UNIVERSITY OF MINNESOTA

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

Shirley Xiaochun Liu

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. C. Hudson Turner

Name of Faculty Adviser

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Deterministic Conformant Planning with the Causal Calculator

by

Shirley Xiaochun Liu

May 2001

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science
under the instruction of Dr. Hudson Turner

Department of Computer Science
University of Minnesota
Duluth, Minnesota 55812
U.S.A.

Abstract

A planning problem has three components: a description of an initial state, a description of an action domain (specifying how actions can affect the state of the world), and a description of a goal. A "valid" plan is one that (starting in any state consistent with the description of the initial state) is guaranteed (i) to be executable and (ii) to achieve the goal when executed. In classical planning, the initial state is completely described, the effects of actions are deterministic, and a plan is a finite sequence of (possibly concurrent) actions. This thesis investigates a more general form of planning called deterministic conformant planning, in which we no longer assume that the initial state is completely known. There are few implemented systems for this form of planning, which is computationally quite challenging.

The Causal Calculator (CCalc) is an automated system for reasoning about the effects of actions. CCalc takes as input an action domain description in an expressive high-level action language called $\mathcal{C}$, which it translates into classical propositional logic. Query answering is then performed using standard satisfiability solvers-fast-improving (widely-studied) programs used to find a model of an arbitrary propositional theory. CCalc can be used for classical planning: simply describe a deterministic action domain in $\mathcal{C}$, and provide in addition formulas describing an initial state and a goal. The resulting propositional theory is submitted to a satisfiability solver, and if a model is found, the actions that occur in it constitute a valid plan.

In this thesis, CCALC is extended to accommodate conformant deterministic planning. This approach is notable for its use of an expressive action language and its application of the satisfiability method to conformant planning. The thesis includes a preliminary experimental comparison with the best current conformant planning systems which suggests that the ability to conveniently represent concurrent actions in $\mathcal{C}$ contributes to the effectiveness of the approach.

Acknowledgments

I would like to thank my advisor, Dr. Hudson Turner, for all his help and guidance that he has given me over the past two years. Especially during the last few months, he patiently read through several drafts of everything I wrote and always gave me suggestions that led to significant improvements in clarity and rigor. I am grateful to him for all this.

I would like to express my gratitude to my thesis committee members, Dr. Tim Colburn and Dr. Ron Regal. Each one of them has been very generous with his time. I am grateful as well to the CS faculty for providing me support during these two years of my graduate study.

I would like to thank my family, without their support, it would not be possible for me to finish the thesis work.

Dedication

This thesis is dedicated to my dear husband, Gan Chen, for his always love and support!

Contents

1 Introduction **1**

1.1 Classical Planning 1

1.2 Deterministic Conformant Planning 2

1.3 Classical Planning as Satisfiability 4

1.4 Deterministic Conformant Planning as Satisfiability 8

1.5 Action Language $\mathcal{C}$ 10

1.6 The Planning System – CCalc 11

1.7 Main Issues of the Thesis 13

2 Background on Conformant Planning **15**

3 Deterministic Conformant Planning With $\mathcal{C}$ **20**

3.1 Action Language $\mathcal{C}$ 20

3.1.1 Syntax and Semantics of Propositional Logic 21

3.1.2 Syntax and Semantics of $\mathcal{C}$ 22

3.1.3 Examples of $\mathcal{C}$ 27

3.2 Classical Planning 35

3.2.1 Definitions in Classical Planning 35

3.2.2 Examples of Classical Planning 35
3.3 Deterministic Conformant Planning 38
3.3.1 Definitions in Conformant Planning 38

4 Implementing Deterministic Conformant Planning as Satisfiability 40

5 Experimental Results and Analysis 49

5.1 Results 49
5.2 Summary Remark 58

6 Conclusions and Future Work 59

Appendix A CC.h 62

List of Figures

1.1 A History View of Solution to “bomb in toilet” Problem 4

1.2 A World History of BT Domain Problem in Classical Planning . . . 7

3.1 A Simplified Transition System of BT Domain: 2 packages and 1 toilet 29

3.2 A Complete Transition System of BTC Domain Problems: 2 packages and 1 toilet 34

3.3 A World History of BTC Domain Problem in Classical Planning . . 37

3.4 A World History of BTC Domain Problem in Deterministic Conformant Planning 39

4.1 An Input File for the BMTC Domain: 4 packages; 2 toilets 46

4.2 A Planning File for the BMTC Domain 47

4.3 Sample Output for the BMTC Domain Problem: 4 packages and 2 toilets 48

5.1 A Performance Comparison of BTC Problem 52

5.2 A Performance Comparison of the BMTC Problems: 3 toilets(left), 6 toilets(right) 57

List of Tables

5.1 Results for BTC Problems 53
5.2 Results for BMTC Problems (2t) 54
5.3 Results for BMTC Problems (3t) 55
5.4 Results for BMTC Problems (4t) 55
5.5 Results for BMTC Problems (5t) 56
5.6 Results for BMTC Problems (6t) 56

Chapter 1

Introduction

How to plan in a domain where we don't have complete knowledge of initial states and effects of actions is a very challenging topic for many years. Research works have been done to extend classical planning to this field. Based on the research of representing action domain by causal theory and planning as satisfiability, this thesis presents a novel approach of conformant planning when uncertainty is present in the initial states.

1.1 Classical Planning

A planning problem consists of three components:

- a description of the initial state of the world,
- a description of a goal to be achieved, and
- a description of the actions that can be executed and how they can affect the state of the world.

In classical planning, we assume that the initial state is completely specified and that actions have deterministic effects. A plan is simply a finite sequence of actions. Intuitively, a plan is valid with respect to a classical planning problem if performing the actions in the plan takes the world from the initial state to a state in which the goal is satisfied.

1.2 Deterministic Conformant Planning

Conformant planning is one of the basic approaches to do planning when uncertainty presents in the world. In conformant planning, we no longer assume that the initial state is completely specified and actions are allowed to have nondeterministic effects, but a plan is still a sequence of actions. In order for a plan to be valid, it must be sure to work no matter which of the possible initial states actually obtains when it is executed, and no matter what is the outcome of nondeterministic actions during execution. In this thesis, we focus on a limited form of conformant planning, deterministic conformant planning, in which actions are assured to have deterministic effects but the initial state may be incompletely known.

A standard example is the "bomb in toilet" problem from [McD87] in which there are two packages: one of them contains a bomb, but we don't know which one. Dunking the package with the bomb in a toilet makes the bomb disarmed. After a dunk operation occurring to a toilet, the toilet is clogged by the package and needs to be flushed. To illustrate the problem, we formalize the initial conditions as $in(p1)$ and $in(p2)$ to indicate the two possible positions of the bomb. Intuitively, $in(p1)$ means the bomb is in the package p1 and $in(p2)$ means the bomb is in the package

p2. In the initial state, the bomb is armed, the bomb is in one of the two packages, the toilet is not clogged and the two packages are not dunked. The goal is $\neg armed$ (that is, not armed). Figure 1.1 shows a history view of a solution to "bomb in toilet" problem, from which we can see how the action sequence $dunk(p1), flush, dunk(p2)$ leads in all the possible worlds[1] from $armed$ to $\neg armed$. In world 1, the action $dunk(p1)$ disarms the bomb, causes the package to be dunked and also clogs the toilet. An action $flush$ is performed next and makes the toilet to be available for dunking again. The next action $dunk(p2)$ again causes the toilet to be clogged and the package 2 to be dunked. In world 2, the action $dunk(p1)$ causes the toilet to be clogged and package 1 to be dunked but the bomb is still armed. The next action $flush$ makes the toilet to be available for dunking again. The next action $dunk(p2)$ causes package 2 to be dunked, the bomb to be disarmed, and again the toilet to be clogged. So, the plan $dunk(p1), flush, dunk(p2)$ successfully disarms the bomb, no matter whether it is in package p1 or package p2. Another valid plan for this conformant planning problem is the action sequence $dunk(p2), flush, dunk(p1)$.

There are very few implemented systems for automated conformant planning. One reason is the computational complexity of conformant planning. According to [Tur01], conformant planning is Σ_3^P-complete for plans of polynomially-bounded length, and Σ_3^P-hard even for plans at length of 1.[2] Intuitively, as the amount of uncertainty grows, the number of possible worlds grows exponentially and the performance of conformant planning systems suffers accordingly.

[1] Possible world is not defined in the thesis. Here, we refer to its intuitive meaning.

[2] $P \leq NP \leq \Sigma_3^P \leq PSPACE$, where P means problem can be solved in polynomial time, NP means problem can be solved in nondeterministic polynomial time and PSPACE means problem can be solved in polynomial space [Pap94].

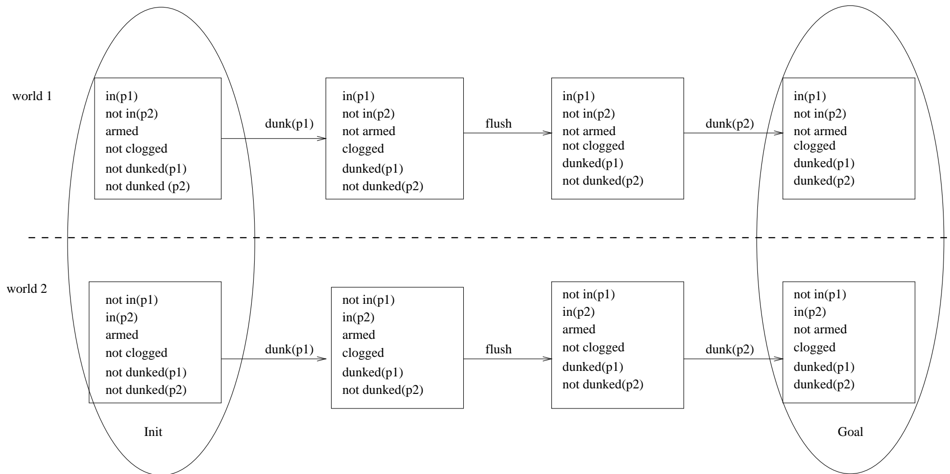

Figure 1.1: A History View of Solution to "bomb in toilet" Problem

1.3 Classical Planning as Satisfiability

Since a formal account of classical planning based on satisfiability of a set of formulas in classical propositional logic was proposed in [KS92], planning as satisfiability has become one of the most effective approaches to classical planning. Planning as satisfiability is, in brief, finding a model of a set of axioms of propositional logic that describes

- an initial state,
- a goal,
- the set of possible histories of some fixed length for the relevant action domain.

Finding a plan is simply finding the sequence of action occurrences in a model of the classical propositional theory, which can be obtained from any standard satisfiability solver for propositional logic.

In classical planning as satisfiability, we still keep the basic assumptions typifying classical planning. We assume that executing an action in any state always leads to a unique state and also that the initial state is completely specified. Under these assumptions, we will look at an example of classical planning as satisfiability.

