fluents f of D ,

$$\text{extend}(f, w) \in \text{affected}(s', \text{extend}(E, W))$$

iff

$$f \in \text{affected}(s' \cdot w, E).$$

Proof. Follows immediately from lemma 3.9.7. □

Lemma 3.9.9. For all states s' of D' , all effects E of D , all $w \in W$ and all fluent propositions $f \text{ in } V$ in the language of D ,

$$\text{extend}(f \text{ in } V, w) \in \text{result}(s', \text{extend}(E, W))$$

iff

$$f \text{ in } V \in \text{result}(s' \cdot w, E).$$

Proof. Left-to-right direction. Assume that

$$\text{extend}(f \text{ in } V, w) \in \text{result}(s', \text{extend}(E, W))$$

for some state s' of D' , effect E of D , $w \in W$ and fluent proposition $f \text{ in } V$ in the language of D . We need to show that

$$f \text{ in } V \in \text{result}(s' \cdot w, E).$$

By definition of *result* and our assumption,

$$\text{extend}(f \text{ in } V, w) \in \text{change}(s', \text{extend}(E, W))$$

or

$$\text{extend}(f \text{ in } V, w) \in s', \text{extend}(f, w) \notin \text{affected}(s', \text{extend}(E, W)).$$

By Lemma 3.9.7,

$$f \text{ in } V \in \text{change}(s' \cdot w, E)$$

or

$$f \text{ in } V \in s' \cdot w, f \notin \text{affected}(s' \cdot w, E).$$

By the definition of *result* we can conclude that

$$f \text{ in } V \in \text{result}(s' \cdot w, E).$$

This is what was needed to be shown.

Right-to-left direction. Similar to above case. □

Lemma 3.9.10. *If $S' \sim S$, $tr(S', a, S'_a)$, $tr(S, a, S_a)$ and $s_a \in S_a$, then there is a $s'_a \in S'_a$ and a $w \in W$ such that $s_a = s'_a \cdot w$.*

Proof. Since we have that $s_a \in S_a$ and that $tr(S, a, S_a)$, then there is an $s \in S$ such that $tr(s, a, s_a)$. Since $S' \sim S$ and $s \in S$ then there is an $s' \in S'$ and a $w \in W$ such that $s = s' \cdot w$. Now consider $result(s', extend(E, W))$. We know that this is a PSD and that there is at least one state that satisfies $result(s', extend(E, W))$. Consider a state s'' that satisfies $result(s', extend(E, W))$. Construct s'_a as follows:

$$\begin{aligned} s'_a(f \cdot w') &= s''(f \cdot w') && \text{if } f \in A_d, w' \in W - \{w\} \\ s'_a(extend(f, w)) &= s_a(f) && \text{otherwise} \end{aligned}$$

From the above construction, we can observe that $s_a = s'_a \cdot w$. Next, it is sufficient to show that $s'_a \in S'_a$, by showing that s'_a satisfies

$result(s', extend(E, W))$. Consider any fluent proposition $extend(f \text{ in } V, w')$ in $result(s', extend(E, W))$ where $w' \in W$. We have two cases:

1. $f \in A_d$ and can be written as $f \cdot w'$ where $w' \neq w$. In this case $s'_a(f \cdot w') = s''(f \cdot w')$, by construction of s'_a . But $s'' \models extend(E, W)$ and thus $s'_a \models f \cdot w'$
2. All other f such that $w' = w$ or $f \notin A_d$. In this case, the fluent proposition can be written as $extend(f, w) \text{ in } V$ and $s'_a(extend(f, w)) = s_a(f)$. It is enough to show that $s_a \models f \text{ in } V$. From Lemma 3.9.9, $f \text{ in } V$ belongs to $result(s' \cdot w, E)$. But $s = s' \cdot w$ and so $f \text{ in } V$ belongs to $result(s, E)$. Since we have $tr(s, a, s_a)$ then $s_a \models result(s, E)$. It follows that $s_a \models f \text{ in } V$.

□

Lemma 3.9.11. *If $S' \sim S$, $tr(S', a, S'_a)$, $tr(S, a, S_a)$ and $s_a = s'_a \cdot w$ for some $s'_a \in S'_a$ and $w \in W$, then $s_a \in S_a$*

Proof. Since $s'_a \in S'_a$ and $tr(S', a, S'_a)$, then there is an $s' \in S'$ such that $tr(s', a, s'_a)$. Let $s = s' \cdot w$, then $s \in S$ since $s' \in S'$ and $S' \sim S$. We can show that $s_a \in S_a$ by showing that $s_a \models result(s, E)$. Consider any $f \text{ in } V$ in $result(s, E)$. By Lemma 3.9.9, $extend(f \text{ in } V, w)$ is in $result(s', extend(E, W))$. Since $tr(s', a, s'_a)$, then $s'_a \models result(s', extend(E, W))$ and consequently $s'_a \models extend(f \text{ in } V, w)$. It follows by Lemma 3.9.1 that

$$s'_a \cdot w \models f \text{ in } V.$$

But $s_a = s'_a \cdot w$ and so $s_a \models f$ in V . Since we considered an arbitrary fluent proposition in $result(s, E)$ it follows that $s_a \models result(s, E)$. □

Lemma 3.9.12. *If $S' \sim S$, $tr(S', a, S'_a)$ and $tr(S, a, S_a)$ then $S'_a \sim S_a$.*

Proof. Assume $S' \sim S$, $tr(S', a, S'_a)$ and $tr(S, a, S_a)$. We need to show that

$$S'_a \sim S_a .$$

To do this, it is enough to show that

$$S_a = \{s'_a \cdot w : s'_a \in S'_a, w \in W\} .$$

Consider two cases:

1. Consider an arbitrary $s_a \in S_a$. By Lemma 3.9.10, there is an $s'_a \in S'_a$ and a $w \in W$ such that $s_a = s'_a \cdot w$,
2. Consider an arbitrary $s_a = s'_a \cdot w$ for some $s'_a \in S'_a$ and $w \in W$. We need to show that $s_a \in S_a$. By Lemma 3.9.11, $s_a \in S_a$.

We have shown set inclusion in both directions and thus the equality holds. □

3.10 Proof of Theorem 2

To prove Theorem 2, we will follow the same approach as used to prove Theorem 1: we will state several definitions and lemmas and then show the proof based on these.

For conformant problem $\langle I, D, G \rangle$, let g in W be a nondeterministic fluent proposition in D and $\langle I', D', G' \rangle$ be the result of determinizing g in W .

Definition 3.10.1. An extended state s' approximates s with respect to w , denoted

$$s' \cdot w \approx s,$$

if for all fluents $f \neq g$, $s(f) = s' \cdot w(f)$.

Definition 3.10.2. A set of extended states S' is full with respect to a set of states S , denoted $S' \rightarrow S$, if for all $s \in S$, there are $s' \in S'$ and $w' \in W$ such that $s' \cdot w' = s$ and for all $w \in W$, $s' \cdot w \approx s$.

Next we will present several lemmas that will be used to prove Theorem 2.

Lemma 3.10.1. $Mod(extend(I, W)) \rightarrow Mod(I)$.

Lemma 3.10.2. If $S' \sim S$, then action a is executable in S' iff a is executable in S .

Lemma 3.10.3. If D is uniformity preserving, S' is uniform, $S' \sim S$, $S' \rightarrow S$, $tr(S', a, S'_a)$, $tr(S, a, S_a)$ then $S'_a \sim S_a$.

Lemma 3.10.4. If $S' \rightarrow S$, $tr(S, a, S_a)$, $tr(S', a, S'_a)$ and S' is uniform then $S'_a \rightarrow S_a$.

Having the above lemmas at our disposal we will next prove Theorem 2.

Proof of Theorem 2. Left-to-Right direction. Consider a solution for $\langle I', D', G' \rangle$ of size n , $A^* = \langle a_1, a_2, \dots, a_n \rangle$. This corresponds to an epistemic history

$$\langle S'_1, a_1, S'_2, \dots, a_n, S'_{n+1} \rangle$$

of size $n + 1$ in D' such that $S'_1 = mod(I')$ and $S'_{n+1} \models G'$.

It is enough to show that there is an epistemic history in D ,

$$\langle S_1, a_1, S_2, \dots, a_n, S_{n+1} \rangle,$$

such that $S_1 = Mod(I)$ and $S_{n+1} \models G$. By Lemma 3.9.4 $Mod(I') \sim Mod(I)$ and so $S'_1 \sim S_1$. Furthermore, by Lemma 3.10.1, $Mod(extend(I, W)) \rightarrow Mod(I)$ and so $S'_1 \rightarrow S_1$. Since a_1 is executable in S'_1 then, by Lemma 3.10.2, a_1 is executable in S_1 . Take S_2 such that $tr(S_1, a, S_2)$. By Lemma 3.10.3 $S'_2 \sim S_2$. Furthermore, by Lemma 3.10.4, $S'_2 \rightarrow S_2$. This construction continues until the end where we have that $S'_{n+1} \sim S_{n+1}$. It follows, by Lemma 3.9.1, that $S_{n+1} \models G$ which means that $A^* = \langle a_1, a_2, \dots, a_n \rangle$ is a solution for $\langle I, D, G \rangle$.

Right-to-Left direction. Similar to above case.

This completes the proof of Theorem 2. □

3.10.1 Proof of lemmas for Theorem 2

Let $\langle I_0, D_0, G_0 \rangle, \dots, \langle I_n, D_n, G_n \rangle$ be planning problems where $\langle I_0, D_0, G_0 \rangle = \langle I, D, G \rangle$ and for all $i = 0, \dots, n-1$, $\langle I_{i+1}, D_{i+1}, G_{i+1} \rangle$ is obtained by determinizing a multiple initial fluent, d_i , in $\langle I_i, D_i, G_i \rangle$.

Lemma 3.10.5. $Mod(extend(I, W)) \rightarrow Mod(I)$.

Proof. Take any $s \in Mod(I)$. We need to show that there are $s' \in Mod(extend(I, W))$ and $w_s \in W$ such that

$$s' \cdot w_s = s \text{ and for all } w \in W, s' \cdot w \approx s.$$

Construct s' as follows:

For all fluents f and all $w \in W$

$$s'(extend(f, w)) = s(f).$$

From the construction of s' we have that

$$s' \cdot w = s \text{ for all } w \in W.$$

It remains only to show that $s' \models extend(I, W)$. Take any

$$extend(f \text{ in } V, w) \in extend(I, W).$$

Then, by definition of $extend$, $f \text{ in } V \in I$. Since $s \models I$, then $s \models f \text{ in } V$ and therefore by Lemma 3.9.1, $s' \models extend(f \text{ in } V, w)$. Since $extend(f \text{ in } V, w)$ is arbitrary, it follows that

$$s' \models extend(I, W).$$

□

Lemma 3.10.6. *If $S' \sim S$, then action a is executable in S' iff a is executable in S .*

Proof. Let a be an operator in D such that a is executable in S . By the procedure of determinizing a nondeterministic fluent proposition, there is an operator a in D' with the precondition $extend(Pre, W)$. Since $S \models Pre$ and $S' \sim S$, then, by Lemma 3.9.3, $S' \models extend(Pre, W)$. It follows that a is executable in S' . □

Lemma 3.10.7. *For any extended state s' , any $w \in W$ and any fluent proposition f in V such that $f \neq g$ or $V \neq W$*

$$s' \models extend'(f \text{ in } V, w) \text{ iff } s' \cdot w \models f \text{ in } V.$$

Proof. For all $w \in W$, if $f \neq g$ or $V \neq W$,

$$extend'(f \text{ in } V, w) = extend(f \text{ in } V, w).$$

Then, by Lemma 3.9.1,

$$s' \models extend'(f \text{ in } V, w) \text{ iff } s' \cdot w \models f \text{ in } V.$$

□

Lemma 3.10.8. *For any extended state s' that is uniform and any $w \in W$, if*

$$f \text{ in } V \in change(s' \cdot w, E)$$

then for all $w' \in W$,

$$f \text{ in } V \in change(s' \cdot w', E).$$

Proof. Assume $f \text{ in } V$ in $change(s' \cdot w, E)$. By definition of *change*, there is an effect proposition

$$f \text{ in } V \text{ when } \phi$$

in E such that

$$s' \cdot w \models \phi.$$

By Lemma 3.9.1, $s' \models \text{extend}(\phi, w)$. Since s' is uniform then

$$s' \models \text{extend}(\phi, w') \text{ for all } w' \in W .$$

It follows that $f \text{ in } V \in \text{change}(s' \cdot w', E)$ for all $w' \in W$. □

Lemma 3.10.9. *For any extended state s' that is uniform and any $w \in W$, if*

$$f \in \text{affected}(s' \cdot w, E)$$

then for all $w' \in W$,

$$f \in \text{affected}(s' \cdot w', E) .$$

Proof. Follows immediately from Lemma 3.10.8. □

Lemma 3.10.10. *If extended state s' is uniform and $f \in \text{affected}(s' \cdot w, E)$*

then if $f \text{ in } V \in \text{result}(s' \cdot w, E)$ then for all $w' \in W$, $f \text{ in } V \in \text{result}(s' \cdot w', E)$.

Proof. Let s' be an extended state that is uniform and f be a fluent such that $f \in \text{affected}(s' \cdot w, E)$.

Assume $f \text{ in } V \in \text{result}(s' \cdot w, E)$. Need to show that for all $w' \in W$, $f \text{ in } V \in \text{result}(s' \cdot w', E)$.

From our assumption and choice of f ,

$$f \text{ in } V \in \text{change}(s' \cdot w, E) .$$

By Lemma 3.10.8,

$$f \text{ in } V \in \text{change}(s' \cdot w', E) \text{ for all } w' \in W .$$

Then, by definition of *result*

$$f \text{ in } V \in \text{result}(s' \cdot w', E) \text{ for all } w' \in W .$$

This is what was needed to be shown. □

Lemma 3.10.11. *If extended state s' is uniform and $f \notin \text{affected}(s' \cdot w, E)$*

then for all $w' \in W$, $f = s' \cdot w'(f) \in \text{result}(s' \cdot w', E)$.

Proof. Let s' be an extended state that is uniform and f be a fluent such that $f \notin \text{affected}(s' \cdot w, E)$. Need to show that for all $w' \in W$, $f = s' \cdot w'(f) \in \text{result}(s' \cdot w', E)$.

By Lemma 3.10.9,

$$f \notin \text{affected}(s' \cdot w', E) \text{ for all } w' \in W.$$

Then, by definition of *result*

$$f = s' \cdot w'(f) \in \text{result}(s' \cdot w', E) \text{ for all } w' \in W.$$

This is what was needed to be shown. □

Lemma 3.10.12. *If $S' \rightarrow S$ and S' is uniform then for any $s \in S$, there is a $s' \in S'$, such that if $f \mathbf{in} V \in \text{result}(s, E)$ and $f \neq g$ then*

$$\text{for all } w \in W, f \mathbf{in} V \in \text{result}(s' \cdot w, E).$$

Proof. Since $s \in S$ and $S' \rightarrow S$ then there are an $s' \in S'$ and $w_s \in W$ such that

$$s' \cdot w_s = s \text{ and for all } w \in W s' \cdot w \approx s.$$

Assume that $f \mathbf{in} V \in \text{result}(s, E)$. That is $f \mathbf{in} V \in \text{result}(s' \cdot w_s, E)$. It remains to prove that for all $w \in W$, $f \mathbf{in} V \in \text{result}(s' \cdot w, E)$.

Since $f \mathbf{in} V \in \text{result}(s' \cdot w_s, E)$ then either $f \in \text{affected}(s' \cdot w_s, E)$ or it is not. We will consider these two cases separately and show that our goal is achieved in both cases. This then completes the proof.

Let $w \in W$. Consider 2 cases:

1. $f \in \text{affected}(s' \cdot w_s, E)$. Then, since

$$f \mathbf{in} V \in \text{result}(s' \cdot w_s, E)$$

and s' is uniform then, by Lemma 3.10.10,

$$f \mathbf{in} V \in \text{result}(s' \cdot w, E).$$

2. $f \notin \text{affected}(s' \cdot w_s, E)$. Then $V = \{s' \cdot w_s(f)\}$. Since s' is uniform then by Lemma 3.10.11,

$$f = s' \cdot w(f) \in \text{result}(s' \cdot w, E).$$

But $s' \cdot w \approx s$ and so

$$\begin{aligned} s' \cdot w(f) &= s(f) \\ &= s' \cdot w_s(f) \end{aligned}$$

Thus, $f \text{ in } V \in \text{result}(s' \cdot w, E)$.

Since w is arbitrary then this completes the proof. □

Lemma 3.10.13. *For all extended states s' , all effects E of D , all $w \in W$, and all fluent propositions $f \text{ in } V$ in the language of D , the following hold.*

- (i) *If $f \text{ in } V \in \text{change}(s' \cdot w, E)$,
then $\text{extend}'(f \text{ in } V, w) \in \text{change}(s', \text{extend}'(E, W))$.*
- (ii) *If $\text{extend}'(f \text{ in } V, w) \in \text{change}(s', \text{extend}'(E, W))$ and
($f \neq g$ or ($V \neq W$ and $V \neq \{w\}$)),
then $f \text{ in } V \in \text{change}(s' \cdot w, E)$.*
- (iii) *If $\text{extend}'(g \text{ in } W, w) \in \text{change}(s', \text{extend}'(E, W))$, then exactly one of
 $g \text{ in } \{w\}$ and $g \text{ in } W$ belong to $\text{change}(s' \cdot w, E)$.*

Proof. (i) Assume $f \text{ in } V \in \text{change}(s' \cdot w, E)$. Need to show that

$$\text{extend}'(f \text{ in } V, w) \in \text{change}(s', \text{extend}'(E, W)).$$

Since $f \text{ in } V \in \text{change}(s' \cdot w, E)$, there is an effect proposition

$f \text{ in } V \text{ when } \phi \text{ in } E$ such that $s' \cdot w \models \phi$. It follows that

$$\text{extend}'(f \text{ in } V \text{ when } \phi, w) \in \text{extend}'(E, W).$$

But, $\text{extend}'(f \text{ in } V \text{ when } \phi, w)$ can be written as

$$\text{extend}'(f \text{ in } V, w) \text{ when } \text{extend}(\phi, w).$$

Since $s' \cdot w \models \phi$ then, by Lemma 3.9.1, $s' \models \text{extend}(\phi, w)$ and thus, by definition of *change*,

$$\text{extend}'(f \text{ in } V, w) \in \text{change}(s', \text{extend}'(E, W)).$$

(ii) Assume $f \text{ in } V$ not $g \text{ in } W$ and $f \text{ in } V$ not $g \text{ in } \{w\}$ for $w \in W$ and that $\text{extend}'(f \text{ in } V, w)$ belongs to $\text{change}(s', \text{extend}'(E, W))$. Need to show that $f \text{ in } V$ belongs to $\text{change}(s' \cdot w, E)$.

From our assumptions, there is an effect proposition

$$\text{extend}'(f \text{ in } V, w) \text{ when } \text{extend}(\phi, w) \in \text{extend}'(E, W)$$

such that

$$s' \models \text{extend}(\phi, w).$$

By Lemma 3.9.1,

$$s' \cdot w \models \phi.$$

Then, since $f \text{ in } V$ when $\phi \in E$, it follows that

$$f \text{ in } V \in \text{change}(s' \cdot w, E).$$

(iii) Assume $\text{extend}'(g \text{ in } W, w) \in \text{change}(s', \text{extend}'(E, W))$. Need to show that exactly one of $g \text{ in } \{w\}$ and $g \text{ in } W$ belong to $\text{change}(s' \cdot w, E)$.

By definition of extend' , $\text{extend}'(g \text{ in } W, w)$ is $g.w \text{ in } \{w\}$. By definition of change and our assumption, the effect proposition

$$g.w \text{ in } \{w\} \text{ when } \text{extend}(\phi, w)$$

belongs to $\text{extend}'(E, W)$, and $s' \models \text{extend}(\phi, w)$. So either $g \text{ in } W$ when ϕ or $g \text{ in } \{w\}$ when ϕ must belong to E . Moreover, at most one of them does; otherwise D would violate Restriction 2.1.1. Since $s' \models \text{extend}(\phi, w)$ then, by Lemma 3.9.1, $s' \cdot w \models \phi$ and we conclude that exactly one of $g \text{ in } W$ and $g \text{ in } \{w\}$ belongs to $\text{change}(s' \cdot w, E)$. \square

Lemma 3.10.14. For all states s' in D' , all effects E of D , all $w \in W$ and all fluents f of D ,

$$\text{extend}(f, w) \in \text{affected}(s', \text{extend}'(E, W))$$

iff

$$f \in \text{affected}(s' \cdot w, E).$$

Proof. Follows immediately from Lemma 3.10.13. \square

Lemma 3.10.15. *For all states s' of D' , all effects E of D , all $w \in W$ and all fluent propositions f in V in the language of D , the following hold.*

(i) *If f in $V \in \text{result}(s' \cdot w, E)$,*

then $\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W))$.

(ii) *If $\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W))$ and*

($f \neq g$ or ($V \neq W$ and $V \neq \{w\}$)),

then f in $V \in \text{result}(s' \cdot w, E)$.

(iii) *If $\text{extend}'(g \text{ in } W, w) \in \text{result}(s', \text{extend}'(E, W))$, then exactly one of*

g in $\{w\}$ and g in W belong to $\text{result}(s' \cdot w, E)$.

Proof. Part (i) Assume f in $V \in \text{result}(s' \cdot w, E)$.

Need to show that

$$\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W)).$$

From our assumption and definition of *result*, we have 2 cases:

1. f in $V \in \text{change}(s' \cdot w, E)$.

By Lemma 3.10.13,

$$\text{extend}'(f \text{ in } V, w) \in \text{change}(s', \text{extend}'(E, W)).$$

It follows that,

$$\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W)).$$

2. f in $V \notin \text{change}(s' \cdot w, E)$.

Then

$$f \notin \text{affected}(s' \cdot w, E)$$

and

$$f = s' \cdot w(f) \in \text{result}(s' \cdot w, E).$$

By Lemma 3.10.14,

$$\text{extend}(f, w) \notin \text{affected}(s', \text{extend}'(E, W)).$$

Then by definition of *result*,

$$\text{extend}(f, w) = s'(\text{extend}(f, w)) \in \text{result}(s', \text{extend}'(E, W)).$$

We need to show that $\text{extend}(f, w) = s'(\text{extend}(f, w))$ is the same as $\text{extend}'(f \text{ in } V, w)$. By definition of *extend*, $\text{extend}(f, w) = s'(\text{extend}(f, w))$ is the same as

$$\text{extend}(f \text{ in } \{s'(\text{extend}(f, w))\}, w).$$

Since $\text{result}(s' \cdot w, E)$ is a PSD and also contains $f \text{ in } V$, we conclude that

$$V = \{s' \cdot w(f)\}.$$

Since W is not a singleton then

$$\{s' \cdot w(f)\} \neq W.$$

So

$$\begin{aligned} \text{extend}(f \text{ in } \{s' \cdot w(f)\}, w) &= \text{extend}'(f \text{ in } \{s' \cdot w(f)\}, w) \\ &= \text{extend}'(f \text{ in } V, w) \end{aligned}$$

This is what was needed to be shown.

Part (ii) Assume $\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W))$ and

$$(f \neq g \text{ or } (V \neq W \text{ and } V \neq \{w\})).$$

We need to show that $f \text{ in } V \in \text{result}(s' \cdot w, E)$.

By definition of *result* we have two cases:

1. $\text{extend}'(f \text{ in } V, w) \in \text{change}(s', \text{extend}'(E, W))$.

By Lemma 3.10.13,

$$f \text{ in } V \in \text{change}(s' \cdot w, E)$$

and so it follows that

$$f \text{ in } V \in \text{result}(s' \cdot w, E).$$

2. $extend'(f \text{ in } V, w) \notin change(s', extend'(E, W))$.

Then $extend(f, w) \notin affected(s', extend'(E, W))$.

By Lemma 3.10.14,

$$f \notin affected(s' \cdot w, E).$$

Then

$$f \text{ in } \{s' \cdot w(f)\} \in result(s' \cdot w, E).$$

To complete the proof we need to show that

$$V = \{s' \cdot w(f)\}.$$

Since

$$extend'(f \text{ in } V, w) \in result(s', extend'(E, W))$$

and

$$extend'(f \text{ in } V, w) \notin change(s', extend'(E, W))$$

then $V = \{s'(extend(f, w))\}$. That is $V = \{s' \cdot w(f)\}$.

This is what was needed to be shown.

Part (iii) Assume $extend'(g \text{ in } W, w) \in result(s', extend'(E, W))$. Need to show that exactly one of

$$g \text{ in } \{w\} \text{ and } g \text{ in } W \text{ belong to } result(s' \cdot w, E).$$

From our assumption and definition of *result*, we have 2 cases:

1. $extend'(g \text{ in } W, w) \in change(s', extend'(E, W))$.

By Lemma 3.10.13, exactly one of

$$g \text{ in } \{w\} \text{ and } g \text{ in } W \text{ belong to } change(s' \cdot w, E).$$

It follows that exactly one of

$$g \text{ in } \{w\} \text{ and } g \text{ in } W \text{ belong to } result(s' \cdot w, E).$$

2. $extend'(g \text{ in } W, w) \in s'$, $extend(g, w) \notin affected(s', extend'(E, W))$.

From our assumptions and Lemma 3.10.14, $g \notin affected(s' \cdot w, E)$. From the definition of $extend'$,

$$extend'(g \text{ in } W, w) = g \cdot w = w$$

and from our assumption we have that

$$s'(extend(g, w)) = w .$$

That is,

$$s' \cdot w(g) = w .$$

Since $w \in W$, it follows that exactly one of

$$g \text{ in } \{w\} \text{ and } g \text{ in } W \text{ in } s' \cdot w .$$

Thus exactly one of

$$g \text{ in } \{w\} \text{ and } g \text{ in } W \text{ belong to } result(s' \cdot w, E) .$$

Having proven the above two cases, completes the proof.

□

Lemma 3.10.16. *Assume that D is uniformity preserving. If S' is uniform, $S' \sim S$, $S' \rightarrow S$, $tr(S', a, S'_a)$, $tr(S, a, S_a)$, and $s_a \in S_a$, then for some $s'_a \in S'_a$ and $w \in W$, $s_a = s'_a \cdot w$.*

Proof. Since $tr(S, a, S_a)$ and $s_a \in S_a$,

$$tr(s, a, s_a) \text{ for some } s \in S .$$

Since $S' \sim S$, there are $s' \in S'$ and $w_s \in W$ such that

$$s = s' \cdot w_s .$$

The proof is organized as follows: First we will construct s'_a such that

$$s_a = s'_a \cdot w \text{ for some } w \in W .$$

Then we will show that

$$s'_a \models \text{result}(s', \text{extend}'(E, W)).$$

Since $s' \in S'$, it then follows that $s'_a \in S'_a$ and this completes the proof.

Consider $\text{result}(s, E)$.

We have 2 cases:

1. $g \text{ in } W \notin \text{result}(s, E)$.

Let s''_a be such that

$$s''_a \models \text{result}(s', \text{extend}'(E, W)). \quad (3.25)$$

We know that s''_a exists because $\text{result}(s', \text{extend}'(E, W))$ is a PSD and therefore there is at least one state that satisfies it.

Construct s'_a as follows:

$$s'_a(\text{extend}(f, w)) = \begin{cases} s''_a(\text{extend}(f, w)) & \text{if } w \neq w_s, f \in A_g \\ s_a(f) & \text{otherwise} \end{cases}$$

By construction of s'_a

$$s_a = s'_a \cdot w_s.$$

Next we need to show that

$$s'_a \models \text{result}(s', \text{extend}'(E, W)).$$

Take any $\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W))$.