Let us consider how to describe the histories of length 2 in a simplified "bomb in the toilet" action domain in which the only action is dunk, and the only properties of interest are which package the bomb is in and whether it is armed. We can write

$$in(p1,0) \equiv \neg in(p2,0) \tag{1.1}$$

to say that, at time 0, the bomb is in package p1 iff it is not in package 2. Next we would like to express that the bomb stays in the package it is in initially. For instance, we can write

$$in(p1,1) \equiv in(p1,0)$$

to say that the bomb is in package p1 at time 1 iff it is in package p1 at time 0. Since there are a number of such facts, it is convenient to use a "schematic formula" (or "schema"), which can stand for a family of propositional formulas.

$$in(P,T') \equiv in(P,T) \ \ (T \in \{0,1\},\ T' = T+1,\ P \in \{p1,p2\}) \tag{1.2}$$

Here P, T and T' are "meta-variables": we obtain in this case four distinct propositional formulas, one for each way of instantiating the meta-variables. (We'll find it convenient to use such schemas throughout the thesis.)

To say that the bomb is armed at time $t+1$ iff it was armed at time t and not dunked at time t, we can write

$$in(P,T) \supset (armed(T') \equiv (armed(T) \wedge \neg dunk(P,T))) \tag{1.3}$$

where $T \in \{0,1\}$, $T' = T+1$, $P \in \{p1, p1\}$. Roughly, the bomb remains armed iff it was armed before and the package that contains it hasn't been dunked.

Finally, we'll stipulate that at most one package can be dunked at a time, by writing

$$\neg(dunk(p1,T) \wedge dunk(p2,T)) \tag{1.4}$$

where $T \in \{0,1\}$.

Each model of (1.1)-(1.4) represents a possible history of length 2 in the action domain. For instance, in one such model, the following facts hold.

$$\begin{gathered} \neg in(p1,0) \;\; in(p2,0) \;\; armed(0) \\ dunk(p1,0) \;\; \neg dunk(p2,0) \\ \neg in(p1,1) \;\; in(p2,1) \;\; armed(1) \\ \neg dunk(p1,1) \;\; dunk(p2,1) \\ \neg in(p1,2) \;\; in(p2,2) \;\; \neg armed(2) \end{gathered}$$

So formulas (1.1)-(1.4) describe the histories of length 2 in this simplified "bomb in the toilet" domain. (By the way, this action domain is indeed deterministic, and therefore suitable for classical planning.) In order to finish specifying a classical planning problem, we need a formula that (completely) describes the initial state of the world, as well as formula describing the goal. For example, we may use

$$in(p2,0) \wedge armed(0) \tag{1.5}$$

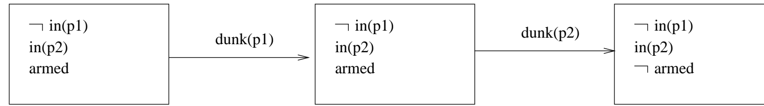

Figure 1.2: A World History of BT Domain Problem in Classical Planning

as our initial state description. In combination with formula (1.1), this formula completely determines the facts at time 0—the bomb is armed, it is in package p2, and by (1.1) it follows that the bomb is not in package p1. The goal can be described by writing

$$\neg armed(2). \tag{1.6}$$

Each model of (1.1)-(1.6) corresponds to a plan of length 2. Figure 1.2 gives a graphic representation of one such model; the associated plan is $dunk(p1),\ dunk(p2)$.

A primary advantage of the satisfiability planning method is that the main computation is done using general-purpose satisfiability solvers such as SATO [Zha97] and REL_SAT [BS97]. This is significant because satisfiability methods are applicable to a wide range of problems, and so satisfiability solvers have received a great deal of attention, and are rapidly improving.

In a recent survey article with current status of AI planning research, Weld [Wel99] gave a very nice summary on this issue:

> A common thread running through all of this [recent planning] research is the use of propositional representations, which support extremely fast inference.

Of course, another potential advantage of the satisfiability method for planning is that is uses an expressive language — classical propositional logic — to represent the planning problem. Hence it is possible to take advantage of recent advances in action representation methods, as we do in this thesis.

1.4 Deterministic Conformant Planning as Satisfiability

As far as we know, this thesis describes the first attempt to solve deterministic conformant planning directly by means of satisfiability. The idea of the extension is straightforward. Normally, a domain description describes the effects of actions in a (single) world. But in conformant planning, we are concerned with a number of worlds—all those with initial states consistent with what is known about the initial state. It is not difficult to modify the action description so that it instead describes the effects of actions in each of the possible worlds. The problem then becomes simply to find a sequence of actions (a plan) that accomplishes the goal in all of the possible worlds, and this can be done by a satisfiability planning system essentially as it is done for classical planning problems.

For example, let's consider how to adapt the propositional theory (1.1)-(1.4) from the previous section in order to represent "simultaneously" two histories in which the same actions occur but otherwise the facts may differ. In place of (1.1), we can write

$$in(p1, W, 0) \equiv \neg in(p2, W, 0) \tag{1.7}$$

where W is a meta-variable ranging over $w1, w2$, corresponding intuitively to the two histories or "possible worlds". Thus, (1.7) expresses that in each of the histories

the bomb is in exactly one of the two packages at time 0. Similarly, (1.2) becomes

$$in(P, W, T') \equiv in(P, W, T) \tag{1.8}$$

where $T \in \{0, 1\}$, $T' = T + 1$, $P \in \{p1, p1\}$, and $W \in \{w1, w2\}$.

Recall that we wish to have the same actions occur in each of the histories. Hence (1.3) becomes

$$in(P, W, T) \supset (armed(W, T') \equiv (armed(W, T) \wedge \neg dunk(P, T))) \tag{1.9}$$

where again $T \in \{0, 1\}$, $T' = T + 1$, $P \in \{p1, p1\}$, and $W \in \{w1, w2\}$. Notice in particular that no "world argument" is added to dunk, since, intuitively, the same action will occur in each of the possible worlds.

Finally, (1.4) need not be changed, since it mentions only actions.

Here are facts true in one of the models of (1.7)-(1.9), (1.4).

$$in(p1, w1, 0) \;\; \neg in(p2, w1, 0) \;\; armed(w1, 0)$$
$$\neg in(p1, w2, 0) \;\; in(p2, w2, 0) \;\; armed(w2, 0)$$
$$dunk(p1, 0) \;\; \neg dunk(p2, 0)$$
$$in(p1, w1, 1) \;\; \neg in(p2, w1, 1) \;\; \neg armed(w1, 1)$$
$$\neg in(p1, w2, 1) \;\; in(p2, w2, 1) \;\; armed(w2, 1)$$
$$\neg dunk(p1, 1) \;\; dunk(p2, 1)$$
$$in(p1, w1, 2) \;\; \neg in(p2, w1, 2) \;\; \neg armed(w1, 2)$$
$$\neg in(p1, w2, 2) \;\; in(p2, w2, 2) \;\; \neg armed(w2, 2)$$

In order to finish specifying a conformant planning problem in this fashion, we need a formula that completely describes each of the possible initial states, as well

as a formula saying that the goal holds in each of the possible final states. For instance, we may use

$$in(p1, w1) \wedge armed(w1) \wedge in(p2, w2) \wedge armed(w2) \tag{1.10}$$

to describe the two possible initial states. In combination with formula (1.7), this formula completely specifies the facts at time 0—in world w1, the bomb is in package p1, not in p2, and the bomb is armed; in world w2, the bomb is in package p2, not in p1, and the bomb is armed. For the goal, we can write

$$\neg armed(w1) \wedge \neg armed(w2). \tag{1.11}$$

Each model of (1.7)-(1.9), (1.4), (1.10)-(1.11) corresponds to a two-step plan solving the "bomb in the toilet" problem for the simplified "bomb in the toilet" domain.

A big issue in conformant planning is scaling. This thesis proposes a naive approach. It won't scale well with the increase of problem size. As the number of possible initial states grows, the size of the set of propositional formulas describing the set of histories increases linearly, and the cost of solving associated planning problems potentially grows exponentially. Nevertheless, according to our experimental results, this approach is very competitive with current conformant planners.

1.5 Action Language $\mathcal{C}$

Action languages are formal models of parts of the natural language that are used for describing the effects of actions [GL98a]. The action language $\mathcal{C}$ is one of such languages. It has the ability to conveniently express features like indirect effects of actions (ramifications), implied actions preconditions (qualifications), concurrent

actions and dynamic worlds in which things change by themselves. Moreover, a useful subset of $\mathcal{C}$ has a concise translation into classical propositional logic; thus $\mathcal{C}$ is useful for describing action domains in planning problems that are to be solved by the satisfiability method.

Here are examples of $\mathcal{C}$ propositions that express knowledge about the "bomb in the toilet" action domain. The proposition

$$\textbf{caused } \neg armed \textbf{ if } dunked(P1) \ \wedge \ in(P1)$$

expresses the fact that if package P1 contains the bomb and is dunked at some point in time, then there is a cause for the bomb to be disarmed at the same time. This is an example of a "static law" in $\mathcal{C}$—that is, a causal law that does not involve the passage of time. Another example of a $\mathcal{C}$ proposition is

$$dunk(P1) \textbf{ causes } dunked(P1)$$

which expresses the fact that dunking package P1 at some point in time causes it to be dunked at the next point in time. This is an example of a "dynamic law" in $\mathcal{C}$.

1.6 The Planning System – CCALC

The Causal Calculator, also known as CCALC[3], is an automated system for reasoning about actions. It is the system tool being used for investigating and testing the approach to solving deterministic conformant planning introduced in this thesis. It can take an action domain description in $\mathcal{C}$ and answer questions about it. It can also be used to solve classical planning problems, in which case a separate planning

[3]CCALC is available at http://www.cs.utexas.edu/users/tag/cc/.

file containing the descriptions of the initial state and the goal is also given as input.

Based on the description in [McC99], the procedures of CCalc in planning can be summarized as follows.

- First, a $\mathcal{C}$ domain description file is given as input. It includes schemas representing the $\mathcal{C}$ propositions that describe the action domain as well as additional declarations that specify the range of the metavariables used in the schemas. These $\mathcal{C}$ language schemas are translated into schemas of the internal language of CCalc, namely the language of causal theories [MT97].

- Second, the Causal Calculator instantiates the schemas. For example, assume we have following declarations in the input file.

```
:- sorts package.
:- variables
 P :: package.
:- constants
 p1,p2 ::package.
```

  Conceptually, the schema

$$dunk(P1) \textbf{ causes } dunked(P1)$$

  will be replaced by two $\mathcal{C}$ propositions

$$dunk(p1) \textbf{ causes } dunked(p1),$$
$$dunk(p2) \textbf{ causes } dunked(p2).$$

In practice, this instantiation process is done after the $\mathcal{C}$ schemas have been translated into the language of causal theories.

- Third, the resulting causal theory is translated into classical propositional logic, in the form of a set of clauses suitable for input to a standard satisfiability solver.

- Last, this set of clauses is combined with a set of clauses describing a possible initial state and a set of time-specific goals generated from a planning file. The model of the union of the two sets of clauses is sought. If a model is found, a plan is extracted from it and displayed. (Optionally, CCALC also verifies the plan, that is, checks that exactly one state satisfies the initial state description, and that each action as it is performed in the plan has a unique outcome, i.e., is deterministic.)

1.7 Main Issues of the Thesis

This thesis makes the following contributions.

- First, a precise account of deterministic conformant planning using action language $\mathcal{C}$ is developed to provide a theoretical basis for our approach.

- Second, we extend CCALC to make it more convenient for solving deterministic conformant planning problems. This approach is interesting for two reasons: it utilizes an expressive high-level action description language; and it solves (deterministic) conformant planning problems directly by the satisfiability method.

- Third, we provide an experimental evaluation of our approach of conformant planning, comparing to existing conformant planning systems such as Conformant GraphPlan (CGP)[SW98] and Conformant Model Based Planner (CMBP) [CR00].