Consider 2 cases:

- (a) $w \neq w_s, f \in A_g$.

Follows from (3.25) and construction of s'_a in terms of s''_a .

- (b) Otherwise.

By construction of s'_a

$$s'_a(\text{extend}(f, w_s)) = s_a(f).$$

Since $\text{tr}(s, a, s_a)$ then $s_a \models \text{result}(s, E)$.

We have 3 cases:

i. $w = w_s$ and $f \notin A_g$.

Then $f \neq g$ and so by Lemma 3.10.15(ii) it follows that

$$f \text{ in } V \in \text{result}(s, E).$$

Since $s_a \models f \text{ in } V$, then $s_a(f) \in V$. So $s'_a(\text{extend}(f, w)) \in V$. Thus

$$s'_a \models \text{extend}(f, w) \text{ in } V.$$

But by definition of *extend*

$$\text{extend}(f \text{ in } V, w) = \text{extend}(f, w) \text{ in } V.$$

And since $f \neq g$ then

$$\text{extend}'(f \text{ in } V, w) = \text{extend}(f \text{ in } V, w).$$

It then follows that

$$s'_a \models \text{extend}'(f \text{ in } V, w).$$

ii. $w \neq w_s$ and $f \notin A_g$.

Similar to above case.

iii. $w = w_s$ and $f \in A_g$.

If $f \neq g$, then we have a case similar to (i).

Otherwise, $f = g$. By Lemma 3.10.15(iii) it follows that exactly one of

$$g \text{ in } \{w\} \text{ and } g \text{ in } W \text{ belong to } \text{result}(s, E).$$

From our assumptions, we know that $g \text{ in } W \notin \text{result}(s, E)$ and so we conclude that

$$V = \{w\}.$$

We know that $s_a \models g \text{ in } \{w\}$ and so $s_a(g) = w$. Then, by construction of s'_a , we have that $s'_a(\text{extend}(g, w)) = w$. But,

$$\begin{aligned} \text{extend}'(g \text{ in } \{w\}) &= \text{extend}(g \text{ in } \{w\}) \\ &= \text{extend}(g, w) \text{ in } \{w\} \end{aligned}$$

So it follows that $s'_a \models \text{extend}'(f \text{ in } V, w)$.

2. $g \text{ in } W \in \text{result}(s, E)$.

Since $\text{result}(s', \text{extend}'(E, W))$ is a PSD then there is at least one state that satisfies it. Let s''_a be such a state. That is

$$s''_a \models \text{result}(s', \text{extend}'(E, W)).$$

Let w_{s_a} be $s_a(g)$. Since $g \text{ in } W \in \text{result}(s, E)$ and $\text{tr}(s, a, s_a)$ then $s_a \models g \text{ in } W$. Then $s_a(g) \in W$.

That is $w_{s_a} \in W$. Construct s'_a as follows:

$$s'_a(\text{extend}(f, w)) = \begin{cases} s''_a(\text{extend}(f, w)) & \text{if } w \neq w_{s_a}, f \in A_g \\ w & \text{if } f = g \\ s_a(f) & \text{Otherwise} \end{cases}$$

By construction of s'_a and choice of w_{s_a} ,

$$s_a = s'_a \cdot w_{s_a}.$$

Next we need to show that

$$s'_a \models \text{result}(s', \text{extend}'(E, W)).$$

Take any $\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W))$.

Consider the following cases:

- $w \neq w_{s_a}, f \in A_g$.

Follows by construction of s'_a with respect to s''_a .

- $f = g$.

Since $g \text{ in } W \in \text{result}(s, E)$ and $s = s' \cdot w_s$, then

$$g \text{ in } W \in \text{result}(s' \cdot w_s, E).$$

Since W is not a singleton then

$$g \text{ in } W \in \text{change}(s' \cdot w_s, E)$$

which implies that $f \in \text{affected}(s' \cdot w_s, E)$. It then follows, by Lemma 3.10.10, that

$$g \text{ in } W \in \text{result}(s' \cdot w, E).$$

Then, by Lemma 3.10.15(i)

$$\text{extend}'(g \text{ in } W, w) \in \text{result}(s', \text{extend}'(E, W)).$$

Thus, we conclude that $V = W$. That is $f \text{ in } V$ is $g \text{ in } W$. By definition of extend' ,

$$\text{extend}'(f \text{ in } V, w) = f.w = w.$$

But, by construction of s'_a ,

$$s'_a(\text{extend}(f, w)) = w$$

and so it follows that

$$s'_a \models \text{extend}'(f \text{ in } V, w).$$

- Otherwise. Then $f \neq g$.

By Lemma 3.10.15(ii),

$$f \text{ in } V \in \text{result}(s'.w, E).$$

Since $S' \rightarrow S$, $s \in S$ and $f \neq g$ then, by Lemma 3.10.12,

$$f \text{ in } V \in \text{result}(s'.w_s, E).$$

Then, since $\text{tr}(s, a, s_a)$ and $s = s'.w_s$ it follows that $s_a \models f \text{ in } V$. That is $s_a(f) \in V$. By construction of s'_a ,

$$s'_a(\text{extend}(f, w)) = s_a(f)$$

and so $s'_a(\text{extend}(f, w)) \in V$. Then $s'_a \models \text{extend}(f \text{ in } V, w)$. Since $f \neq g$ then, by definition of extend' ,

$$\text{extend}(f \text{ in } V, w) = \text{extend}'(f \text{ in } V, w).$$

Thus, it follows that

$$s'_a \models \text{extend}'(f \text{ in } V, w).$$

Since $\text{extend}'(f \text{ in } V, w)$ is arbitrary it then follows that

$$s'_a \models \text{result}(s', \text{extend}'(E, W)).$$

□

Lemma 3.10.17. *If $S' \sim S$, $tr(S', a, S'_a)$, $tr(S, a, S_a)$, then for all $s'_a \in S'_a$ and $w \in W$,*

$$s'_a \cdot w \in S_a.$$

Proof. Take any $s'_a \in S'_a$, $w \in W$. Then, since $tr(S', a, S'_a)$, there is a $s' \in S'$ such that

$$tr(s', a, s'_a).$$

Let

$$s = s' \cdot w$$

and

$$s_a = s'_a \cdot w$$

for $w \in W$. Since $S' \sim S$, then $s \in S$.

We need to show that $s_a \in S_a$. To this extent we will show that

$$tr(s, a, s_a).$$

By definition of tr , this implies that

$$s_a \in S_a.$$

Consider $result(s, E)$ where E is the effect of a in D . By showing that

$$s_a \models result(s, E)$$

it follows that $tr(s, a, s_a)$ which is what was needed to be shown.

Take any $f \mathbf{in} V \in result(s, E)$. By Lemma 3.10.15,

$$extend'(f \mathbf{in} V, w) \in result(s', extend'(E, W)).$$

Since $tr(s', a, s'_a)$, then $s'_a \models result(s', extend'(E, W))$.

This then implies that $s'_a \models extend'(f \mathbf{in} V, w)$.

Consider 2 cases as follows:

- $f = g$ and $V = W$.

By definition, $extend'(f \mathbf{in} V, w) = f \cdot w = w$. Since

$$s'_a \models extend'(f \mathbf{in} V, w)$$

then

$$s'_a(\text{extend}(f, w)) = w.$$

That is

$$s'_a \cdot w(f) = w.$$

Since $s_a = s'_a \cdot w$ and $w \in W$, it follows that

$$s_a \models f \text{ in } V.$$

- Otherwise. By definition of *extend'* and *extend*

$$\begin{aligned} \text{extend}'(f \text{ in } V, w) &= \text{extend}(f \text{ in } V, w) \\ &= \text{extend}(f, w) \text{ in } V. \end{aligned}$$

But

$$s'_a \models \text{extend}'(f \text{ in } V, w)$$

and so

$$s'_a \models \text{extend}(f, w) \text{ in } V.$$

This implies that

$$s'_a(\text{extend}(f, w)) \in V.$$

That is

$$s'_a \cdot w(f) \in V.$$

It follows that

$$s_a(f) \in V.$$

This is what was needed to be shown.

Since we chose an arbitrary $f \text{ in } V \in \text{result}(s, E)$, then

$$s_a \models \text{result}(s, E)$$

and so it follows that $s_a \in S_a$. □

Lemma 3.10.18. *If D is uniformity preserving, S' is uniform, $S' \sim S$, $S' \rightarrow S$, $\text{tr}(S', a, S'_a)$, $\text{tr}(S, a, S_a)$ then $S'_a \sim S_a$.*

Proof. We need to show that

$$S_a = \{s'_a \cdot w : s'_a \in S'_a, w \in W\}.$$

Consider 2 cases:

- Consider an arbitrary $s_a \in S_a$. By Lemma 3.10.16, there is an $s'_a \in S'_a$ and a $w \in W$ such that

$$s_a = s'_a \cdot w.$$

- Consider an arbitrary $s'_a \cdot w$ for some $s'_a \in S'_a$ and $w \in W$. By Lemma 3.10.17

$$s'_a \cdot w \in S_a.$$

We have shown set inclusion in both directions and thus the equality holds. □

Lemma 3.10.19. *If $f \notin \text{affected}(s, E)$ and $\text{tr}(s, a, s_a)$ then $s(f) = s_a(f)$.*

Proof. Since $f \notin \text{affected}(s, E)$, then by definition of *result*,

$$f \text{ in } \{s(f)\} \in \text{result}(s, E).$$

Since $\text{tr}(s, a, s_a)$, then

$$s_a \models \text{result}(s, E)$$

and therefore

$$s_a \models f \text{ in } \{s(f)\}.$$

It thus follows that

$$s_a(f) = s(f).$$

□

Lemma 3.10.20. *For any extended state s' , every fluent proposition in*

$$\text{result}(s', \text{extend}'(E, W))$$

can be written in the form

$$\text{extend}'(f \text{ in } V, w).$$

Proof. Take any $extend(f \text{ in } V, w) \in result(s', extend'(E, W))$. If

$$extend(f \text{ in } V, w) = extend'(f \text{ in } V, w)$$

then we are done. Otherwise, $f \text{ in } V$ must be $g \text{ in } W$. Then

$$extend(f \text{ in } V, w) \in change(s', extend'(E, W))$$

since W is not a singleton. It follows, by definition of *change*, that there is an effect proposition

$$extend(f \text{ in } V, w) \text{ when } extend(\phi, w) \in extend'(E, W)$$

such that $s' \models extend(\phi, w)$. But,

$$extend(f \text{ in } V, w) \text{ when } extend(\phi, w)$$

can be written as

$$extend(f \text{ in } V \text{ when } \phi, w).$$

Since E is a set of effect propositions, then by definition of $extend'$,

$$extend'(E, W) = \bigcup_{w \in W} extend'(E, w).$$

It follows that

$$extend(f \text{ in } V \text{ when } \phi, w) = extend'(f \text{ in } V' \text{ when } \phi, w)$$

for some V' . But

$$extend'(f \text{ in } V' \text{ when } \phi, w) = extend'(f \text{ in } V', w) \text{ when } extend(\phi, w).$$

It thus follows that $extend(f \text{ in } V, w) = extend'(f \text{ in } V', w)$. □

Lemma 3.10.21. *If $S' \rightarrow S$, $tr(S, a, S_a)$, $tr(S', a, S'_a)$ and S' is uniform then $S'_a \rightarrow S_a$.*

Proof. Let $s_a \in S_a$. Need to show that there are $s'_a \in S'_a$, $w_{s_a} \in W$ such that

$$s'_a \cdot w_{s_a} = s_a \text{ and for all } w \in W, s'_a \cdot w \approx s_a.$$

Since $s_a \in S_a$, $tr(S, a, S_a)$ then there is an $s \in S$ such that $tr(s, a, s_a)$. Since $S' \rightarrow S$, then there are an $s' \in S'$ and $w_s \in W$ such that

$$s' \cdot w_s = s \text{ and for all } w \in W, s' \cdot w \approx s.$$

Next we will split our proof into 2 cases based on whether

$$g \text{ in } W \in result(s' \cdot w_s, E)$$

or not. We will show in each case that the goal of the lemma holds. That is $S'_a \rightarrow S_a$.