The remainder of the thesis is structured as follows. A brief review of background knowledge and related research is given in Chapter 2. Chapter 3 is devoted to further explainning our approach to conformant planning, and to giving precise definitions of concepts and the language used in the thesis. In Chapter 4, we explain some implementation issues on deterministic conformant planning under the system CCalc. In Chapter 5, we present the experimental results, and in Chapter 6 we draw conclusions and discuss future research directions.

Chapter 2

Background on Conformant Planning

Past research in AI planning can be roughly divided into two camps: the logic-based common sense knowledge representing and STRIPS-style descriptions of primitive actions. On the one hand, there is long line of work concerned with the adequate representation of action domains in logic-based languages. Underlying this work is the hope that general-purpose methods for reasoning with logical formalisms will eventually provide adequate means for computing with them. On the other hand, there is a largely separate line of work in which actions are represented in relatively inexpressive languages, and special-purpose algorithms are developed to take advantage of special features of these languages.

In the seminal paper "Program with Common Sense", John McCarthy [McC59] speculated that some of the most interesting and difficult problems for artificial intelligence (AI) research would involve the kinds of knowledge and reasoning we

commonly refer to as “common sense”. In particular, he proposed the study of common reasoning about the effects of actions. In the 40 years since, automated planning has received a great deal of attention and still has far to go.

McCarthy’s paper is more widely known for the methodology it proposed: representing common sense knowledge in formal logic and reasoning with it by means of general-purpose logical methods. The use of logic-based methods to represent common sense knowledge of various kinds has interested many researchers. Over the years, the problem of adequately representing knowledge about the effects of action proved to be surprisingly difficult, but a great deal of progress has been made in the last decade.

Similarly, initial efforts to implement reasoning methods based on logic were rather discouraging. The first approaches, including influential, resolution-based the method proposed by Green [Gre69], formulated planning as deduction, in which planning is implemented as the process of finding a deductive proof of a statement that the goal conditions can be deduced from the initial conditions together with a sequence of actions.

For many years, the more promising computational approaches to planning were based on the use of variants of the STRIPS language for representing the effects of actions [FN71]. (STRIPS stands for Stanford Research Institute Planning System.) As mentioned previously, the research in STRIPS-based planning is typically concerned with special-purpose search algorithms that depend on specific properties of the STRIPS language. A noteworthy approach of classical STRIPS-style planning

is GraphPlan [BF97]. GraphPlan planners are based on compiling a STRIPS-style planning problems into a compact (polynomial size) graph structure in which information can be rapidly dispersed to aid in the search for a plan.

Logic-based deductive planning and STRIPS-style planning represent two lines of research on automated planning. Both of them have limitations. Planning using first-order resolution theorem-proving techniques can not scale up gracefully to realistically-sized domain problems. On the other hand, STRIPS-style planning is not as expressive as logic-based planning.

Recently, the situation has changed somewhat. A formal model of planning based on satisfiability rather than deduction was introduced by Kautz and Selman in [KS92] and in a subsequent paper [KS96] was shown to be surprisingly effective. Different from deductive planning in first-order logic, planning as satisfiability formalizes planning in terms of propositional satisfiability. From deduction in first order logic to propositional satisfiability, we move from an unsolvable problem complexity to an NP-problem. Another advantage is that it inherits the expressive feature of logic-based knowledge representation. Most current work on automated classical planning incorporates at least some elements at the satisfiability method.

All the planning methods we have been talking about so far, such as deduction, STRIPS-style planning and planning as satisfiability, follow the traditional view of planning and ignore uncertainty from both actions and initial states. They can be classified under that category of classical planning, in which a fundamental assumption underlying most of them is that the planning domain is deterministic and the

initial state is fully specified. Conformant planning addresses the problem of finding a plan that is guaranteed to achieve the goal regardless of the uncertainty on the initial condition and on the effect of actions. The history of work on conformant planning is not very long. It is first proposed in [GB96]. There are several conformant planning system besides the system developed in the thesis.

CGP, which stands for Conformant GraphPlan Planner, extends the ideas of GraphPlan to deal with uncertainty [SW98]. To model uncertain initial conditions, CGP is provided with a set of possible worlds so that in each possible world, the initial conditions are completely specified. A separate plan graph is initialized for each possible world, then GraphPlan procedure is run on each of these possible worlds in parallel. As with our approach, CGP does not deal with nondeterministic actions.

Conformant planning via Symbolic Model Checking (or CMBP, which stands for Conformant Model Based Planner) is based on the representation of the action domain as a finite state automaton. Symbolic Model Checking techniques are used to compactly represent the finite state automaton and efficiently implement the search steps [CR00]. In addition to its (relative) effectiveness, CMBP is notable for its handling of nondeterministic actions.

Another system for conformant planning is described in [FG00]. The approach also formalizes conformant planning in terms of propositional satisfiability, but not directly as in our approach. It mainly involves two steps. First, find a path from one possible initial state to a state that satisfies the goal. This step generates a candidate plan. The second step is that the candidate plan is checked to determine

if it is valid. Each of these steps can be accomplished by using a satisfiability solver. This system is also notable for its ability to handle a limited form of nondeterminism in action effects. (In doing so, it relies on special features of its STRIPS-based action description language.)

Chapter 3

Deterministic Conformant Planning With $\mathcal{C}$

In the following sections, we will first introduce the action description language $\mathcal{C}$, which is the representation language of the action domains we work with in the thesis. Based on the definitions related to $\mathcal{C}$, we will also give definitions of some basic concepts in classical planning and extend them to deterministic conformant planning. Examples are given to illustrate the theoretical concepts defined in the chapter.

3.1 Action Language $\mathcal{C}$

Below we first review the syntax and semantics of classical propositional logic. After that we will give a review of syntax and semantics of the action language $\mathcal{C}$. Our presentation closely follows that of [GL98b].

3.1.1 Syntax and Semantics of Propositional Logic

A *propositional signature* is a set of symbols called atoms. The symbols $True$, $False$, $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication) and $\equiv$ (equivalence) are called *propositional connectives.* $True$ and $False$ are zero-ary connectives. The $\neg$ is a unary connective and the rest are binary connectives. We will assume that propositional signatures σ do not contain the propositional connectives nor the parentheses (,). The set of propositional formulas over a signature σ is the least set X such that

- $\sigma \subseteq X$,
- $True$, $False \in$ X,
- if $F \in X$, then $\neg F \in X$,
- for any binary connective $\odot$, if F, $G \in X$, then $(F \odot G) \in X$.

An *interpretation* of a propositional signature σ is a function from σ into the set $\{\mathbf{t},\mathbf{f}\}$ of truth values. For any formula F and any interpretation I, the truth value F^I that is assigned to F by I is defined recursively as follows.

- $F^I \;=\; I(F)$ if F is an atom
- $True^I = \mathbf{t}$; $False^I = \mathbf{f}$
- $(\neg F)^I = \begin{cases} \mathbf{t}, & \text{if } F^I = \mathbf{f} \\ \mathbf{f}, & \text{otherwise} \end{cases}$
- $(F \wedge G)^I = \begin{cases} \mathbf{t}, & \text{if } F^I = \mathbf{t} \text{ and } G^I = \mathbf{t} \\ \mathbf{f}, & \text{otherwise} \end{cases}$

- $(F \vee G)^I = \begin{cases} \mathbf{t}, & \text{if } F^I = \mathbf{t} \text{ or } G^I = \mathbf{t} \\ \mathbf{f}, & \text{otherwise} \end{cases}$

- $(F \supset G)^I = \begin{cases} \mathbf{f}, & \text{if } F^I = \mathbf{t} \text{ and } G^I = \mathbf{f} \\ \mathbf{t}, & \text{otherwise} \end{cases}$

- $(F \equiv G)^I = \begin{cases} \mathbf{t}, & \text{if } F^I = G^I \\ \mathbf{f}, & \text{otherwise} \end{cases}$

If $F^I = \mathbf{t}$, we say the interpretation I satisfies F (symbolically, $I \models F$). If there exists an interpretation satisfying a formula F, we say that F is *satisfiable.* This terminology applies also to sets of formulas : A set Γ of formulas is satisfiable if there exists an interpretation that satisfies all formulas in Γ. An interpretation that satisfies F (resp. Γ) is called a *model* of F (resp. Γ).

3.1.2 Syntax and Semantics of $\mathcal{C}$

Syntax

Consider a propositional signature σ, partitioned into the fluent names σ^{fl} and the action names σ^{act}.Intuitively, fluent symbols are used to describe the facts that hold in a state of the world, and action symbols describe occurrences of actions that take the world from one state to another. By *formula* we mean a propositional formula over signature σ. By *fluent formula* we mean a propositional formula over σ^{fl}. By *action formula* we mean a propositional formula over σ^{act}.

Action language $\mathcal{C}$ has two kinds of propositions, static laws and dynamic laws, defined as follows. Assume F and G are fluent formulas and H is an action formula.

Static laws have the form

$$\textbf{caused } F \textbf{ if } G. \tag{3.1}$$

An example of a static law for the "bomb in toilet" problem, as we have seen in Chapter 1, is

$$\textbf{caused } \neg armed \textbf{ if } dunked(p1) \;\wedge\; in(p1)$$

which means if the package p1 is dunked and if a bomb is in the package, then there is a cause for the bomb to be disarmed. Dynamic laws have the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H. \tag{3.2}$$

We will soon see a number of examples of dynamic laws.

The following are abbreviations commonly used in the language $\mathcal{C}$.

- An expression of the form

  $$\textbf{inertial } F \tag{3.3}$$

  where F is a fluent formula, stands for the dynamic law

  $$\textbf{caused } F \textbf{ if } F \textbf{ after } F.$$

  This abbreviation is used to express that a fluent F tends to keep the value it had before. An example is

  $$\textbf{inertial } clogged.$$

- If $F_1, ..., F_k$ are fluent formulas,

  $$\textbf{inertial } F_1, ..., F_k \tag{3.4}$$

  stands for the dynamic laws

$$\textbf{inertial } F_1$$
$$\vdots$$
$$\textbf{inertial } F_k.$$

- An expression of the form

$$\textbf{always } F \tag{3.5}$$

where F is fluent formula, stands for the static law

$$\textbf{caused } False \textbf{ if } \neg F.$$

An example is

$$\textbf{always } in(p1) \vee in(p2).$$

- An expression of the form

$$\textbf{never } F \tag{3.6}$$

where F is fluent formula, stands for the static law

$$\textbf{caused } False \textbf{ if } F.$$

An example is

$$\textbf{never } in(p1) \wedge in(p2).$$

- An expression of the form

$$U \textbf{ causes } F \textbf{ if } G \tag{3.7}$$

where U is an action formula and F, G are fluent formulas, stands for the dynamic law

$$\textbf{caused } F \textbf{ if } True \textbf{ after } G \wedge U.$$

An example is

$$dunk(P) \textbf{ causes } \neg armed \textbf{ if } in(P).$$

- An expression of the form

$$\textbf{nonexecutable } U \textbf{ if } F \tag{3.8}$$

where U is an action formula and F is a fluent formula, stands for the dynamic law

$$\textbf{caused } False \textbf{ after } F \wedge U.$$

An example is

$$\textbf{nonexecutable } dunk(P1) \textbf{ if } clogged.$$

An *action description* is a set of static and dynamic laws.