1. $g \text{ in } W \in result(s' \cdot w_s, E)$. Construct s'_a as follows:

$$s'_a(extend(f, w)) = \begin{cases} s_a(f) & \text{if } f \neq g, f \in affected(s' \cdot w_s, E) \\ w & \text{if } f = g \\ s'(extend(f, w)) & \text{Otherwise} \end{cases}$$

Let w_{s_a} be $s_a(g)$. Since $tr(s, a, s_a)$ and $s' \cdot w_s = s$, then $s_a \models result(s' \cdot w_s, E)$. So in this case $s_a \models g \text{ in } W$ and so we conclude that $s_a(g) \in W$. That is $w_{s_a} \in W$. It remains to show that $s'_a \in S'_a$ and that

$$s'_a \cdot w_{s_a} = s_a \text{ and for all } w \in W, s'_a \cdot w \approx s_a. \quad (3.26)$$

First we will show (3.26). Observe that $s'_a \cdot w_{s_a}(g) = s_a(g)$. It then remains to show that for all $f \neq g$ and $w \in W$, $s'_a \cdot w(f) = s_a(f)$. Consider any fluent $f \neq g$ and $w \in W$. Consider 2 cases as follows:

- (a) $f \in affected(s' \cdot w_s, E)$. Then $s'_a(extend(f, w)) = s_a(f)$ by construction of s'_a . That is $s'_a \cdot w(f) = s_a(f)$.
- (b) $f \notin affected(s' \cdot w_s, E)$. Then, since $tr(s, a, s_a)$ then by Lemma 3.10.19 $s(f) = s_a(f)$. But $s' \cdot w \approx s$ and so $s' \cdot w(f) = s_a(f)$. That is $s'(extend(f, w)) = s_a(f)$. But, by construction of s'_a ,

$$s'_a(extend(f, w)) = s'(extend(f, w))$$

and so it follows that $s'_a \cdot w(f) = s_a(f)$. Next we need to show that $s'_a \in S'_a$. It suffices to show that

$$s'_a \models result(s', extend'(E, W)).$$

Since $s' \in S'$ and $tr(S', a, S'_a)$ then by Lemma 3.10.20 every element of

$$result(s', extend'(E, W))$$

can be written in the form $extend'(f \text{ in } V, w)$ and it remains to show that s'_a satisfies any such $extend'(f \text{ in } V, w)$. Consider 3 cases as follows:

i. $f \neq g, f \in affected(s' \cdot w_s, E)$. Then there is a fluent proposition

$$f \text{ in } V' \in change(s' \cdot w_s, E)$$

which implies that

$$f \text{ in } V' \in result(s' \cdot w_s, E).$$

Then by Lemma 3.10.10

$$f \text{ in } V' \in result(s' \cdot w, E).$$

It follows by Lemma 3.10.15(i) that

$$extend'(f \text{ in } V', w) \in result(s', extend'(E, W))$$

and so we conclude that $V' = V$. Thus $f \text{ in } V \in result(s' \cdot w_s, E)$. That is $f \text{ in } V \in result(s, E)$. Since $tr(s, a, s_a)$ then $s_a \models f \text{ in } V$. That is $s_a(f) \in V$. But, by construction of s'_a ,

$$s'_a(extend(f, w)) = s_a(f)$$

and so $s'_a(extend(f, w)) \in V$ which implies that

$$s'_a \models extend(f \text{ in } V, w).$$

Since $f \neq g$ then

$$extend(f \text{ in } V, w) = extend'(f \text{ in } V, w).$$

So $s'_a \models extend'(f \text{ in } V, w)$.

ii. $f = g$. Since $g \text{ in } W \in result(s' \cdot w_s, E)$ then

$$g \text{ in } W \in change(s' \cdot w_s, E),$$

since W is not a singleton. Then by Lemma 3.10.8,

$$g \text{ in } W \in \text{change}(s' \cdot w, E).$$

Then $g \text{ in } W \in \text{result}(s' \cdot w, E)$. Then, by Lemma 3.10.15(i), it follows that

$$\text{extend}'(g \text{ in } W, w) \in \text{result}(s', \text{extend}'(E, W)).$$

So we conclude that $V = W$. By construction of s'_a , $s'_a(\text{extend}(g, w)) = w$ and since $\text{extend}'(g \text{ in } W, w)$ is $g \cdot w = w$, then $s'_a \models \text{extend}'(g \text{ in } W, w)$. That is

$$s'_a \models \text{extend}'(f \text{ in } W, w).$$

iii. Otherwise. Then $f \neq g$, $f \notin \text{affected}(s' \cdot w_s, E)$. Then, by Lemma 3.10.11,

$$f = s' \cdot w(f) \in \text{result}(s' \cdot w, E).$$

From Lemma 3.10.15(i),

$$\text{extend}'(f = s' \cdot w(f), w) \in \text{result}(s', \text{extend}'(E, W)).$$

Thus $V = \{s' \cdot w(f)\}$. We need to show that

$$s'_a(\text{extend}(f, w)) = s' \cdot w(f).$$

By construction of s'_a , $s'_a(\text{extend}(f, w)) = s'(\text{extend}(f, w))$. That is

$$s'_a(\text{extend}(f, w)) = s' \cdot w(f).$$

2. $g \text{ in } W \notin \text{result}(s' \cdot w_s, E)$. Construct s'_a as follows:

$$s'_a(\text{extend}(f, w)) = \begin{cases} s_a(f) & \text{if } f \in \text{affected}(s' \cdot w_s, E) \\ s'(\text{extend}(f, w)) & \text{Otherwise} \end{cases}$$

Let $w_{s_a} = w_s$. Then $w_{s_a} \in W$ since $w_s \in W$. It remains to show that $s'_a \in S'_a$ and that

$$s'_a \cdot w_{s_a} = s_a \text{ and for all } w \in W, s'_a \cdot w \approx s_a. \quad (3.27)$$

First we will show (3.27). We will do this by showing

- (a) $s'_a(\text{extend}(g, w_{s_a})) = s_a(g)$ and
 (b) for any fluent $f \neq g$ and $w \in W$,

$$s'_a(\text{extend}(f, w)) = s_a(f).$$

For (a), we need to show that $s'_a(\text{extend}(g, w_{s_a})) = s_a(g)$. If $g \in \text{affected}(s' \cdot w_s, E)$, then, by construction of s'_a , $s'_a(\text{extend}(g, w_{s_a})) = s_a(g)$.

Otherwise, $g \notin \text{affected}(s' \cdot w_s, E)$ and since $\text{tr}(s, a, s_a)$, then by Lemma 3.10.19, $s(g) = s_a(g)$. But $s = s' \cdot w_s$ and so $s(g) = s' \cdot w_s(g)$. Thus $s' \cdot w_s(g) = s_a(g)$. That is $s'(\text{extend}(g, w_s)) = s_a(g)$. But $w_{s_a} = w_s$ and so

$$s'(\text{extend}(g, w_{s_a})) = s_a(g).$$

But by construction of s'_a , $s'_a(\text{extend}(g, w_{s_a})) = s'(\text{extend}(g, w_{s_a}))$ and so

$$s'_a(\text{extend}(g, w_{s_a})) = s_a(g).$$

For (b) we need to show that for any fluent $f \neq g$ and $w \in W$,

$$s'_a(\text{extend}(f, w)) = s_a(f).$$

This proof is similar to case $g \text{ in } W \in \text{result}(s' \cdot w_s, E)$ above.

Next we need to show that $s'_a \in S'_a$. Since $s' \in S'$, then it suffices to show that

$$s'_a \models \text{result}(s', \text{extend}'(E, W)).$$

By Lemma 3.10.20 every element of

$$\text{result}(s', \text{extend}'(E, W))$$

can be written in the form $\text{extend}'(f \text{ in } V, w)$ and it remains to show that s'_a satisfies any such $\text{extend}'(f \text{ in } V, w)$. Consider 3 cases as follows:

- (a) $f \neq g, f \in \text{affected}(s' \cdot w_s, E)$. This proof is similar to case $g \text{ in } W \in \text{result}(s' \cdot w_s, E)$ above.
 (b) $f = g$.
- If $f \notin \text{affected}(s' \cdot w_s, E)$ then by Lemma 3.10.11,

$$f = s' \cdot w(f) \in \text{result}(s' \cdot w, E)$$

and by Lemma 3.10.15(i)

$$\text{extend}'(f = s' \cdot w(f), w) \in \text{result}(s', \text{extend}'(E, W)).$$

Therefore, $V = \{s' \cdot w(f)\}$. By construction of s'_a , $s'_a(\text{extend}(f, w)) = s'(\text{extend}(f, w))$. That is $s'_a(\text{extend}(f, w)) = s' \cdot w(f)$. Therefore $s'_a \models \text{extend}(f \text{ in } V, w)$. Since $V \neq W$ then

$$\text{extend}(f \text{ in } V, w) = \text{extend}'(f \text{ in } V, w)$$

and so it follows that

$$s'_a \models \text{extend}'(f \text{ in } V, w).$$

- If $f \in \text{affected}(s' \cdot w_s, E)$ then there is a fluent proposition

$$f \text{ in } V' \in \text{change}(s' \cdot w_s, E)$$

which implies that

$$f \text{ in } V' \in \text{result}(s' \cdot w_s, E).$$

Then $V' \neq W$ because $g \text{ in } W \notin \text{result}(s' \cdot w_s, E)$.

Since $f \text{ in } V' \in \text{result}(s' \cdot w_s, E)$ then by Lemma 3.10.10

$$f \text{ in } V' \in \text{result}(s' \cdot w, E).$$

Then, by Lemma 3.10.15(i),

$$\text{extend}'(f \text{ in } V', w) \in \text{result}(s', \text{extend}'(E, W))$$

and so we conclude that $V' = V$. Thus $V \neq W$. Furthermore, $f \text{ in } V \in \text{result}(s' \cdot w_s, E)$.

That is $f \text{ in } V \in \text{result}(s, E)$. Since $\text{tr}(s, a, s_a)$, then $s_a \models f \text{ in } V$. But by construction of s'_a ,

$$s'_a(\text{extend}(f, w)) = s_a(f)$$

and since $s_a(f) \in V$ then $s'_a(\text{extend}(f, w)) \in V$. Thus $s'_a \models \text{extend}(f \text{ in } V, w)$. Since $V \neq W$ then

$$\text{extend}(f \text{ in } V, w) = \text{extend}'(f \text{ in } V, w)$$

and so $s'_a \models \text{extend}'(f \text{ in } V, w)$.

(c) Otherwise. This proof is similar to case g in $W \in \text{result}(s' \cdot w_s, E)$ above.

Since $\text{extend}'(f \text{ in } V, w)$ is arbitrary it follows that

$$s'_a \models \text{result}(s', \text{extend}'(E, W)).$$

□

3.11 Proof of Theorem 3

Proof of Theorem 3. Assume D is adequate. We need to show that D preserves uniformity. Take any uniform extended state s' , action a and extended state s'_a such that $\text{tr}(s', a, s'_a)$. Next we need to prove that s'_a is uniform. To this end we need to show that for any fluent precondition ψ , s'_a is uniform on ψ . That is for any $w, w' \in W$

$$s'_a \models \text{extend}(\psi, w) \text{ iff } s'_a \models \text{extend}(\psi, w').$$

It is sufficient to show that for any $f \text{ in } V'$ in ψ , s'_a is uniform on $f \text{ in } V'$. That is

$$s'_a \cdot w \models f \text{ in } V' \text{ iff } s'_a \cdot w' \models f \text{ in } V'.$$

Since $\text{tr}(s', a, s'_a)$ then $s'_a \models \text{result}(s', \text{extend}'(E, W))$. If $f \notin A_g$, then $s'_a \cdot w(f) = s'_a \cdot w'(f)$. Otherwise $f \in A_g$. Then consider 2 cases:

- $f \notin \text{affected}(s' \cdot w, E)$. Then, by Lemma 3.10.9, $f \notin \text{affected}(s' \cdot w', E)$. It follows by Lemma 3.10.19 that

$$s'_a \cdot w(f) = s' \cdot w(f)$$

and

$$s'_a \cdot w'(f) = s' \cdot w'(f).$$

Then

$$s'_a \cdot w \models f \text{ in } V' \text{ iff } s' \cdot w \models f \text{ in } V' \tag{3.28}$$

and

$$s'_a \cdot w' \models f \text{ in } V' \text{ iff } s' \cdot w' \models f \text{ in } V'. \tag{3.29}$$

Since s' is uniform on ψ , then it is uniform on f in V' and so

$$s' \cdot w \models f \text{ in } V' \text{ iff } s' \cdot w' \models f \text{ in } V'.$$

Then, from (3.28) and (3.29) we conclude that

$$s'_a \cdot w \models f \text{ in } V' \text{ iff } s'_a \cdot w' \models f \text{ in } V'.$$

- $f \in \text{affected}(s' \cdot w, E)$. Then there is an effect proposition f in V when ϕ in E such that $s' \cdot w \models \phi$ and f in $V \in \text{result}(s' \cdot w, E)$. It then follows, by Lemma 3.10.10 that f in $V \in \text{result}(s' \cdot w', E)$. Then, by Lemma 3.10.15(i),

$$\text{extend}'(f \text{ in } V, w') \in \text{result}(s', \text{extend}'(E, W))$$

and

$$\text{extend}'(f \text{ in } V, w) \in \text{result}(s', \text{extend}'(E, W)).$$

Since $\text{tr}(s', a, s'_a)$ then

$$s'_a \models \text{extend}'(f \text{ in } V, w')$$

and

$$s'_a \models \text{extend}'(f \text{ in } V, w).$$

Next we need to show that $s'_a \cdot w \models f$ in V and $s'_a \cdot w' \models f$ in V . Consider the case $s'_a \cdot w \models f$ in V .

We will show this using the following two cases:

1. If $f = g$ and $V = W$ then $\text{extend}'(f \text{ in } V, w)$ is $f \cdot w = w$. Since $s'_a \models \text{extend}'(f \text{ in } V, w)$, then $s'_a(\text{extend}(f, w)) = w$. That is $s'_a \cdot w(f) = w$ and since $w \in W$ then $s'_a \cdot w(f) \in W$ which implies that $s'_a \cdot w \models f$ in V .
2. Otherwise. Then $\text{extend}'(f \text{ in } V, w) = \text{extend}(f \text{ in } V, w)$. But $s'_a \models \text{extend}'(f \text{ in } V, w)$ and so it follows that $s'_a \models \text{extend}(f \text{ in } V, w)$. It follows by Lemma 3.9.1 that $s'_a \cdot w \models f$ in V .

Using the same arguments as above we conclude that $s'_a \cdot w' \models f$ in V . Then, since D is adequate either $V \subseteq V'$ or $V \cap V' = \emptyset$. Either way

$$s'_a \cdot w \models f \text{ in } V' \text{ iff } s'_a \cdot w' \models f \text{ in } V'.$$

□

3.12 Properties of Affected Sets

Lemma 3.12.1. *If $extend(f, w') \in A_{extend(g, w)}^{D'}$ then $f \in A_g^D$ where D' is obtained by determinizing D .*

Proof. Using structural induction on the definition of A_g .

Case 1: $extend(f, w') = extend(g, w)$. Then $f = g$ and so $f \in A_g$.

Case 2: IH: $f' \in A_g$.