Semantics

An *action* is defined as an interpretation of σ^{act}. Intuitively such an action a represents the concurrent execution of the set of "atomic" actions corresponding to the action symbols that are true in the interpretation a. By having this point of view, the action language $\mathcal{C}$ inherently has the ability of representing concurrent actions.[1] A *state* is an interpretation of σ^{fl} that satisfies the fluent formula $G \supset F$ for every static law

[1] The word "action" here is understood intuitively.

caused F **if** G

in action description D.

A *transition* is any triple $< s, a, s' >$ where s, s' are states and a is an action. We say s is the *initial state* of the transition, and s' is its *resulting state.* A formula F is *caused* in a transition $< s, a, s' >$ if

- there is a static law

  caused F **if** G

  in D such that $s' \models G$, or

- there is a dynamic law

  caused F **if** G **after** H

  in D such that $s' \models G$ and $s \cup a \models H$.

A transition $< s, a, s' >$ is *causally explained* according to D if its resulting state s' is the only interpretation of σ^{fl} that satisfies all formulas caused in this transition.

The *transition system* described by an action description D is the labeled directed graph whose vertices are the states of D, with an edge leading from s to s' labeled a for each casually explained transition $< s, a, s' >$.

A *history* for action description D is a path in the corresponding transition diagram: that is, a finite sequence $< s_0$, a_0, s_1, a_1,..., s_{T-1}, a_{T-1}, $s_T >$ such that s_0

is a state and for all i $(0 \leq i < T)$, $< s_i,\ a_i,\ s_{i+1} >$ is a causally explained transition.

We say the action description is *deterministic* if for each state s and action a, there is at most one s' such that the transition $< s, a, s' >$ is causally explained.

3.1.3 Examples of $\mathcal{C}$

Assume that we have action symbols $dunk(p1)$ and $dunk(p2)$ and fluent symbols $armed$, $in(p1)$ and $in(p2)$. The action description for the scenario of the BT domain problem we mentioned in Chapter 1 can be rewritten in $\mathcal{C}$ as follows:

$$\textbf{inertial}\ in(p1),\ \neg in(p1),\ in(p2),\ \neg in(p2),\ armed,\ \neg armed \tag{3.9}$$

$$\textbf{always}\ in(p1) \vee in(p2) \tag{3.10}$$

$$\textbf{never}\ in(p1) \wedge in(p2) \tag{3.11}$$

$$dunk(p1)\ \textbf{causes}\ \neg\ armed\ \textbf{if}\ in(p1)$$

$$dunk(p2)\ \textbf{causes}\ \neg\ armed\ \textbf{if}\ in(p2)$$

$$\textbf{nonexecutable}\ dunk(p1) \wedge dunk(p2) \tag{3.12}$$

We sometimes represent such a description somewhat more succinctly by using meta-variables, as follows. For example, if P1 is a meta-variable ranging over $\{p1, p2\}$, we can write

$$dunk(P1)\ \textbf{causes}\ \neg armed\ \textbf{if}\ in(P1) \tag{3.13}$$

in place of

$$dunk(p1)\ \textbf{causes}\ \neg\ armed\ \textbf{if}\ in(p1)$$

and

$$dunk(p2)\ \textbf{causes}\ \neg\ armed\ \textbf{if}\ in(p2).$$

(3.9) is an abbreviation of dynamic laws. It describes the inertial feature of fluents in this domain. (3.10) describes a constraint on states: a bomb is in at least one of the packages. (3.11) says the bomb is in at most one package. (3.12) describes that ("atomic") actions can not be executed concurrently. (3.13) describes the causal relation between the action $dunk(P1)$ and the fluents $armed$ and $in(P1)$: essentially dunking the package with bomb disarms the bomb.

The transition system of (3.9)-(3.13)is shown in Figure 3.1. Note that in this figure, we use $\{dunk(p1)\}, \{dunk(p2)\}$, $\{\}$ to represent three distinct actions: each action is represented by the set of action symbols that are true in it. Thus, intuitively the label $\{\}$ stands for the case in which no actions are performed.

As shown in this figure, each state in the transition system is determined by fluent formulas $armed$, $in(p1)$, $in(p2)$ or their negations. The fluent formulas in State1 indicate that the bomb is in package 1, not in package 2, and the bomb is armed. In this case, dunk package 2 doesn't change the state while dunk package 1 disarms the bomb and leads to State3, which is determined by fluent formulas $\neg armed$, $in(p1)$, and $\neg in(p2)$. State2 indicates that a bomb is in package 2. In this case, dunk package 1 doesn't change the state while dunk package 2 disarms the bomb and leads to State4. We can apply the dunk action to either package again at State3 and State4, but nothing changes.

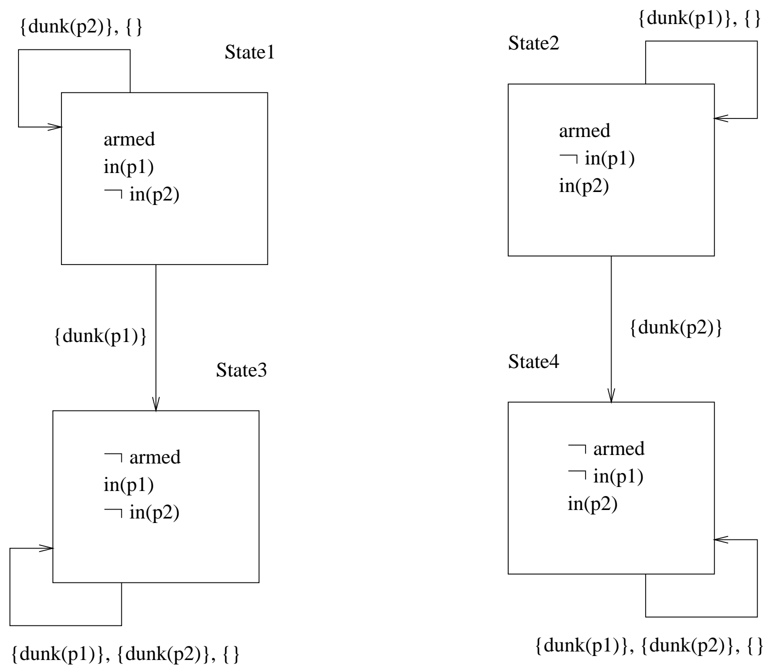

Figure 3.1: A Simplified Transition System of BT Domain: 2 packages and 1 toilet

Transition $< State1,\ \{dunk(p1)\},\ State3 >$ is an example of a causally explained transition in D because the formulas caused in this transition are $\neg armed$, $in(p1)$, $\neg in(p2)$ and State3 is the only state that satisfies all these formulas. Transition $< State1, \{dunk(p2)\}, State2 >$ is not causally explained since the only formula caused in this transition is *armed* and it is satisfied by more than one state (namely, State1 and State2).

Following we give an example of an input file to CCALC which describes the BT domain problem with two packages.

```
% File: 'BT.h'
:-macros maxtime->2.
:-include 'C.h'.
:-sorts package.

:-variables
 P1, P2 :: package.

:-constants
p1,p2 :: package;
dunk(package) :: action;
armed   :: inertialFluent;
in(package)             :: inertialFluent.

always \/P1: in(P1).
never in(P1) && in(P2) && -(P1=P2).
caused -armed if dunked(P1) & in(P1).
nonexecutable dunk(P1) && dunk(P2) && -(P1=P2).
```

Here the declaration:

```
:- macros maxtime ->2.
```

tells CCALC that we are interested in history of length 2. The delaration

```
:-include 'C.h'.
```

tells CCALC to include with this description a "standard" file `C.h` which includes additional macros and other information that CCALC uses to convert a $\mathcal{C}$ language description into the internal (causal theories) language of CCALC, from which it is subsequently translated into classical propositional logic. Declaring a fluent F as `inertialFluent` implies the $\mathcal{C}$ proposition

$$\textbf{inertial}\ F,\ \neg F$$

For example, the declaration `armed :: inertialFluent;` declares *armed* as a fluent, and, moreover, can be understood to indicate that the $\mathcal{C}$ description also includes the $\mathcal{C}$ propositions

$$\textbf{inertial}\ armed,\ \neg armed.$$

The symbol \/ is used to abbreviate a finite disjunction. In this case, the proposition

$$\textbf{always} \bigvee P1 :\ in(P1) \tag{3.14}$$

stands for

$$\textbf{always}\ in(p1)\ \vee\ in(p2)$$

This file is written in a manner that makes it easy to add additional packages — simply declare additional constants of sort package, for instance, adding the declaration

```
:-constants
  p3,p4 :: package.
```

would yield a description of the BT domain with 4 packages instead of 2. In that case, the proposition 3.14 stands for

$$\textbf{always}\ in(p1)\ \vee\ in(p2)\ \vee in(p3)\ \vee\ in(p4)$$

Below we show example output obtained from CCALC, which shows a request for CCALC to find a model of the input description, i.e., a history of length 2.

```
| ?- sat.

calling sato...
  run time (seconds)                  0.01

0.  armed  in(p1) -in(p2)
1.  armed  in(p1) -in(p2)
2.  armed  in(p1) -in(p2)
```

Referring back to the transition diagram as shown in Figure 3.1, the model displayed in this input corresponds to the history $< State1, \{\}, State1, \{\}, State1 >$.

We next extend the action description so that it includes the case that the dunk actions cause the toilet to be clogged. We call it "bomb in toilet with clogging" or BTC domain. To make the toilet to be available for dunk again, an action $flush$ is added into the domain. To specify the states, fluents $inBowl(P1)$, $dunked(P1)$, and $clogged$ are added. We specify a sequence of schemas as follows. Note that $P1$ and $P2$ are meta-variables over all the packages.

$$\textbf{inertial}\ in(P1),\ \neg in(P1),\ inBowl(P1),\ \neg inBowl(P1),\ dunked(P1) \tag{3.15}$$

$$\textbf{inertial}\ \neg dunked(P1),\ armed,\ \neg armed,\ clogged,\ \neg clogged \tag{3.16}$$

$$\textbf{always} \bigvee P1:\ in(P1) \tag{3.17}$$

$$\textbf{never}\ in(P1) \wedge in(P2)\ \ (P1\ \neq\ P2) \tag{3.18}$$

$$dunk(P1)\ \textbf{causes}\ inBowl(P1) \tag{3.19}$$

$$\textbf{caused}\ dunked(P1)\ \textbf{if}\ inBowl(P1) \tag{3.20}$$

$$\textbf{caused } clogged \textbf{ if } inBowl(P1) \tag{3.21}$$

$$\textbf{caused } \neg armed \textbf{ if } in(P1) \wedge dunked(P1) \tag{3.22}$$

$$\textbf{nonexecutable } dunk(P1) \textbf{ if } clogged \tag{3.23}$$

$$\textbf{nonexecutable } dunk(P1) \textbf{ if } dunked(P1) \tag{3.24}$$

$$flush \textbf{ causes } \neg inBowl(P1) \tag{3.25}$$

$$\textbf{always } clogged \ \equiv \ \bigvee P: \ inBowl(P) \tag{3.26}$$

$$flush \textbf{ causes } \neg clogged \tag{3.27}$$

$$\textbf{never } inBowl(P1) \wedge inBowl(P2) \ \ (P1 \ \neq \ P2). \tag{3.28}$$

Figure 3.2 illustrates one of the transition systems described by these schemas: the one in which we assume that there are two packages, $P1$ and $P2$.

It is interesting to notice that the domain description reflects some of the expressive power of the action language $\mathcal{C}$. For instance, (3.19) and (3.25) implicitly eliminate the possibility of actions dunk and flush happening at the same time by having contradicting effects of the two actions. Also, by saying (3.21) and (3.19), we imply that *clogged* is an "indirect" effect of the action $dunk(P1)$. Its "direct" effect is $inBowl(P1)$.

The domain description (3.15)-(3.28) is deterministic, since in any state in the transition system, execution of any of the actions leads to at most one resulting state. The domain description is still a slightly simplified version of the one used to evaluate the performance of our approach to deterministic conformant planning. In the general case, we consider "bomb in toilet with clogging" domains in which there may be more than one toilet.

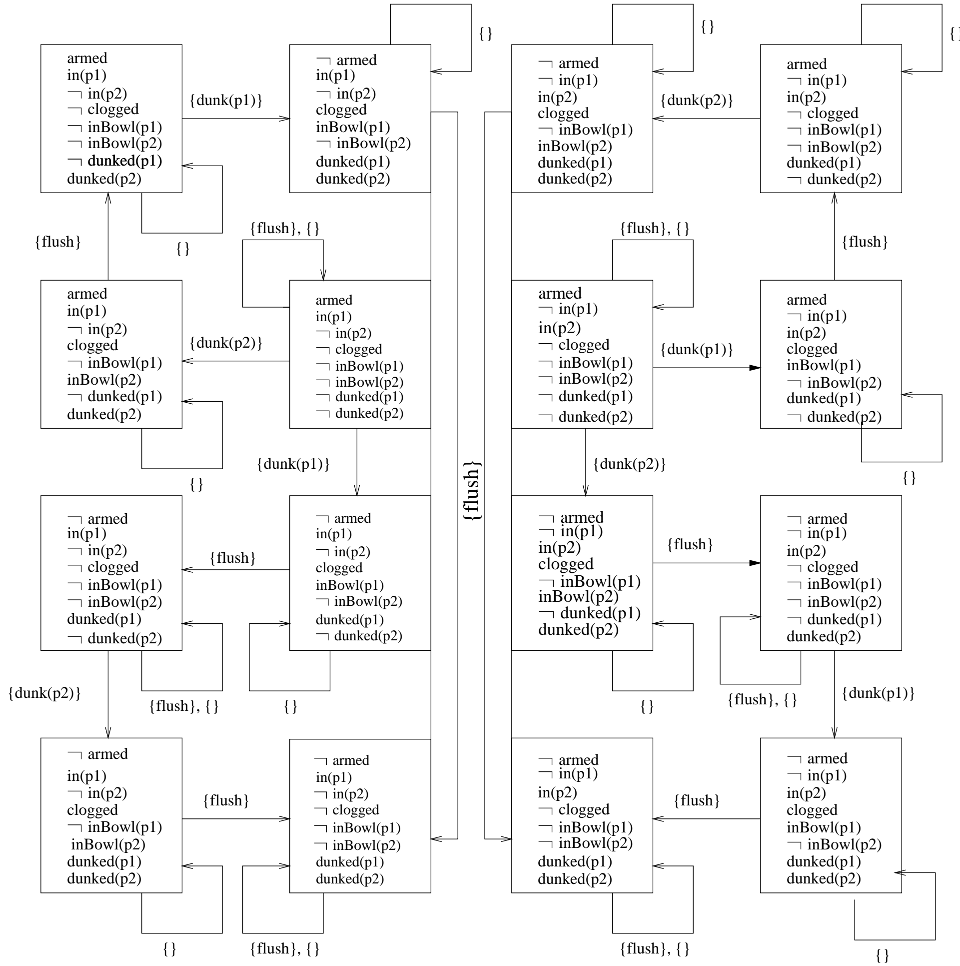

Figure 3.2: A Complete Transition System of BTC Domain Problems: 2 packages and 1 toilet

3.2 Classical Planning

3.2.1 Definitions in Classical Planning

A *planning problem* for D is a triple $< I, D, G >$, where I and G are fluent formulas, known as the initial state description and goal description, respectively. A planning problem $< I, D, G >$ is *classical* if I is satisfied by exactly one state of D and D is deterministic.

A *plan* of length T is a finite sequence $< a_0, a_1, ..., a_{T-1} >$ of actions.

A plan $< a_0, a_1, ..., a_{T-1} >$ is valid for a classical planning problem $< I, D, G >$ if there is a history $< s_0$, a_0, s_1, a_1,..., s_{T-1}, a_{T-1}, $s_T >$ such that $s_0 \models I$ and $s_T \models G$.

3.2.2 Examples of Classical Planning

Given the BT domain description shown in Section 3.1.3, we give an example of classical planning by showing the execution results of the system CCALC. First, we pose a planning file describing an initial condition and a desired goal as below.

```
%File 'BT.p'
:- plan
facts ::
 0: armed,
 0: in(p1),
 0: -in(p2);

goal ::
 2: -armed.
```

It describes the initial world: package p1 contains the bomb; package 2 doesn't contain the bomb; the bomb is armed. It also describes a goal—not armed —which

is to hold at time 2. The following result shows that after a standard satisfiability solver `sato` is called, a history of length 2 is found and displayed.

```
calling sato...
  run time (seconds)                  0.00

0.  armed  in(p1) -in(p2)

1.  armed  in(p1) -in(p2)

ACTIONS: dunk(p1)

2.  in(p1) -armed -in(p2)
```

In this case, the plan is $< \{\}, \{dunk(p1)\} >$.

To correspond to the domain description of BTC problem given in the previous section, we also give a planning file and show a classical plan of the domain problem.

```
:- plan
facts ::
 0: armed,
 0: -clogged,
 0: -dunked(P1)
 0: -inBowl(P1),
 0: in(p1),
 0: -in(p2);

goals ::
 2: (-armed && -clogged).
```

From above description, we know that initially the world is armed and toilet is not clogged. All the packages are not dunked and also not in the bowl of the toilet. The bomb is in package 1, not in package 2. The goal we want to achieve at time 2 is that the bomb is disarmed and toilet is not clogged. By the same procedures we have described, a plan is found and shown below.

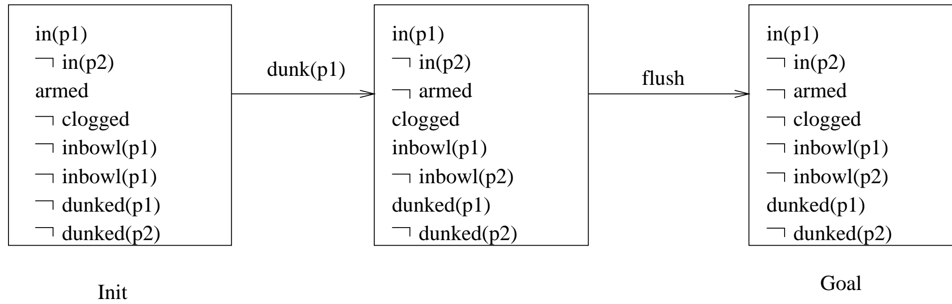

Figure 3.3: A World History of BTC Domain Problem in Classical Planning

```
calling sato...
  run time (seconds)                 0.02


0. armed in(p1) -clogged -dunked(p1) -dunked(p2) -in(p2)
   -inBowl(p1) -inBowl(p2)

ACTIONS: dunk(p1)

1. clogged  dunked(p1) in(p1) inBowl(p1) -armed -dunked(p2)
   -in(p2) -inBowl(p2)

ACTIONS: flush

2. -clogged  dunked(p1) in(p1) -inBowl(p1) -armed -dunked(p2)
   -in(p2) -inBowl(p2)
```

The plan in this case is the action sequence $< \{dunk(p1)\}, \{flush\} >$. Figure 3.3 shows the world history of the BTC domain problem corresponding to this solution of the planning problem.

3.3 Deterministic Conformant Planning

3.3.1 Definitions in Conformant Planning

In conformant planning, the actions may be non-deterministic and the initial state may be incompletely specified. In this thesis, to simplify the problem, we assume that actions are deterministic, and we call this *deterministic conformant planning.*

As before, a planning problem is a triple $< I, D, G >$. Initial state description I, domain description D and goal description G are as they were for classical planning, except we no longer assume that I is satisfied by exactly one state. We say a plan $< a_0, a_1, ..., a_{T-1} >$ is *valid* for deterministic conformant planning problem $< I, D, G >$ if for each state s_0 such that $s_0 \models I$, there is a history $< s_0$, a_0, s_1, a_1,..., s_{T-1}, a_{T-1}, $s_T >$ such that $s_T \models G$.

Figure 3.4 depicts a history view of deterministic conformant planning. I is initial state description, D is domain description, and G is goal description. We see that I is satisfied by a set of initial states (in this case, 2 initial states). Same action sequence $\{dunk(p1)\}$, $\{dunk(p2)\}$ applies to all the possible worlds. Goal is also satisfied by a set of states.

Next chapter, we will discuss some of the implementation issues of deterministic of conformant planning.

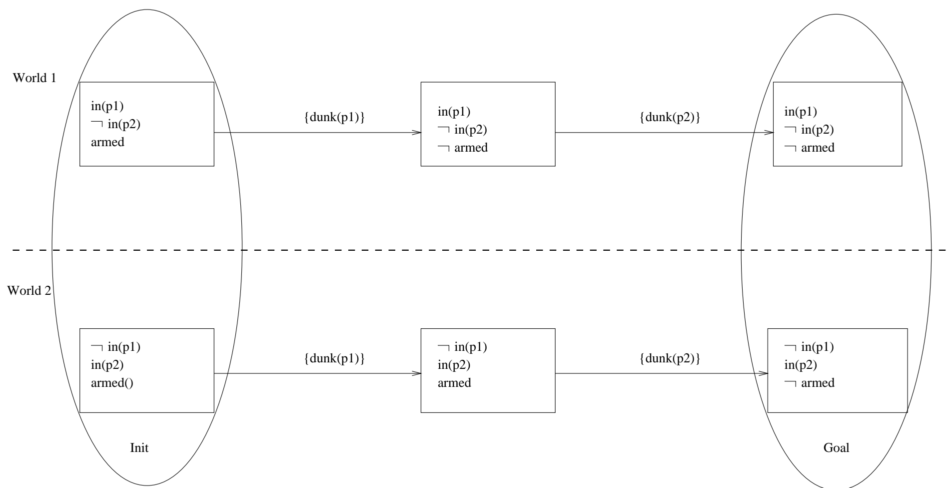

Figure 3.4: A World History of BTC Domain Problem in Deterministic Conformant Planning

Chapter 4

Implementing Deterministic Conformant Planning as Satisfiability

As we mentioned, the idea is very simple. Since we are concerned with a number of possible worlds, instead of having a $\mathcal{C}$ domain description describing the effects of actions in a single world, we want to have a $\mathcal{C}$ description describing the effects of actions in all the possible worlds. A very intuitive approach is taken, that is, adding a world argument represented by a schematic variable W to each fluent atom. By doing so, roughly speaking, each fluent formula has the ability to indicate which possible world it holds in, and the $\mathcal{C}$ domain description thereby is able to describe the effects of actions in all the possible worlds. To achieve this, we don't need to make any changes in the syntax or the semantics of $\mathcal{C}$.