Then there is an $extend(f', w''') \in A_{extend(g, w)}$ that affects $extend(f, w')$. That is, there is an effect proposition $extend'(f \text{ in } V' \text{ when } \phi, w'')$ in D' such that $extend(f', w''')$ occurs in $extend(\phi, w'')$. By IH, $f' \in A_g$. Then $extend(f', w''') = f'.w'''$ and since $extend(f', w''')$ occurs in $extend(\phi, w'')$ we conclude that $w''' = w''$. That is $extend(f', w''') = extend(f', w'')$ and so by definition of $extend$, f' occurs in ϕ . Since $extend'(f \text{ in } V' \text{ when } \phi, w'')$ in D' then $f \text{ in } V' \text{ when } \phi$ in D . Thus, since $f' \in A_g$ and f' occurs in ϕ then $f \in A_g$. \square

Lemma 3.12.2. *Let D' be obtained from a copyfree domain D by determinizing $g \text{ in } W$. Take any $f \in A_g^D$ and $w \in W$. Then every fluent in $A_{extend(f, w)}^{D'}$ can be written in the form*

$$extend(f', w)$$

where $f' \in A_g^D$.

Proof. Structural induction on $A_{extend(f, w)}^{D'}$.

Base Case: Trivial

Recursive Case: Take any fluent in $A_{extend(f, w)}^{D'}$. By the IH we can write it in the form $extend(f'', w)$ where $f'' \in A_g^D$. Need to show that any fluent affected by $extend(f'', w)$ in D' can be written in the form

$$extend(f', w)$$

where $f' \in A_g^D$.

Consider any fluent affected by $extend(f'', w)$ in D' . Then there is an effect proposition

$$extend'(f' \text{ in } V \text{ when } \phi, w)$$

in D' where $extend(f'', w)$ appears in $extend(\phi, w)$ and the affected fluent is $extend(f', w)$. Next we need to show that $f' \in A_g^D$. Since $extend(f' \text{ in } V \text{ when } \phi, w)$ is in D' then

$$f' \text{ in } V \text{ when } \phi \text{ is in } D. \tag{3.30}$$

3.12. PROPERTIES OF AFFECTED SETS

Since $extend(f'', w)$ appears in $extend(\phi, w)$ then either f'' or $extend(f'', w)$ appear in ϕ . Since $f'' \in A_g^D$ then $extend(f'', w) = f''.w$ and so $extend(f'', w)$ cannot appear in ϕ because D is a *copyfree*. So f'' appears in ϕ and so by $f'' \in A_g^D$ and (3.30), $f' \in A_g^D$. \square

Lemma 3.12.3. *Let D be a copyfree domain.*

Let D_0, D_1, \dots, D_n be a sequence of determinizations of D such that $D_0 = D$ and D_{i+1} is obtained by extending D_i for $0 \leq i < n$. Then for any i ($0 \leq i < n$) for any original fluent f and any distinct copies f', f'' of f in the language of D_i , $f' \notin A_{f''}^{D_i}$.

Proof. Take any original fluent f . Proof by induction:

Let $P(i) =$ For any distinct copies f', f'' of f in D_i , $f' \notin A_{f''}^{D_i}$.

Base Case: $P(0)$. Trivial because D_0 is *copyfree* and so all fluents in D_0 are original (so there can be no distinct copies of a fluent f).

Inductive Case:

IH: $P(i)$.

Need to show $P(i+1)$. Take $extend(f', w'), extend(f'', w'')$ distinct copies of f in the language of D_{i+1} such that f', f'' are fluents in the language of D_i . We need to show that $extend(f', w') \notin A_{extend(f'', w'')}^{D_{i+1}}$. From the definition of copy, it follows that f', f'' are copies of f . Consider 2 cases:

1. $f' \neq f''$. Then, since f' and f'' are copies of f , by the IH $f' \notin A_{f''}^{D_i}$. Then, by Lemma 3.12.1, $extend(f', w') \notin A_{extend(f'', w'')}^{D_{i+1}}$.
2. $f' = f''$. Then $w' \neq w''$. Since $extend(f', w') \neq extend(f'', w'')$ then $f'' \in A_g^{D_i}$ where g in W is a nondeterministic fluent proposition in D_i that was determinized to obtain D_{i+1} . It follows by Lemma 3.12.2 that every fluent in $A_{extend(f'', w'')}^{D_{i+1}}$ can be written in the form

$$extend(f''', w'')$$

where $f''' \in A_g^{D_i}$. But $w' \neq w''$ and so $extend(f', w') \notin A_{extend(f'', w'')}^{D_{i+1}}$.

\square

3.13 Proof of Preservation lemmas

Lemma 3.13.1. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree and adequate, if $\langle I', D', G' \rangle$ is obtained by determinizing a nondeterministic fluent proposition g in W , then D' is adequate.*

Proof. Any nondeterministic fluent proposition in D' can be written in the form $extend'(g' \text{ in } W', w)$ for some nondeterministic fluent proposition $g' \text{ in } W'$ in D and $w \in W$. Any effect proposition in D' can be written in the form $extend'(f \text{ in } V \text{ when } \phi, w')$ for some effect proposition $f \text{ in } V \text{ when } \phi$ in D and $w' \in W$. We need to show that if $extend(f, w') \in A_{extend(g, w)}$, then for any fluent precondition $extend(\psi, w'')$ in D' and $w'' \in W$, if $extend(f \text{ in } V', w')$ occurs in $extend(\psi, w'')$ then $V \subseteq V'$ or $V \cap V' = \emptyset$.

Take any $extend(f \text{ in } V', w')$ in $extend(\psi, w'')$ such that $extend(\psi, w'')$ is a fluent precondition in D' and $extend(f, w') \in A_{extend(g, w)}$. Then, by Lemma 3.12.1, $f \in A_g$ and so $extend(f, w') = f.w'$. Since $extend(f \text{ in } V', w')$ occurs in $extend(\psi, w'')$, we conclude that $w'' = w'$. Thus, $f \text{ in } V'$ occurs in ψ which is a fluent precondition in D . Then, since D is adequate, $f \text{ in } V \text{ when } \psi$ is an effect proposition in D and $f \in A_g$ then $V \subseteq V'$ or $V \cap V' = \emptyset$. \square

Lemma 3.13.2. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree and adequate, if $\langle I', D', G' \rangle$ is obtained by determinizing a multiple initial fluent d , then D' is adequate.*

Proof. Similar to proof of Lemma 3.13.1 \square

Conjecture 3.13.1. *For any conformant planning problem $\langle I, D, G \rangle$ where D is copyfree and uniformity preserving, if $\langle I', D', G' \rangle$ is obtained by determinizing a nondeterministic action effect g in W , then D' is uniformity preserving.*

Proof. This proof is left as a future exercise. \square

Lemma 3.13.3. *For any conformant problem $\langle I, D, G \rangle$ where D is copyfree and d a multiple initial fluent in I , if we obtain $\langle I', D', G' \rangle$ by determinizing d , then D' is copyfree.*

Proof. Let f be an original fluent of D . We need to show that $extend(f, w)$, where $w \in W$, does not have a copy in D' other than itself. Consider 2 cases:

1. $f \notin A_d$. Then $extend(f, w) = f$. Suppose there is a fluent f' in D' such that f' is a copy of f

and $f' \neq f$. We will derive a contradiction. Notice that f' cannot be a fluent of D (otherwise D would not be *copyfree*). Since f' is a fluent in D' , then there is a fluent f'' in D and $w' \in W$ such that $f' = \text{extend}(f'', w')$. Since f' is not in D , then $f'' \neq f'$. So, $f' = f'' \cdot w'$ and so we conclude that $f'' \in A_d$. Since $f'' \in A_d$ and $f \notin A_d$, then $f'' \neq f$. Since $f'' \cdot w'$ is a copy of f and $f'' \neq f$ then f'' is a copy of f . Then, since f'' and f are in D , we have our contradiction.

2. $f \in A_d$. Then $\text{extend}(f, w) = f \cdot w$. Suppose there is a fluent f' in D' such that f' is a copy of $f \cdot w$ and $f' \neq f \cdot w$. We will derive a contradiction. Since f' is a copy of $f \cdot w$ and $f' \neq f \cdot w$, then by definition of copy, f' is a copy of f . Since f' is in D' , $f' = \text{extend}(f'', w')$ where f'' in D and $w' \in \text{domain}(d)$. Notice that $f' \neq f$, since f' is a copy of $f \cdot w$. It follows that f' is not in D , since f' is a copy of f and $f' \neq f$ and D is *copyfree*. Therefore, $f' = f'' \cdot w'$ and $f'' \in A_d$ and $w' \in \text{domain}(d)$. So we know that $f'' \cdot w'$ is a copy of $f \cdot w$ and $f'' \cdot w' \neq f \cdot w$. Either $f'' \neq f$ or $w' \neq w$. Next we need to show that $f'' \neq f$. So assume that $w' \neq w$. (If not then we have $f'' \neq f$ already.) Since $f'' \cdot w'$ is a copy of $f \cdot w$ then f'' is a copy of $f \cdot w$, from which it follows that $f'' \neq f$. And since $f'' \cdot w'$ is a copy of $f \cdot w$, we know that f'' is a copy of f . Furthermore, $f'' \neq f$ and $f'' \in A_d$ and $f \in A_d$ contradicting the assumption that D is *copyfree*. \square

Lemma 3.13.4. *For any conformant problem $\langle I, D, G \rangle$ where D is copyfree and g in W is a nondeterministic fluent proposition in D , if we obtain $\langle I', D', G' \rangle$ by determinizing g in W , then D' is copyfree.*

Proof. The proof is similar to the proof of Lemma 3.13.3. \square

3.14 Proof of Termination

Definition 3.14.1. *Assume that I mentions all fluents. Then, for each fluent f ,*

$$W_f \text{ is the unique subset of } \text{domain}(f) \text{ such that } f \text{ in } W_f \in I.$$

Lemma 3.14.1. *For a conformant problem $\langle I, D, G \rangle$ where D is copyfree, the procedure of determinizing all multiple initial fluents in I , one at a time, terminates.*

Proof. Let $\langle f_1, \dots, f_n \rangle$ be an arbitrary ordering of the multiple initial fluents in I . Since there is exactly one fluent proposition f in V in I for every fluent f in $\langle I, D, G \rangle$ and the set of fluents in $\langle I, D, G \rangle$ is finite, then we conclude that I is finite. Take any PSD P , in any extension of the language of D , M_P will be the

sequence of length n where the i^{th} element is the set consisting of all copies of f_i that are multiple in P . So for instance

$$M_I = \langle \{f_1\}, \dots, \{f_n\} \rangle.$$

Moreover, if $\langle I', D', G' \rangle$ is the result of determinizing f_1 then

$$M_{I'} = \langle \emptyset, extend(f_2, W_{f_1}), \dots, extend(f_n, W_{f_1}) \rangle.$$

This follows immediately from definition of I' in term of I . Next we determinize an element of $extend(f_2, W_{f_1}), extend(f_2, w)$, to obtain $\langle I'', D'', G'' \rangle$. There are two cases to consider:

Case 1: $f_2 \notin A_{f_1}$. Then $extend(f_2, W_{f_1}) = \{f_2\}$. By same argument as above,

$$M_{I''} = \langle \emptyset, \emptyset, extend(extend(f_3, W_{f_1}), W_{f_2}), \dots, extend(extend(f_n, W_{f_1}), W_{f_2}) \rangle.$$

Case 2: $f_2 \in A_{f_1}$. Since all elements of $extend(f_2, W_{f_1})$ are copies of f_2 , then by Lemma 3.12.3, for any $extend(f_2, w') \in extend(f_2, W_{f_1})$ other than $extend(f_2, w)$,

$$extend(f_2, w') \notin A_{extend(f_2, w)}.$$

Then,

$$extend(extend(f_2, W_{f_1}), W_{extend(f_2, w)}) = extend(f_2, W_{f_1}) - \{extend(f_2, w)\}.$$

Consequently,

$$M_{I'''} = \langle \emptyset, extend(extend(f_2, W_{f_1}), W_{extend(f_2, w)}), \dots, extend(extend(f_n, W_{f_1}), W_{extend(f_2, w)}) \rangle.$$

Although $|M_{I'}| = |M_{I''}|$, we have that the second element of $M_{I''}$ is smaller than the second element of $M_{I'}$ by one. Next we determinize another element of $extend(f_2, W_{f_1})$ to obtain $\langle I''', D''', G''' \rangle$ and using the same argument as above, the size of the second element of $M_{I''''}$ is smaller than the second element of $M_{I''}$ by one. Since the number of copies of each original fluent is finite then at some point in the determinizing we will have that the second element of M_{I^n} is the empty set, in which case we move to determinize one element at a time from the family of copies from the third element. Since $|M_{I^m}|$ is finite then after some finite number of determinizing steps, we will have a problem description $\langle I^p, D^p, G^p \rangle$ with $M_{I^p} = \langle \emptyset, \dots, \emptyset \rangle$ and so there are no more multiple initial fluents. Thus in this case the determinizing procedure with respect to multiple initial fluents terminates. \square

3.14. PROOF OF TERMINATION

Lemma 3.14.2. *For a conformant problem $\langle I, D, G \rangle$ where D is copyfree, the procedure of determinizing all nondeterministic fluent propositions in D , one at a time, terminates.*

Due to time constraints, a proof for Lemma 3.14.2 is not given in this thesis. However, formulating such a proof is left for future work and is mentioned in Section 7.1.

Chapter 4

Implementation

4.1 Implementation

In this section, we describe the implementation of the determinizing procedure into a computer program. This allows for determinizing larger planning problems more easily and efficiently. The implementation was done in the following environment:

- C++ programming language with the STL library
- gcc version 3.2.2 compiler collection
- Sun Solaris 9 Operating environment

The application framework is decomposed into three components as follows (for a pictorial overview see Figure 4.1):

1. Input
2. Determinizer
3. Output

The input layer is responsible for parsing a planning problem in the language presented in Appendix A. This language is based on the language presented in Section 2.1. It is important to note here that in the current implementation, the input parser does not support fluent formulas in the goal (recall that the planning framework introduced in Section 2.1 allows for that). Once the input has been parsed and given there are no syntax errors, the problem is encoded in an internal representation and passed to the “Determinizer”.

The Determinizer runs the determinizing algorithm and transforms the problem into a classical problem. Currently, there is no guarantee that the resulting problem will be correct (i.e., equivalent to the input conformant problem). This is because the Determinizer has no means to check for uniformity in the initial state and uniformity preserving in the domain of the input problem. It is the responsibility of the user to check for these two conditions using other means. This shortfall should be remedied in future releases of the software.

After the problem has been determinized, it is passed to the output layer which will transform it to an output language, as defined by the output layer. The end result is a planning problem equivalent to the

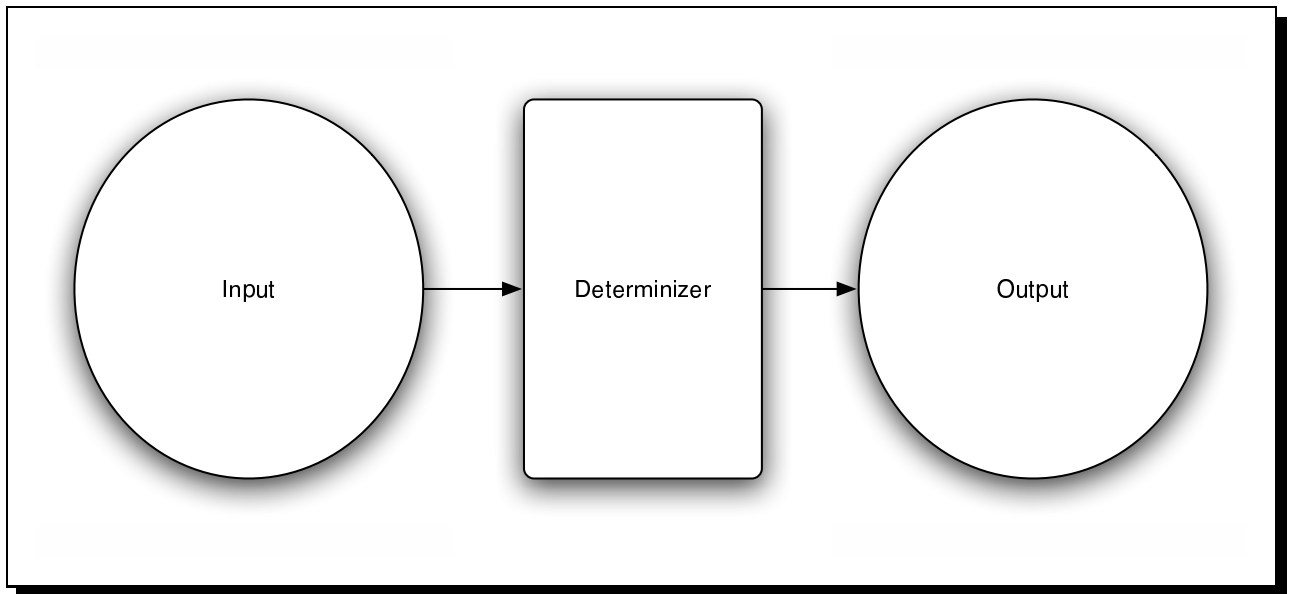


Figure 4.1: Overview of application framework.

input conformant problem, if possible, with no uncertainty in the language supported by the output layer. We have chosen a modularized approach for the output layer to allow for more flexibility of the output language. A user can supply their own output plugin, integrate it into the system and recompile for it to take effect. A more appealing approach here is to eliminate the need for recompilation and introduce dynamic plugins for the output layer. However, due to time limitations, this was not possible in the current implementation. Therefore, we have made the modifications needed to the main program be minimal to allow for easier plug-in development. A default compiled output plugin is provided for the PDDL language (see Subsection 4.1.2).

To make implementation easier we chose the path of Object Oriented programming paradigm. At the same time consideration is needed for speed and efficiency, as the determinizing process is CPU and memory intensive. For this reason, our decision favored the C++ programming language with the STL library as opposed to the Java programming language. Next, the different classes that make up the determinizer module will be described, in addition to the data structures used to internally represent the planning problem.

As mentioned in Section 2.1, one of the two basic building blocks of a problem description is the fluent. Therefore to model a fluent in our language we introduce the *Fluent* class (fluent.h). Since every fluent has a domain (recall that a domain is a nonempty finite set of symbols), we include in the *Fluent* class a set of

strings object (*set < string >*) to represent the domain. One can think of a Boolean fluent as a special type of fluent whose domain is a Boolean domain and therefore we have an is-a relationship between a Fluent and a Boolean Fluent. As a result we create a subclass *BoolFluent* that, as the name suggests, represents a Boolean fluent. Since the domain is fixed (by definition), there is no need to explicitly specify the domain, as it is initialized by default to the Boolean domain (i.e., $\{true, false\}$) by the class constructor.

Next, we have the *FluentProp* class (*fluentprop.h*) that represents a fluent proposition f in V . There are basically two parts to a fluent proposition; a fluent f and a set $V \subseteq domain(f)$. Therefore, in the *FluentProp* class, we include a *Fluent* object for f and a set of strings for V . For example, to initialize a *FluentProp* object for the fluent proposition f in $\{P1, P2\}$, one could do

```
string domain_in[2] = { 'P1', 'P2' };    // Initialize the domain
Fluent *f_in = new Fluent( 'in', domain_in, 2); // Create the fluent
set<string> V;    // Initialize V
V.insert( domain_in[0] );    // Set the elements of V
FluentProp fp( f_in, V );    // Create the fluent proposition
```

Here it should be noted that the destructor of the *FluentProp* class will perform the necessary cleanup to free any memory from heap used by the object. This frees the programmer from explicitly deleting such memory. However, care needs to be taken to not delete objects while the *FluentProp* object is still alive.

Class *PSD* (*psd.h*) describes a partial state description. It contains functions to traverse, as well as manipulate, the elements of the partial state description.

The next class is *EffectProp* (*effectprop.h*) that describes an effect proposition. This class provides access to the antecedant (here, a *PSD*) as well as to the consequent (a set of fluent propositions) of the effect proposition object through its public interface. As stated earlier, the current implementation supports partial state descriptions as the antecedant of effect propositions and not fluent formulas as allowed in the language defined in Section 2.1.

Class *Operator* (*operator.h*) describes an operator (as defined in section 2). The public interface of the class provides functions to access and modify the name, precondition and effect of the operator. As with the case of the *EffectProp* class, the current implementation allows only for the operator precondition to be a partial

state description and not a fluent formula as allowed in the language defined in Section 2.1.

An operator has three constructors as follows:

```
Operator(); // Empty constructor
Operator( string ); // Action constructor
Operator( const Operator & ); // Copy constructor
```

One should note here that the current implementation does not allow for the definition of schematic operators defined over a metavariable (as used to define operator *dunk* in Figure 1.3). Rather, the user needs to define operators explicitly.

For example, one could create a new operator as follows:

```
Operator dunkP1( ``dunk-P1`` )
dunkP1.addPrecon( inP1 ); // inP1 must be defined
dunkP1.addEffect( clogged ); // clogged must be defined
```

The above code creates the following new operator:

```
action constant: dunk-P1
precondition: in={P1}
effects: clogged
```

Class *Conformant_problem*, as the name suggests, represents a conformant planning problem. The implementation follows the same format as defined in Section 2.1 (recall that a planning problem is defined in terms of a triple $\langle I, D, G \rangle$). However, the current implementation allows for G to be a partial state description and not a fluent formula. Access to I, D, G is provided through the public interface of the class. Classes in need to deal with a problem should do so by creating instances of the *Conformant_problem* class.

The main class that implements the determinizing procedure is the *Determinizer* class (*determinizer.h*) whose algorithm is summarized as follows:

- Determinize all multiple fluents in the initial state,

- Determinize all nondeterministic effects in the domain D .

Notice that the algorithm has no means to check for uniformity of the initial state and uniformity preserving of the domain. Therefore, currently, this responsibility falls in the hands of the user. In this current implementation, the order of determinizing multiple fluents is purely random. This should be changed in future releases since the order affects the size of the determinized problem and thus the time required to solve the resulting classical planning problem.

4.1.1 Input parser

The input parser module is written in yacc and lex with the grammar as specified in Appendix A. This module is responsible for creating the appropriate data structures, as described above, initializing and running the Determinizer module, and finally creating the appropriate output plugin.

4.1.2 PDDL output plugin

The PDDL output plugin is responsible for transforming a planning problem to PDDL and commit the new description to permanent memory in the form of a file. The transformation between the two descriptions, in this case, was not straightforward. One major incompatibility between these two representations is in the definition of the building blocks of a planning problem. Recall that our representation defines the primitive element to be the fluent which is a constant that takes one value at a time from its domain. However, PDDL does not have this notion of a fluent. In contrast, it works on the level of Boolean predicates, where a predicate is an expression that can either be “on” or “off”. This representation translates well to Boolean fluents but not to nonBoolean fluents. To better understand the difficulty here, we will present an example.

Consider the operator *move_right* in D' expressed in our language as

```
operator move_right
{
  effects =
  [
```

```

in_room_b=[b] when [in_room_b a]
in_room_b=[a] when [in_room_b b]

in_room_a=[a] when [in_room_a b]
in_room_a=[b] when [in_room_a a]
];
}

```

A direct translation of this description to PDDL will yield:

```

(:action move_right
  :effect
  (and
    (when (in_room_b a) (in_room_b b))
    (when (in_room_a a) (in_room_a b))
    (when (in_room_b b) (in_room_b a))
    (when (in_room_a b) (in_room_a a))
  )
)

```

A closer look at the above PDDL description shows that, semantically, this is not what we want. This is because it is possible to have the predicates

```
(in_room_b a) (in_room_b b)
```

be “on” at the same time. What this means is that the robot can be in both rooms at the same time, which is not what we want (our original description allows the robot to be in only one room at a time). The reason that this occurs is because we are transforming a nonBoolean fluent, *in_room_b*, into a set of Boolean predicates, $\{(in_room_ba), (in_room_bb)\}$, where each Boolean predicate occurs independent of the other. To remedy this situation, we need to force the elements this set to behave in the same way as the single nonBoolean fluent they represent. So, generally speaking, every time we change the value of one of

the nonBoolean fluents, we will “delete” all the other values, thus preserving the idea that a nonBoolean fluent can have at most one value at a time. As can be seen, a consequence of this procedure is an increase in the size of the problem, which in turn makes it harder for the classical planner to solve. The effects of this will be further illustrated in Section 5.1 with the Ring of Rooms problem. Therefore, our transformation rules need to be modified to take account of this discrepancy. In this example, the operator should be of the following form (in PDDL):

```
(:action move_right
  :effect
  (and
    (when (in_room_b a) (and (in_room_b b) (not (in_room_b a)) ))
    (when (in_room_a a) (and (in_room_a b) (not (in_room_a a)) ))
    (when (in_room_b b) (and (in_room_b a) (not (in_room_b b)) ))
    (when (in_room_a b) (and (in_room_a a) (not (in_room_a b)) ))
  )
)
```

Chapter 5

Experimental Results

5.1 Experimental Results

In this section we will present our experimental results and compare some of them to the results of Chen [4] (were applicable) as well as to other conformant planning systems. The problems discussed are the **Bomb in the toilet** problem, **Ring of rooms** problem, **SQUARE** problem and **CUBE** problem.

We ran all the experiments using a SUN UltraSparc 60 workstation with the following specifications:

- Sun UltraSparc 60 - elite3D
 - UltraSparc II CPU at 450MHz with 4MB Cache x 2
 - 1GB Physical RAM
- Solaris OE 9
- gcc v3.2.2

As noted earlier, the determinizing procedure transforms a conformant problem to a classical problem (if possible) which in turn is fed to a classical planner. For the purpose of our experiments, we decided to use a classical planner from the AIPS competition. There were several options to choose from: Blackbox, FF¹, IPP, SGP, and SHOP. (An overview of these planners is found in Section 6.1.) All these classical planners, except for FF, created different issues and as a result we decided to use FF. Furthermore, FF ranked high in several planning competitions. Thus, it seemed to be the best candidate. For example, SHOP would not run the generated classical problems for reasons not clear.

In the sections to follow we discuss our experimental results of each problem individually.

5.1.1 Bomb in the Toilet

The bomb in the toilet problem is a classic benchmark for conformant planning systems and is used by most people in analysing and comparing performance results. As described earlier, this problem comes in many flavors but for our experiments we will be dealing with the following four:

¹As of this writing, the current version is 2.3 and it can be found at <http://www.informatik.uni-freiburg.de/hoffmann/ff.html>.

- $BTC(n)$ - uncertainty only in the initial state with one toilet and n packages
- $BMTC(n,t)$ - uncertainty only in the initial state with t toilets and n packages
- $BTUC(n)$ - uncertainty in the initial state and the domain with one toilet and n packages
- $BMTUC(n,t)$ - uncertainty in the initial state and the domain with t toilets and n packages

We will follow the generally accepted notation of adding an ‘M’ after the ‘B’ when we have multiple toilets (for example, $BMTC(t,2)$ is $BTC(2)$ with t toilets) and adding a ‘U’ before the ‘C’ when there are nondeterministic effects in the domain as in $BTUC(2)$ which is $BTC(2)$ with nondeterministic effects.