Following is an example of an action description of BTC domain problem, in

which these extra world arguments are added explicitly. (This is not ultimately how we implement the approach, but is included to help explain the idea.) Compared to the schemas (3.15) – (3.28), we can see that the only difference is that each fluent atom has an added W argument, which ranges over "possible worlds". (There will be one such possible world for each possible initial state in the relevant planning problem.)

$$\textbf{inertial}\ in(P1,W),\ \neg in(P1,W),\ inBowl(P1,W) \tag{4.1}$$

$$\textbf{inertial}\ \neg inBowl(P1,W),\ dunked(P1,W),\ \neg dunked(P1,W) \tag{4.2}$$

$$\textbf{inertial}\ armed(W),\ \neg armed(W),\ clogged(W),\ \neg clogged(W) \tag{4.3}$$

$$\textbf{always}\ \bigvee P1:\ in(P1,W) \tag{4.4}$$

$$\textbf{never}\ in(P1,W)\ \wedge\ in(P2,W)\ (P1\ \neq\ P2) \tag{4.5}$$

$$dunk(P1)\ \textbf{causes}\ inBowl(P1,W) \tag{4.6}$$

$$\textbf{caused}\ dunked(P1,W)\ \textbf{if}\ inBowl(P1,W) \tag{4.7}$$

$$\textbf{caused}\ clogged(W)\ \textbf{if}\ inBowl(P1,W) \tag{4.8}$$

$$\textbf{caused}\ \neg armed(W)\ \textbf{if}\ in(P1,W) \wedge dunked(P1,W) \tag{4.9}$$

$$\textbf{nonexecutable}\ dunk(P1)\ \textbf{if}\ clogged(W) \tag{4.10}$$

$$\textbf{nonexecutable}\ dunk(P1)\ \textbf{if}\ dunked(P1,W) \tag{4.11}$$

$$flush\ \textbf{causes}\ \neg inBowl(P1,W) \tag{4.12}$$

$$\textbf{always}\ clogged(W)\ \equiv\ \bigvee P1:\ inBowl(P1,W) \tag{4.13}$$

$$flush\ \textbf{causes}\ \neg clogged(W) \tag{4.14}$$

$$\textbf{never}\ inBowl(P1, W) \wedge inBowl(P2, W)\ \ (P1 \neq P2) \tag{4.15}$$

Although this syntax modification is not difficult, we would like to be able to do conformant planning using existing action domain descriptions. To make action descriptions consistent in both classical planning and deterministic conformant planning, we want to hide the fact that an extra argument is added into fluent atoms when we wish to solve a conformant planning problem.

As we mentioned in Chapter 2, CCALC translates a $\mathcal{C}$ input file into schemas of the internal language of the CCALC, the language of causal theories. This is implemented by a macro facility. The macros used for this step are contained in a standard file `C.h` and they are made available to CCALC by including the declaration

```
:-include 'C.h'.
```

in the domain description file, as mentioned previously. To make CCALC to be more friendly to conformant planning, we have prepared a similar standard file to be used for conformant planning. This new file is called `CC.h`. It is very similar to `C.h`, but now we add world argument to fluent atoms in the process of translating from $\mathcal{C}$ to the language of causal theories. See Appendix for file `CC.h`.

Next we show the changes we made on the two major macros. The original macros are

```
caused #1 if #2 after #3
          -> (next(_N,_T) && h(#3,_N) && h(#2,_T) => h(#1,_T)) ;
caused #1 if #2
```

```
          -> ( (h(#2,0) ->> h(#1,0))
             ; (_T>0 && h(#2,_T) => h(#1,_T)) ) ;
```

The first macro replaces a $\mathcal{C}$ language schema (for dynamic laws) with a corresponding schema in the language of causal theories. The first macro's right-hand side first defines that _N is the next time point after _T and then it takes the first two patterns matched, adds a time argument _T to them and adds _N to the third pattern. The pattern `h(#1,T)` intuitively says that the first argument of h holds at time T. Roughly speaking, this process creates fluent expressions suitable for use in the language of causal theories, and produces a schema in the language of causal theories. The second macro does a similar thing except that it translates one schema in $\mathcal{C}$ (for static laws) into two schemas of the language of causal theories: one for time 0 and another for all subsequent times.

In order to add the argument W to the fluents of causal theories during the process of translation, we change the previous two macros into:

```
caused #1 if #2 after #3
          -> (next(_N,_T) && h(n(#3,W),_N) && h(n(#2,W),_T)
                => h(n(#1,W),_T)) ;
caused #1 if #2
          -> ( (h(n(#2,W),0) ->> h(n(#1,W),0))
             ; (_T>0 && h(n(#2,W),_T) => h(n(#1,W),_T)) ) ;
```

Intuitively, the pattern `h(n(#1,W),T)` says that the first argument of `n` holds at time `T` in possible world `W`.

In modifying the standard file `C.h` to obtain `CC.h`, a number of other changes were required, but the two macros discussed here illustrate the main idea.

This is essentially all that was required to allow us to use standard $\mathcal{C}$ input files for conformant planning. We did make a few minor changes to the main prolog program `ccalc.pl` that implements most features of CCALC. To indicate the difference of the modified system to the original CCALC, we gave it a new name C_CCALC. In particular, C_CCALC accepts input files with a `.w` extension, and for such input files, a more suitable output format is used for display of planning results.

For better understanding the C_CCALC system, we include an example domain description file `BinT.w` in Figure 4.1 and a plan file in Figure 4.2. It is worth mentioning that in the file `BinT.w`, two macros have to be defined before the standard `CC.h` file is included: one is `maxtime`, which defines the range of times {`0...maxtime`} to be considered; the other one is `maxworld`, which defines the range of possible world {`1...maxworld`} needed for the conformant planning problem. At the end, we show sample output in Figure 4.3.

In the current implementation, the format for planning files is a bit awkward, because we must use the "internal" representation for fluents. For instance, we write $n(\mathit{clogged}, W)$ to express that the toilet is clogged in possible world W. Also, we require not only that the number of possible initial states be explicitly given (via the `maxworld` declaration) but also that each individual possible initial state be completely specified. For example, by instantiating the metavariable W in the fact $n(\mathit{armed}, W)$ at time 0 shown in Figure 4.2, we specify that all the initial states are

armed at time 0, which is reflected in the output shown in Figure 4.3. Other initial facts listed in Figure 4.2 are instantiated at the same way so that each individual initial state is fully described — that is, there is exactly one state that satisfies the conditions described.

Finally, we briefly mentioned in Chapter 1 that CCALC can verify a classical plan automatically by checking that exactly one state satisfies the initial state description and that each action as executed in the plan has a deterministic effect. The same mechanism works for deterministic conformant planning because, after each of the possible initial states is fully described, the verification procedure can be done as it is done for classical planning problems. In effect, the system will check that (i) each of the possible initial states is completely described, and (ii) each action as it is performed in each possible world has a deterministic outcome.

```
% File: 'BinT.w'

:- macros numPackages -> 4.
:- macros numToilets -> 2.
:- macros maxtime -> 3.
:- macros maxworld -> numPackages.

:- include 'CC.h'.
:- sorts package; toilet.
:- variables
  P,P1 :: package;
  T,T1 :: toilet.

:- constants
  1..numToilets                :: toilet;
  1..numPackages               :: package;
  flush(toilet),
  dunk(package,toilet)         :: action;
  in(package),
  armed,
  inBowl(package,toilet),
  clogged(toilet),
  dunked(package)        :: inertialFluent.

:- show armed/0; clogged/1.

  constant in(P).
  always \/P:in(P).
  never in(P) && in(P1) && -(P=P1).
  never inBowl(P,T) && inBowl(P,T1) && -(T=T1).
  never inBowl(P,T) && inBowl(P1,T) && -(P=P1).
  dunk(P,T) causes inBowl(P,T).
  nonexecutable dunk(P,T) if clogged(T).
  nonexecutable dunk(P,T) if dunked(P).
  flush(T) causes -inBowl(P,T).
  flush(T) causes -clogged(T).
  caused clogged(T) if inBowl(P,T).
  caused dunked(P) if inBowl(P,T).
  caused -armed if dunked(P) & in(P).
```

Figure 4.1: An Input File for the BMTC Domain: 4 packages; 2 toilets

```
% File: 'BinT.p'
:- variables
  W :: world.

:- plan
facts ::
  0 : P=W->>n(in(P),W),
  0 : n(-clogged(T),W)),
  0 : n(-dunked(P),W)),
  0 : n(armed,W);
goals ::
  maxtime : n(-armed,W).
```

Figure 4.2: A Planning File for the BMTC Domain

```
% 334 atoms, 2212 rules, 2120 clauses (60 new atoms), 6680 msec.
(6680 ms) yes
| ?- plan(0).
calling sato...
  run time (seconds)                 0.01

 **** Time 0 ****

World 1. armed
World 2. armed
World 3. armed
World 4. armed

ACTIONS: dunk(1,2) dunk(4,1)

 **** Time 1 ****

World 1. clogged(1)  clogged(2)
World 2. clogged(1)  clogged(2)  armed
World 3. clogged(1)  clogged(2)  armed
World 4. clogged(1)  clogged(2)

ACTIONS: flush(1) flush(2)

 **** Time 2 ****

World 1.
World 2. armed
World 3. armed
World 4.

ACTIONS: dunk(2,2) dunk(3,1)

 **** Time 3 ****

World 1. clogged(1)  clogged(2)
World 2. clogged(1)  clogged(2)
World 3. clogged(1)  clogged(2)
World 4. clogged(1)  clogged(2)
```

Figure 4.3: Sample Output for the BMTC Domain Problem: 4 packages and 2 toilets

Chapter 5

Experimental Results and Analysis

In this chapter, we present the results of the experiments and also give the evaluation and analysis of the approach described in the previous chapters.

5.1 Results

The conformant planners which are most significant for comparison with our system are CMBP [CR00] and CGP [SW98]. The system C_Plan [FG00] only provides the performance data (CPU time) in the case of toilet number at most 4 and package number at most 6. More importantly, in order to obtain acceptable performance with C_Plan, the action domain description is augmented with plan-specific heuristics, such as having a constraint like "Toilet can not be flushed if it is not clogged". These constraints are mainly for improving their performance in the particular case, which should not be considered for demonstrating the effectiveness of their approach.

This is another reason that we believe their data of performance is not comparable with other planners. However, their results are included in the tables below for reference.

CGP extends the idea of GraphPlan [BF97] to deal with uncertainty. As an offspring of GraphPlan, it also has the ability of dealing with actions concurrently, as long as the effects of those actions cannot interact. For example, in the BMTP domain problems, assume we have 3 toilets. Three dunk operations can be done in parallel at the first time, three flush operations can be performed in parallel at the next time, and so on. As a result, the depth of search required to find a plan is significantly lower than for those planners without the ability to handle concurrency. CGP returns a plan of minimal length, if one is found.

CMBP is based on model checking techniques. The algorithm takes advantage of the symbolic representation rooted from Binary Decision Diagram. The algorithm is based on a breadth-first, backward search and returns a conformant plan of minimal length, if a solution to the planning problem exists.