In particular, we experimented with the following problem instances:

- $BTC(2) \dots BTC(75)$ (Table 5.2)
- $BTUC(2) \dots BTUC(75)$ (Table 5.4)
- $BMTC(t,n)$ for $t = 2, 4, 6$ and $n = 2 \dots 15$ (Table 5.5, Table 5.6, Table 5.7)
- $BMTUC(t,n)$ for $t = 2, 4, 6$ and $n = 2 \dots 15$ (Table 5.8, Table 5.9, Table 5.10)

As it turns out, this problem seems to be too easy for this procedure and FF to solve. In the case that we have a single toilet, we go up to 75 packages and our time still beats the results reported by Chen. However, when we have a few packages, e.g. 2,3,4, our results are slightly inferior to the other people’s results. This does not weigh much importance since the timings are in the milliseconds and thus the difference can be attributed to other factors such as initialization, timing clock inaccuracy, etc. Also, it can be observed from the plots that our curves have a smooth exponential nature, as should be expected.

It should be noted that the timings are reported in seconds and are the average over 20 trials for each experiment.

5.1.2 Ring of Rooms

The ring of rooms problem turned out to be more complex than anticipated. Due to difficulties in translating from our language to PDDL we were not able to run experiments for this problem instance. In fact the difficulties are so severe that we were unable to successfully run experiments for the RING(2) problem.

The main difficulty encountered is in the translation of nonBoolean fluents in our language description to predicates ² in PDDL. Stated otherwise, the problem arises because during this translation, we are transforming a nonBoolean fluent to multiple Boolean fluents while maintaining the property that a nonBoolean fluent takes one value from its domain at a time. We will call this the *all-at-once* problem. To get a better understanding of the issue consider the action *move_left* from the determinized RING(2) problem as shown in Figure 5.1. To keep the example simple, let us consider the effect proposition

$$in_room_a = b \textbf{ when } in_room_a = a .$$

In translating this effect proposition to PDDL, one can write

$$(when(in_room_aa)(in_room_ab)).$$

At first this translation might seem correct, but in fact it is wrong (in other words these two effects are not equivalent). The culprit is that by “activating” the predicate (*in_room_ab*) we do not “deactivate” the predicate (*in_room_aa*) and so the robot is in two rooms at the same time! The issue here arises because of the transformation of a nonBoolean fluent to a Boolean fluent. The remedy to this situation is to explicitly “deactivate” the predicate (*in_room_aa*), with the resulting effect written in PDDL as

$$(when(in_room_aa)(and(in_room_ab)(not(in_room_aa)))).$$

As can be noticed, this increases the size of the resulting problem in PDDL, especially when we do this transformation to every nonBoolean fluent.

As a direct consequence of this problem, FF is not able to solve any instance of the ring of rooms problem given the current hardware. The reason for this is that FF transforms all action preconditions to DNF in its preprocessing phase (which is worst-case exponential). However, when we add the predicate negations, we are effectively making the action preconditions more complex and as a result FF is not able to simplify them before the transformation thus causing FF to fail. (Here I would like to thank the author of FF for his quick response and insightful help to our questions regarding this issue.) Note here that this failure is due to how FF solves problems and does not necessarily apply to other planners.

Even with more resources, it will be highly inefficient and thus not feasible.

²Predicates are the basic building blocks in PDDL. They can be thought of as being equivalent to Boolean fluents as defined in our language

Operator: *move_left*

Precondition:

Effects: $\{in_room_a = b \textbf{ when } in_room_a = a, in_room_a = a \textbf{ when } in_room_a = b,$
 $in_room_b = b \textbf{ when } in_room_b = a, in_room_b = a \textbf{ when } in_room_b = b\}$

Figure 5.1: Description of the *move-left* action from the RING(2) problem

5.1.3 SQUARE

In dealing with the SQUARE problem we only considered the CORNER instance. This enables us to directly compare our results with those given in [2]. Table 5.11 shows the results of our experiments (labeled FF) in addition to the results reported in [2]. As shown in Table 5.11, our results beat the results of GPT(0) by a wide range but are close (although do not beat) the results of GPT(h). (Chen does not provide results for this problem and thus a comparison cannot be made.)

5.1.4 CUBE

In the CUBE problem we experimented with both the CENTER and CORNER instances. Here we were able to compare our results with those reported by [2] (for GPT(h) and GPT(0)) and [3] (for CFF and MBP). In CENTER we beat both CFF and MDB by a big margin. It seems that this problem is too “easy” for our determinizing procedure. Notice that in the CENTER instance, results are only given for $n = 3$ (CFF and MBP) and $n = 5$ (MBP) because for other values of n the problem could not be solved due to time-outs [3]. Also [2] did not report any results for this problem instance.

In the CORNER problem instance we see that our approach beats all four other conformant planners by an order of magnitude for all n . It is interesting here to note that CFF is the conformant version of FF and how by using our approach we beat CFF by such a margin.

5.1.5 Result tables

In this section we present the results of our experiments together with other people’s results in easy to read tabular format. Note here that all results are in seconds and dashes indicate missing results.

# packages	FF	$CCalc_{New}$	# packages	FF	$CCalc_{New}$
2	0.0180	0.0040	15	0.0540	-
3	0.0185	0.0070	16	0.0610	0.8850
4	0.0215	0.0110	17	0.0660	-
5	0.0225	0.0180	18	0.0715	-
6	0.0240	0.0560	19	0.0805	-
7	0.0270	0.0800	20	0.0840	-
8	0.0295	0.0970	21	0.0925	2.9500
9	0.0345	0.1340	22	0.0985	-
10	0.0330	0.1670	23	0.1025	-
11	0.0390	-	24	0.1165	-
12	0.0435	-	25	0.1225	-
13	0.0440	-	26	0.1350	7.4900
14	0.0540	-	27	0.142	-

Table 5.1: Comparing our results for BTC(n) using the FF planner with the results of Chen [4] using $CCalc_{New}$ planner. From the table, it is clear that our results are superior to the results of Chen. The - in an entry field means that the number was not given, either because the experiment was not performed or could not be performed. It should be noted here that the results are in seconds.

5.1. EXPERIMENTAL RESULTS

# packages	Time (seconds)	# packages	Time (seconds)
28	0.1575	52	0.5905
29	0.1675	53	0.617
30	0.1755	54	0.6535
31	0.188	55	0.6905
32	0.2	56	0.713
33	0.2155	57	0.7455
34	0.2375	58	0.778
35	0.2415	59	0.812
36	0.26	60	0.8565
37	0.277	61	0.8815
38	0.295	62	0.9205
39	0.306	63	0.9705
40	0.33	64	0.9945
41	0.341	65	1.0305
42	0.361	66	1.078
43	0.3815	67	1.1135
44	0.402	68	1.1685
45	0.422	69	1.193
46	0.4435	70	1.256
47	0.4675	71	1.2935
48	0.4925	72	1.3545
49	0.514	73	1.3885
50	0.5435	74	1.4445
51	0.5685	75	1.499

Table 5.2: Chen reported results for BTC(2) to BTC(26) (of which the comparison is reported in Table 5.1). Since our results were very promising, we wanted to experiment further with the BTC(n) problem and so we went as far as BTC(75). As shown here, our results are far superior than those reported by Chen. For example, consider that BTC(75) took 1.499 seconds while, for Chen, BTC(21) took 2.95 seconds.

# packages	FF	$CCalc_{New}$	# packages	FF	$CCalc_{New}$
2	0.0185	0.003	16	0.0595	1.000
3	0.0175	0.007	17	0.0670	1.220
4	0.0205	0.009	18	0.0735	-
5	0.0220	0.016	19	0.0795	-
6	0.0245	0.024	20	0.0855	2.490
7	0.0260	0.038	21	0.0900	3.050
8	0.0300	0.061	22	0.1025	-
9	0.0335	0.089	23	0.1120	-
10	0.0345	0.125	24	0.1235	5.620
11	0.0375	-	25	0.1310	-
12	0.0440	-	26	0.1380	7.650
13	0.0460	-	27	0.1480	-
14	0.0525	-	28	0.1620	17.150
15	0.0580	-	29	0.1690	-

Table 5.3: Comparing our results for BTUC(n) using the FF planner with the results of Chen [4] using $CCalc_{New}$ planner. It is evident that our results beat Chen's results for this problem, as was also the case for the BTC problem. The - in an entry field means that the number was not given, either because the experiment was not performed or could not be performed. It should be noted here that the results are in seconds.

5.1. EXPERIMENTAL RESULTS

# packages	FF	# packages	FF
30	0.186	54	0.680
31	0.198	55	0.716
32	0.205	56	0.740
33	0.219	57	0.784
34	0.235	58	0.812
35	0.254	59	0.852
36	0.266	60	0.880
37	0.284	61	0.921
38	0.303	62	0.965
39	0.318	63	1.013
40	0.338	64	1.046
41	0.358	65	1.071
42	0.376	66	1.122
43	0.400	67	1.157
44	0.413	68	1.208
45	0.440	69	1.250
46	0.457	70	1.290
47	0.481	71	1.375
48	0.505	72	1.398
49	0.536	73	1.457
50	0.559	74	1.500
51	0.600	75	1.551
52	0.619		
53	0.654		

Table 5.4: Chen reported results for BTUC(2) to BTUC(28) (of which the comparison is reported in Table 5.3). Since our results were very promising, we wanted to experiment further with the BTUC(n) problem and so we went as far as BTUC(75). As shown here, our results are far superior than those reported by Chen. For example, consider that BTUC(75) took 1.551 seconds while, for Chen, BTUC(28) took 17.150 seconds.

# toilets	# packages	FF	$CCalc_{New}$
2	2	0.018	0.000
2	3	0.018	0.008
2	4	0.024	0.006
2	5	0.024	0.016
2	6	0.029	0.008
2	7	0.033	0.024
2	8	0.035	0.310
2	9	0.040	0.060
2	10	0.044	26.660
2	11	0.051	-
2	12	0.056	-
2	13	0.064	-
2	14	0.068	-
2	15	0.076	-

Table 5.5: Here we compare our results (labeled FF) with the results of Chen (labeled $CCalc_{New}$) for the BMTC(2,n) problem. It is clear from the timings that we beat Chen's results in this problem too. The difference becomes evident on BMTC(2,10) where it takes FF 0.044 seconds to solve as compared with Chen's timing of 26.660 seconds.

# toilets	# packages	FF	$CCalc_{New}$
4	2	0.019	0.002
4	3	0.025	0.000
4	4	0.027	0.010
4	5	0.032	0.012
4	6	0.035	0.018
4	7	0.044	0.026
4	8	0.049	0.033
4	9	0.057	0.133
4	10	0.065	1.492
4	11	0.074	-
4	12	0.083	-
4	13	0.095	-
4	14	0.104	-
4	15	0.121	-

Table 5.6: Here we compare our results (labeled FF) with the results of Chen (labeled $CCalc_{New}$) for the BMTC(4,n) problem. It is clear from the timings that we beat Chen's results in this problem too.

# toilets	# packages	FF	$CCalc_{New}$
6	2	0.024	0.006
6	3	0.027	0.004
6	4	0.034	0.010
6	5	0.040	0.010
6	6	0.044	0.012
6	7	0.057	39.260
6	8	0.063	187.480
6	9	0.076	779.540
6	10	0.089	-
6	11	0.100	-
6	12	0.112	-
6	13	0.132	-
6	14	0.148	-
6	15	0.162	-

Table 5.7: Here we compare our results (labeled FF) with the results of Chen (labeled $CCalc_{New}$) for the BMTC(6,n) problem.

# toilets	# packages	FF	$CCalc_{New}$
2	2	0.017	0.003
2	3	0.021	0.003
2	4	0.025	0.003
2	5	0.027	0.018
2	6	0.031	0.019
2	7	0.037	0.028
2	8	0.039	0.033
2	9	0.046	0.061
2	10	0.047	0.077
2	11	0.055	-
2	12	0.062	-
2	13	0.069	-
2	14	0.075	-
2	15	0.085	-

Table 5.8: Here we compare our results (labeled FF) with the results of Chen (labeled $CCalc_{New}$) for the BMTUC(2,n) problem.

# toilets	# packages	FF	$CCalc_{New}$
4	2	0.023	0.002
4	3	0.027	0.002
4	4	0.033	0.005
4	5	0.035	0.015
4	6	0.043	0.020
4	7	0.049	0.025
4	8	0.056	0.040
4	9	0.062	4.999
4	10	0.074	0.491
4	11	0.085	-
4	12	0.095	-
4	13	0.109	-
4	14	0.119	-
4	15	0.139	-

Table 5.9: Here we compare our results (labeled FF) with the results of Chen (labeled $CCalc_{New}$) for the BMTUC(4,n) problem.

5.1. EXPERIMENTAL RESULTS

# toilets	# packages	FF	$CCalc_{New}$
6	2	0.025	0.006
6	3	0.032	0.006
6	4	0.038	0.007
6	5	0.046	0.006
6	6	0.055	0.008
6	7	0.059	38.640
6	8	0.069	193.700
6	9	0.086	834.090
6	10	0.103	1312.760
6	11	0.116	-
6	12	0.126	-
6	13	0.146	-
6	14	0.166	-
6	15	0.185	-

Table 5.10: Here we compare our results (labeled FF) with the results of Chen (labeled $CCalc_{New}$) for the BMTUC(6,n) problem.

n	FF	GPT(h)	GPT(0)	CFF	MBP
5	0.05	-	-	-	-
12	0.35	0.118	2.995	-	-
14	0.54	0.159	7.103	-	-
16	0.84	0.219	14.909	-	-
18	1.17	0.290	29.580	-	-
20	1.58	0.386	53.851	-	-

Table 5.11: Results for SQUARE(n)

n	FF	GPT(h)	GPT(0)	CFF	MBP
3	0.04	-	-	0.19	2.82
5	0.09	-	-	-	2.74
7	0.29	-	-	-	-
9	1.01	-	-	-	-
11	3.13	-	-	-	-

Table 5.12: Results for CUBE(n) center

n	FF	GPT(h)	GPT(0)	CFF	MBP
3	0.03	-	-	0.00	0.04
5	0.06	-	-	0.06	2.94
6	0.09	0.165	6.022	-	-
7	0.13	0.266	20.347	0.48	-
8	0.18	0.450	66.539	-	-
9	0.24	0.654	-	1.80	-
10	0.32	0.991	-	-	-
11	0.42	-	-	5.66	-

Table 5.13: Results for CUBE(n) corner

Chapter 6

Related Work

6.1 Related Work

One of the earliest conformant planning systems based on classical planning techniques is Conformant Graphplan (CGP) [6]. CGP is based on Graphplan, a very efficient classical planner [1]. CGP works in two phases. First it expands the problem by creating separate “plan graphs”¹ for each of the possible initial states. Then it tries to find a plan that will solve the problem for all initial states. The authors found that the excellent performance of Graphplan on classical planning problems carried over to conformant problems by outperforming previous conformant planners.