The evaluation was performed by comparison of the parallel results from the three approaches — CGP, CMBP and C_CCALC — on a range of "bomb in the toilet" problems. The results for CGP and CMBP are quoted from [CR00], and were obtained on a SUN Ultra Sparc 128Mb RAM running Solaris. To obtain comparable results, our tests are also run on Sun Ultra Sparc 128M RAM. The results shown in the thesis for C_CCALC are the average values of 10 runs. The domain description used in the testing is shown in 4.1, Parameters such as `numPackages`,

numToilets, maxtime, and maxworld are changed accordingly. The performance of the three systems is reported in tables listing only the solution time or search time. In the following tables, the time is expressed in seconds. The symbol "—" in the tables means "can not find a plan in a reasonable length of time".[1] We run C_CCALC under Gnu Prolog to do all the tests.

Bomb in Toilet with Clogging (BTC)

The first domain problem we work on is the "bomb in toilet with clogging" (BTC). As we mentioned in Chapter One, in BTC domain problems, dunk a package always clogs the toilet, flushing can remove the clogging, and toilet is not clogged is a precondition of dunk a package. In this case, we assume that there is only one toilet available for dunk. The test results for this problem, together with the data from CGP and CMBP on the same problems, are listed in table 5.1.

Based on our experimental results, we can see that CGP performance degraded gradually with the increase of number of packages. CMBP performs better than CGP, especially in the case of large package number. C_CCALC has slightly better performance than CMBP when number of packages is 9 or less. However it zooms up if the package number is greater than 9.

Figure 5.1 gives us a demonstration of the points we make here.

Bomb in the Multiple Toilets with Clogging (BMTC)

[1] We assume reasonable length of time is 7200 seconds or two hours.

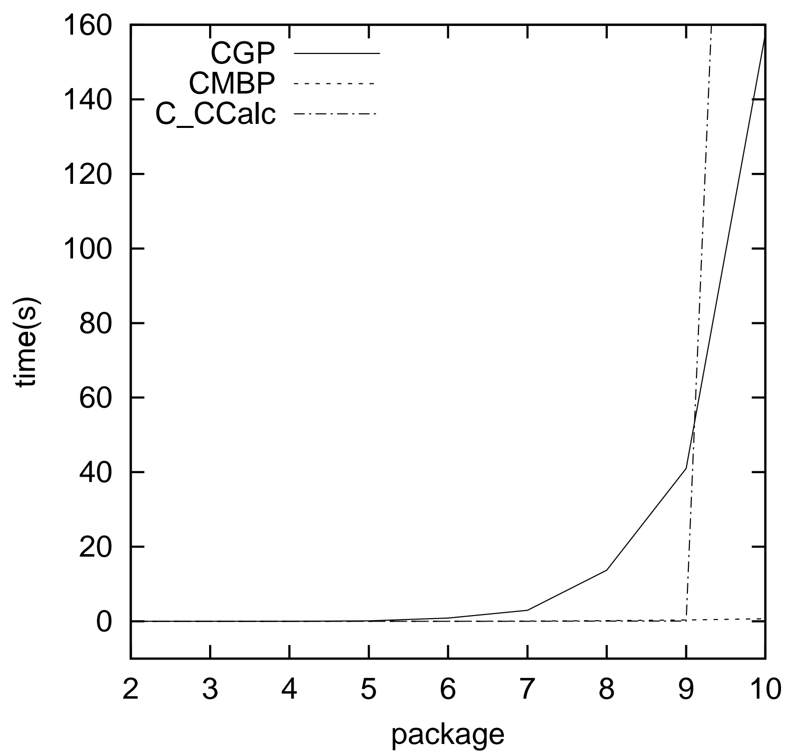

Figure 5.1: A Performance Comparison of BTC Problem

# of Toilet	# of package	$\mathcal{C}$_plan	CGP	CMBP	C_CCalc
1	2	0.00	0.000	0.010	0.000
	3	0.00	0.010	0.000	0.000
	4	0.01	0.030	0.010	0.000
	5	0.03	0.130	0.020	0.000
	6	0.06	0.860	0.020	0.020
	7	N/A	2.980	0.080	0.027
	8	N/A	13.690	0.160	0.036
	9	N/A	41.010	0.330	0.050
	10	N/A	157.590	0.700	—
	16	N/A	—	99.800	—

Table 5.1: Results for BTC Problems

BMTC is the more general case in which there could be more than one toilet available for dunk. It intrinsically adds parallel feature to the problem, which makes it possible to utilize the capability of handling concurrency in $\mathcal{C}$. For example, in the case of two toilets and two packages, the action $dunk(p1, t1)$ and action $dunk(p2, t2)$ can happen at the same time.

In paper [CR00], three versions of the problem with different degrees of uncertainty in the initial states are considered. In the first version with "Low Uncertainty", the only uncertainty is the position of bomb which is unknown, while toilets are known to be not clogged. The "Mid Uncertainty" is the second version in which every odd toilet assumed to have possibility of either clogged or not clogged. In the third version, which has "High Uncertainty", every toilets can be either clogged or

# of Toilet	# of package	C_plan	CGP	CMBP	C_CCALC
2	2	0.00	0.000	0.000	0.000
	3	0.01	0.020	0.000	0.014
	4	0.03	0.030	0.020	0.023
	5	0.06	1.390	0.030	0.043
	6	0.10	3.490	0.070	0.045
	7	N/A	508.510	0.180	0.140
	8	N/A	918.960	0.400	0.193
	9	N/A	—	0.940	0.315
	10	N/A	—	1.820	1.883

Table 5.2: Results for BMTC Problems (2t)

not. We choose the first version as our comparison target.

We have tested all the cases having from 2 to 10 packages and from 2 to 6 toilets. All the results of these tests, together with the data from CGP and CMBP, are listed in Tables 5.2-5.6.

From above tests, we choose the two cases in which number of toilets is 3 and 6 respectively to make some comments on the relative performance of CGP, CMBP and C_CCALC. In both cases, CGP has difficulty with larger numbers of package. It slows down significantly as the number of packages grows beyond six. Figure 5.2 (Left) shows that C_CCALC outperforms CMBP slightly on the smaller problem, but basically they are at the same level. However, as the number of toilets is increased, the gap is widened considerably. See Figure 5.2 (Right).

# of Toilet	# of package	C_plan	CGP	CMBP	C_CCALC
3	2	0.00	0.010	0.000	0.000
	3	0.01	0.010	0.010	0.010
	4	0.06	0.110	0.030	0.017
	5	0.08	0.170	0.080	0.111
	6	0.14	0.340	0.230	0.201
	7		6248.010	0.560	0.345
	8		—	1.300	0.691
	9		—	3.330	1.246
	10		—	7.280	1.572

Table 5.3: Results for BMTC Problems (3t)

#of Toilet	# of package	CGP	CMBP	C_CCALC
4	2	0.000	0.010	0.010
	3	0.010	0.020	0.015
	4	0.010	0.060	0.028
	5	0.500	0.190	0.050
	6	1.160	0.410	0.139
	7	2.410	1.041	0.466
	8	8.540	2.740	0.848
	9	—	6.690	1.389
	10	—	14.420	5.686

Table 5.4: Results for BMTC Problems (4t)

# of Toilet	# of package	CGP	CMBP	C_CCALC
5	2	0.010	0.010	0.011
	3	0.020	0.040	0.028
	4	0.020	0.150	0.038
	5	0.050	0.490	0.052
	6	5.920	1.300	0.074
	7	18.410	3.990	0.313
	8	62.040	9.670	1.241
	9	194.640	24.250	2.211
	10	289.680	54.910	3.457

Table 5.5: Results for BMTC Problems (5t)

# of Toilet	# of package	CGP	CMBP	C_CCALC
6	2	0.010	0.010	0.008
	3	0.010	0.070	0.016
	4	0.040	0.300	0.024
	5	0.060	1.160	0.043
	6	0.100	3.290	0.057
	7	211.720	9.060	0.121
	8	1015.160	20.710	2.377
	9	3051.990	50.610	9.829
	10	—	111.830	39.913

Table 5.6: Results for BMTC Problems (6t)

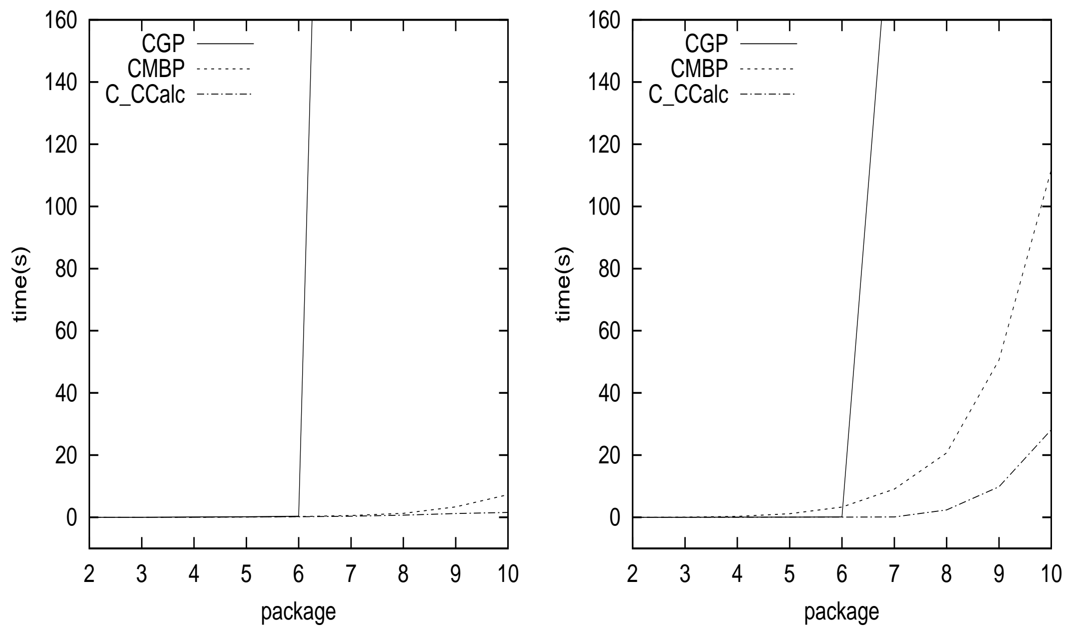

Figure 5.2: A Performance Comparison of the BMTC Problems: 3 toilets(left), 6 toilets(right)

5.2 Summary Remark

In summary, C_CCALC appears to be a promising approach to conformant planning in terms of expressivity and performance. Overall, CGP is least able to deal with large problems, even in the case that parallel actions are allowed. CMBP has no ability to implement concurrent actions, which potentially forms an obstacle to improve its behavior when the size of problem gets large, even though CMBP performs well in all the cases we have tested. C_CCALC performs significantly better than CGP and also in most of the cases outperforms CMBP. As the sizes of the problems increase, the advantage of C_CCALC is more evident.

However, C_CCALC appears to suffer when it cannot take advantage of concurrency. An example is that if there are 10 packages but only one toilet, so that no concurrent actions can be executed, the performance of C_CCALC deteriorates. The same kind of problem is also reported in CGP system [SW98] which also has a (limited) form of concurrent actions.

Chapter 6

Conclusions and Future Work

In this thesis, we introduce a new approach for conformant planning. Beginning with the high-level action description language $\mathcal{C}$, we extend the associated concepts and definitions from classical planning to deterministic conformant plan to provide a theoretical basis for our approach. We illustrate the approach by formalizing a standard conformant planning problem "bomb in the toilet" and we solve the problem involving up to 10 packages and up to 6 toilets.

We extend the system CCALC to C_CCALC to solve the conformant planning problem, and we provide a preliminary experimental evaluation of our approach relative to other conformant planners. We conclude that even though C_CCALC is a naive approach, in many cases it outperforms the other planners and shows strong potential to perform better. Our experiments suggest that the expressivity of concurrency of the action language $\mathcal{C}$ contributes to the overall perform of C_CCALC.

The research presented in this paper has the following deficiencies, and it may

be worthwhile to do some further work.

- We have investigated the way to conformant plan under the condition of not fully specified initial states, but we did not further the research to the case that action is nondeterministic. In particular, we did not implement the case that not only the position of bomb in package is unknown, but also the status of toilets (clogged or not clogged) after dunking is unknown.

- The paper [CR00] describes some other conformant domain problems that we did not attempt, such as "Ring of Rooms" and "Square and Cube". Further experiments would give us a better understanding of effectiveness of our approach.

- We obtain pretty good results in the problems presented, but we don't know how large problem size is the maximum for us to handle. Therefore, it would be good to run some larger examples of problems we have considered in the paper to see how quickly we hit the wall.

- Improve the format of plan file. Minimize the syntax differences between the classial planning files and deterministic conformant planning files.

- Implement front-end to compute set of possible initial possible states directly from the initial state descriptions instead of saying how many there are and describing each completely.

- In the current implementation, plan length is given to the system, and we report how long it takes to find a plan of that length. We should investigate how hard it is to determine that there is no shorter plan.

- Implement version of planner that searches for plan of minimal length, rather that for plan of given (fixed) length. (CCalc already does this for classical planning problems.)

- It would be nice to try to extend this approach of formalizing conformant planning as satisfiability to conditional planning, in which the actions to be performed at each step during execution of the plan can depend on the previous execuation history.

Appendix A

CC.h

```
% File: 'myCC.h' -- for use with C in conformant mode.
% The macro symbol 'maxtime' must be declared before
% including this file
% The macro symbol 'maxworld' must be declared before
% including this file too.

%% Operators for the language {\cal C}.
:- op(1010,fx,caused).
:- op(1010,xfx,causes).
:- op(1010,xf,may).
:- op(1010,yfx,cause).
:- op(1020,xfx,if).
:- op(1030,xfx,after).
:- op(1010,fx,inertial).
:- op(1010,fx,exogenous).
:- op(1010,fx,nonexecutable).
:- op(1010,fx,default).
:- op(1010,fx,always).
:- op(1010,fx,never).
:- op(1010,fx,constant).


%% Macros translating {\cal C} propositions into causal laws.  An
%% alternative to the translation given below for static laws would be
```

```
%%        caused #1 if #2
%%                -> (h(#2,T) => h(#1,T)).
%% If every fluent has its value at time zero determined exogenously,
%% the two translations are equivalent.
:- macros
        caused #1 if #2 after #3
                -> (next(_N,_T) && h(n(#3,_W1),_N) && h(n(#2,_W1),_T)
=> h(n(#1,_W1),_T)) ;
        caused #1 if #2
                -> ( (h(n(#2,_W1),0) ->> h(n(#1,_W1),0))
                   ; (_T>0 && h(n(#2,_W1),_T) => h(n(#1,_W1),_T)) ) ;

        caused #1 after #3
                -> caused #1 if true after #3 ;
        caused #1
                -> caused #1 if true ;

        #1 causes #2 if #3
                -> caused #2 if true after #3 && o(#1) ;
        #1 causes #2
                -> caused #2 if true after o(#1) ;

        inertial #1,#2 if #3
                -> (inertial #1 if #3; inertial #2 if #3) ;
        inertial #1 if #2
                -> caused #1 if #1 after #1 && #2;

        inertial #1,#2
                -> (inertial #1; inertial #2) ;
        inertial #1
                -> caused #1 if #1 after #1 ;

        exogenous #1,#2
                -> (exogenous #1; exogenous #2) ;

        exogenous #1
                -> (caused #1 if #1; caused -(#1) if -(#1)) ;

        never #1
                -> caused false if #1 ;
```

```
always #1
        -> caused false if -(#1) ;

nonexecutable #1 if #2
        -> caused false after o(#1) && #2 ;
nonexecutable #1
        -> caused false after o(#1) ;

default #1 if #2
        -> caused #1 if #1 && #2 ;
default #1
        -> caused #1 if #1 ;

#1 may cause #2 if #3
        -> caused #2 if #2 after #3 && o(#1) ;
#1 may cause #2
        -> caused #2 if #2 after o(#1) ;

constant #1
        -> ( caused false if #1 after -(#1)
           ; caused false if -(#1) after #1 );

n(-(#1),#2)
        -> -n(#1,#2) ;
n((#1 && #2),#3)
        -> n(#1,#3) && n(#2,#3) ;
n((#1 & #2),#3)
        -> n(#1,#3) & n(#2,#3) ;
h((#1 , #2),#3)
        -> n(#1,#3) , n(#2,#3) ;
n((#1 ++ #2),#3)
        -> n(#1,#3) ++ n(#2,#3) ;
n((#1 ->> #2),#3)
        -> n(#1,#3) ->> n(#2,#3) ;
n((#1 <-> #2),#3)
        -> n(#1,#3) <-> n(#2,#3) ;
n((/\ #1 : #2),#3)
        -> /\ #1 : n(#2,#3) ;
n((\/ #1 : #2),#3)
        -> \/ #1 : n(#2,#3) ;
```

```
h(-(#1),#2)
        -> -h(#1,#2) ;
h((#1 && #2),#3)
      -> h(#1,#3) && h(#2,#3) ;
h((#1 & #2),#3)
        -> h(#1,#3) & h(#2,#3) ;
h((#1 , #2),#3)
        -> h(#1,#3) , h(#2,#3) ;
h((#1 ++ #2),#3)
        -> h(#1,#3) ++ h(#2,#3) ;
h((#1 ->> #2),#3)
        -> h(#1,#3) ->> h(#2,#3) ;
h((#1 <-> #2),#3)
        -> h(#1,#3) <-> h(#2,#3) ;
h((/\ #1 : #2),#3)
        -> /\ #1 : h(#2,#3) ;
h((\/ #1 : #2),#3)
        -> \/ #1 : h(#2,#3) ;

o(-(#1),#2)
        -> -o(#1,#2) ;
o((#1 && #2),#3)
        -> o(#1,#3)  && o(#2,#3) ;
o((#1  & #2),#3)
        -> o(#1,#3)   & o(#2,#3) ;
o((#1  , #2),#3)
        -> o(#1,#3)   , o(#2,#3) ;
o((#1 ++ #2),#3)
        -> o(#1,#3)  ++ o(#2,#3) ;
o((#1 ->> #2),#3)
        -> o(#1,#3) ->> o(#2,#3) ;
o((#1 <-> #2),#3)
        -> o(#1,#3) <-> o(#2,#3) ;
o((/\ #1 : #2),#3)
        -> /\ #1 : o(#2,#3) ;
o((\/ #1 : #2),#3)
        -> \/ #1 : o(#2,#3) ;
```

```
        h(n(o(#1),_W1),#2)
                -> o(#1,#2) ;

h(n(#1,_W1),#2)
                -> #1 where test(#1) ;
o(#1,#2)
                -> #1 where test(#1).



%% A few auxilliary macros, some with Prolog 'where' clauses.
:- macros
#1 : #2 -> h(#2,#1)
        where integer(#1) ;
#1 - #2 -> #3
where integer(#1), integer(#2), #3 is #1 - #2 ;
#1 + #2 -> #3
where integer(#1), integer(#2), #3 is #1 + #2 .

%% Standard sort declarations and associated causal laws.
:- sorts
 action;
 fluent
   >> ( ( inertialFalseFluent;
                 inertialTrueFluent    ) >> inertialFluent ;
        ( defaultFalseFluent;
                 defaultTrueFluent     ) >> exogenousFluent
      );
 time >> step;
 world.

:- variables
 _W1, _W2   :: world;
 _N,_N1,_N2 :: step;
 _A,_A1,_A2 :: action;
 _T,_T1,_T2 :: time;
 _F,_F1,_F2 :: fluent;
 _DF :: defaultFalseFluent;
 _DT :: defaultTrueFluent;
```

```
 _IF :: inertialFalseFluent;
 _IT :: inertialTrueFluent.

:- constants
 1..maxworld   ::world;
 0..maxtime   :: time;
 0..maxtime-1 :: step;
 o(action,step), h(n(fluent,world),time) :: atomicFormula.

h(n(_F,_W1),0)  =>  h(n(_F,_W1),0).
-h(n(_F,_W1),0) => -h(n(_F,_W1),0).

next(_N,_T) &&  h(n(_IT,_W1),_N) &&  h(n(_IT,_W1),_T)
=>  h(n(_IT,_W1),_T).
next(_N,_T) && -h(n(_IF,_W1),_N) && -h(n(_IF,_W1),_T)
=> -h(n(_IF,_W1),_T).

_T>0 &&  h(n(_DT,_W1),_T) =>  h(n(_DT,_W1),_T).
_T>0 && -h(n(_DF,_W1),_T) => -h(n(_DF,_W1),_T).

o(_A,_N)  =>  o(_A,_N).
-o(_A,_N) => -o(_A,_N).
```

Bibliography

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Journal of Artificial Intelligence*, 90(1-2):281–300, 1997.

[BS97] R. Bayardo and R. Schrag. Using csp look-back techniques to solve real-world sat instance. *Proc. of the Nat'l Conf. on Artificial Intelligence*, pages 203–208, 1997.

[CR00] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *J. of Artificial Intelligence Research*, 13:305–338, 2000.

[FG00] P. Ferraris and E. Giunchiglia. Planning as satisfiability in nondeterministic domains. In *Proceeding of Seventeenth National Conference on Artificial Intelligence (AAAI'2000)*, pages 754–760, 2000.

[FN71] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 189–208, 1971.

[GB96] R. Goldman and M. Boddy. Expressive planning and explicit knowledge. *Proceedings of the 3rd International Conference on Artificial Intelligence Planning System (AIPS-96)*, pages 279–288, 1996.

[GL98a] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transaction on AI*, 3(16), 1998.

[GL98b] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation. In *Proc. AAAI-98*, pages 623–630, 1998.

[Gre69] C. Green. Application of theorem proving to problem solving. *Proceedings of IJCAI-69*, pages 219–239, 1969.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of the 10th European Conference on Artificial Intelligence (ECAI 92)*, pages 359–363, 1992.

[KS96] H Kautz and B Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press / MIT Press, August 4–8 1996.

[McC59] J. McCarthy. Program with common sense. *Proc. of the Teddington Conference on the Mechanization of Thought Process*, pages 75–91, 1959.

[McC99] Norman McCain. Using the causal calculator with the $\mathcal{C}$ input language (draft). Causal Calculator Manual, 1999.

[McD87] D. McDermott. A critique of pure reason. *Computational Intelligence*, pages 151–160, 1987.

[MT97] N. McCain and H. Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[SW98] D. Smith and D Weld. Conformant graphplan. In *In Proc. 15th National Conference on AI*, pages 889–896, 1998.

[Tur01] H. Turner. Polynomial-length planning spans the polynomial hierarchy. Unpublished manuscript, 2001.

[Wel99] D. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.

[Zha97] Hantao Zhang. Sato: An efficient propositional prover. *Proc. of International Conference on Automated Deduction (CADE-97)*, pages 272–275, 1997.