Conformant Model Based Planner [5] (CMBP) solves conformant planning problems using Symbolic Model Checking Techniques. CMBP is depended on the representation of planning domains as a finite state automaton. To efficiently represent and search the automaton, the authors use Binary Decision Diagrams (BDD). In particular, the algorithm uses a breadth-first backward search approach. If a solution for the problem exists, then the algorithm returns a plan of minimal length, otherwise no plan is returned [5]. The authors found that CMBP outperforms other conformant problems such as QBFPlan, CGP and GPT, in addition to being more expressive.

In [10], the authors introduce the QBFPlan conformant planner that solves conformant planning problems using Quantified Boolean formulas. The approach used is, basically, a generalization of SAT based planning to conformant problems.

GPT is a planner based on a heuristic search of the belief space² of the planning problem. GPT was introduced in [2] where the authors developed an A^* heuristic to help in the search. Two approaches are used; GPT(0) and GPT(h). GPT(0) uses the simple breadth-first search whereas GPT(h) uses a more complex heuristic derived by general transformation from the problem [2].

Conformant-FF (CFF) is a conformant planner based on the ideas of the Fast-Forward (FF) classical planner, developed by [3]. The basic idea of CFF is to search the state space, guided by a heuristic function that estimates the distance to the goal using a *relaxed plan* [3]³. Each state is ranked by the heuristic which

¹A plan graph is a structure for constructing “approximate” solutions to a classical planning problem (in polynomial time). Graphplan uses such approximate solutions to guide the search for a true solution to the planning problem.

²The belief space is the set of all possible belief states where, informally, a belief state is a set of states

³A relaxed plan is a plan that achieves the goal with the assumption that there are no negative effects i.e., all delete-lists are empty.

guides the search.

Chapter 7

Future Work

7.1 Future Work

Research in the field of conformant planning is still in the maturing phase and thus there are many unanswered questions and unsolved mysteries. This is also the case for the topic of this thesis. In this section, we list some of the future work. As mentioned previously, the order of determinizing multiple fluents (both in the initial state and the action effects) impacts the size and complexity of the resulting determinized problem. This in turn affects the performance of solving the determinized problem (which is one of the core reasons for this research). This is an area of great importance to the determinizing process and thus an excellent candidate for future work.

From a theoretical point of view, it would be beneficial if a proof that the determinizing procedure will eventually halt is presented. In Section 3.1 we present such a proof (for initial case only) given the assumption that fluents are determinized in a particular order (i.e., determinize all copies of a fluent before starting to determinize the next noncopy). Obviously this limits the practical applicability of the determinizing process and thus a more general proof of termination would be beneficial. That is, prove that the determinizing procedure terminates regardless of the order of fluents that are determinized. In addition, a proof of termination when determinizing nondeterministic fluents should be presented.

Conjecture 3.8.1 presents interesting results and its proof is left for future work.

As far as the method implementation is concerned, it is beneficial to prove that the program implementing the determinizing procedure terminates and modifying the program to incorporate fluent formulas both in the goal state and action effects. In addition it would be convenient for the user if the program could check if the given conformant problem can be determinized with regards to nondeterministic action effects (i.e., check for uniformity of I' and uniformity preserving of D).

As discussed in Section 3.7, adequacy is a means of checking if a domain, D , preserves uniformity. However, there is still a lack of understanding of the properties of adequacy and of the concept in general. For example, one important issue is to identify any limitations of adequacy and how these affect the practical use of the method. This leads to the next question. What are the consequences of these restrictions and can they be relaxed? These are important issues to investigate and thus are a good starting point for further research.

The “problem” of converting nonBoolean fluents to Boolean, as is the case with the Ring of Rooms problem,

7.1. FUTURE WORK

deserves particular attention. This is useful when expressing conformant problems in the PDDL language. As a result our procedure will be applicable to a wider spectrum of conformant problems.

And last but not least, an investigation of the use of “meta-variables” would be very beneficial.

By any means, this is not an exhaustive list of further research of the determinizing procedure. Nonetheless, these points are interesting and challenging, and enhance the understanding of the determinizing procedure both from a theoretical as well as practical standpoint.

Appendix A

Appendix A

<problem> ::= <fluent_def> <initial> <domain> <goal>

<fluent_def> ::= BeginFluent <fluent_list> EndFluent

<initial> ::= initialState <name> '=' '[' <psd> ']' ';' ;'

<domain> ::= <domain> <operator> ';' ;'
 <operator> ';' ;'

<goal> ::= goalState <name> '=' '[' <psd> ']' ';' ;'

<operator> ::= operator <name> '-' <name>
 operator <name>
 operator <name> '(' var <name> ':' <name> ')'

<precon_effect_prop_list> ::= <precon_effect_prop_list> <effect_prop>
 <effect_prop>

<effect_prop_list> ::= <effect_prop_list> <effect_prop>

<effect_prop>

<effect_prop> ::= <fluent_prop> when '[' <psd_ep> ']'
 <fluent_prop>

<psd> ::= <psd> <fluent_prop>
 <fluent_prop>

<psd_ep> ::= <psd_ep> <fluent_prop>
 <fluent_prop>

<fluent_prop> ::= <name> '=' '[' <value_list> ']'
 '!' <name>
 <name>

<value_list> ::= <value_list> <name>
 <name>

<fluent_list> ::= <fluent_list> <fluent>
 <fluent>

<fluent> ::= fluent <name> '(' <fluent_domain_list> ')'
 boolFluent <name>

<fluent_domain_list> ::= <fluent_domain_list> ',' <name>
 <name>

Appendix B

Appendix B - Blocks world PDDL description (from AIPS2000)

```
;;;;;;;;;;;;;  
;;; 4 Op-blocks world  
;;;;;;;;;;;;;
```

```
(define (domain BLOCKS)  
  (:requirements :strips)  
  (:predicates (on ?x ?y)  
               (ontable ?x)  
               (clear ?x)  
               (handempty)  
               (holding ?x)  
               )  
  
  (:action pick-up  
    :parameters (?x)  
    :precondition (and (clear ?x) (ontable ?x) (handempty)))
```

```
    :effect
      (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)))

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

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

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

```
(not (handempty))
(not (on ?x ?y))))))

(define (problem BLOCKS-4-0)
  (:domain BLOCKS)
  (:objects D B A C )
  (:INIT (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D) (ONTABLE C) (ONTABLE A)
  (ONTABLE B) (ONTABLE D) (HANDEEMPTY))
  (:goal (AND (ON D C) (ON C B) (ON B A)))
)
```

Appendix C

Appendix C - BTUC(2)

```
# BTUC(2) problem
#
#

# Declaration of fluents
BeginFluent
fluent in (P1, P2);
boolFluent clogged;
boolFluent armed;
boolFluent damp_P1;
boolFluent damp_P2;
EndFluent

# Description of the initial state.
initialState I = [ in = [P1 P2] !clogged !damp_P1 !damp_P2 armed ];

# Description of the domain
operator dunk-P1
```

```
{
precondition = [!clogged !damp_P1];
effects = [
    !armed when [ in = [P1] ]
    clogged = [true]
    damp_P1
];
}
```

```
operator dunk-P2
{
precondition = [!clogged !damp_P2];
effects = [
    !armed when [ in = [P2] ]
    clogged = [true]
    damp_P2
];
}
```

```
operator flush
{
effects = [ !clogged ];
}
```

```
# Description of the goal state
goalState G = [ !armed ];
```

Appendix D

Appendix D - RING(2)

```
# RING(2) problem
#
#

# Declaration of fluents
BeginFluent
fluent win_a (open, closed, locked);
fluent win_b (open, closed, locked);
fluent in_room (a, b);
EndFluent

# Description of the initial state.
initialState I = [ win_a = [open closed locked]
                  win_b = [open closed locked]
                  in_room = [a b]  ];

# Description of the domain
operator close
```

```
{
effects = [
    win_a = [closed] when [ in_room =[a] win_a =[open] ]
    win_b = [closed] when [ in_room =[b] win_b =[open] ]
];
}
```

```
operator lock
```

```
{
precondition = [
win_a=[closed locked] when [ in_room=[a] ]
win_b=[closed locked] when [ in_room=[b] ]
];
effects = [
    win_a = [locked] when [ in_room=[a] ]
    win_b = [locked] when [ in_room=[b] ]
];
}
```

```
operator move_right
```

```
{
effects = [
    in_room=[a] when [ in_room=[b] ]
    in_room=[b] when [ in_room=[a] ]
];
}
```

```
operator move_left
```

```
{
effects = [
```

```
    in_room=[b] when [ in_room=[a] ]
    in_room=[a] when [ in_room=[b] ]
];
}

# Description of the goal state
goalState G = [ win_a=[locked] win_b=[locked] ];
```

Appendix E

Appendix E - SQUARE(5)

```
BeginFluent
fluent x (S1, S2, S3, S4, S5);
fluent y (S1, S2, S3, S4, S5);
EndFluent
```

```
initialState I = [ ];
```

```
operator MoveLeft
{
effects =
[
x=[S1] when [x=[S1]]
x=[S1] when [x=[S2]]
x=[S2] when [x=[S3]]
x=[S3] when [x=[S4]]
x=[S4] when [x=[S5]]
];
}
```

```
operator MoveRight
{
effects =
[
x=[S2] when [x=[S1]]
x=[S3] when [x=[S2]]
x=[S4] when [x=[S3]]
x=[S5] when [x=[S4]]
x=[S5] when [x=[S5]]
];
}
```

```
operator MoveUp
{
effects =
[
y=[S2] when [y=[S1]]
y=[S3] when [y=[S2]]
y=[S4] when [y=[S3]]
y=[S5] when [y=[S4]]
y=[S5] when [y=[S5]]
];
}
```

```
operator MoveDown
{
effects =
[
y=[S1] when [y=[S1]]
y=[S1] when [y=[S2]]
];
}
```

```
y=[S2] when [y=[S3]]
```

```
y=[S3] when [y=[S4]]
```

```
y=[S4] when [y=[S5]]
```

```
];
```

```
}
```

```
goalState G = [ x=[S1] y=[S1] ];
```

Appendix F

Appendix F - CUBE(3) [corner]

```
BeginFluent
fluent x (S1, S2, S3);
fluent y (S1, S2, S3);
fluent z (S1, S2, S3);
EndFluent
```

```
initialState I = [ ];
```

```
operator MoveLeft
{
effects =
[
x=[S1] when [x=[S1]]
x=[S1] when [x=[S2]]
x=[S2] when [x=[S3]]
];
}
```

```
operator MoveRight
```

```
{
effects =
[
x=[S2] when [x=[S1]]
x=[S3] when [x=[S2]]
x=[S3] when [x=[S3]]
];
}
```

```
operator MoveUp
```

```
{
effects =
[
y=[S2] when [y=[S1]]
y=[S3] when [y=[S2]]
y=[S3] when [y=[S3]]
];
}
```

```
operator MoveDown
```

```
{
effects =
[
y=[S1] when [y=[S1]]
y=[S1] when [y=[S2]]
y=[S2] when [y=[S3]]
];
}
```

```
operator MoveIn
```

```
{
effects =
[
z=[S2] when [z=[S1]]
z=[S3] when [z=[S2]]
z=[S3] when [z=[S3]]
];
}
```

```
operator MoveOut
{
effects =
[
z=[S1] when [z=[S1]]
z=[S1] when [z=[S2]]
z=[S2] when [z=[S3]]
];
}
```

```
goalState G = [ x=[S1] y=[S1] z=[S1] ];
```

Appendix G

Appendix G - SAFE(5)

```
# safe(5) problem
BeginFluent
boolFluent safe_open;
fluent right_combination (c1, c2, c3, c4, c5);
EndFluent

initialState I = [ !safe_open right_combination=[c1 c2 c3 c4 c5 ] ];

operator try_c1
{
    effects = [ safe_open when [right_combination=[c1]] ];
}

operator try_c2
{
    effects = [ safe_open when [right_combination=[c2]] ];
}

operator try_c3
```

```
{
  effects = [ safe_open when [right_combination=[c3]] ];
}

operator try_c4
{
  effects = [ safe_open when [right_combination=[c4]] ];
}

operator try_c5
{
  effects = [ safe_open when [right_combination=[c5]] ];
}

goalState G=[ safe_open ];
```

Bibliography

- [1] M. Furst A. Blum. Fast planning through planning graph analysis. In *Proc. Int. Joint Conf. AI*, pages 1636–1642, 1995.
- [2] Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *Artificial Intelligence Planning Systems*, pages 52–61, 2000.
- [3] R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search.
- [4] Zhuo Chen. *Determinizing in Conformant Planning*. Masters, University of Minnesota Duluth, 2002.
- [5] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *J. of Artificial Intelligence Research*, 13:305–338, 2000.
- [6] D. Weld D. Smith. Conformant graphplan. *Proc. 15th National Conference on AI*, 15(1), 1998.
- [7] Drew McDermott et. al. Pddl - the planning domain definition language. 1998.
- [8] R.E. Fikes, P. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.
- [9] Shirley Xiaochun Liu. *Deterministic Conformant Planning with the Causal Calculator*. Masters, University of Minnesota Duluth, 2001.
- [10] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
- [11] H. Turner. Polynomial-length planning spans the polynomial hierarchy. *Proc. of Eighth European Conf. on Logics in Artificial Intelligence (JELIA'02)*, pages 111–124, 2002.