

Conformant Planning as QBF Satisfiability

A THESIS

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

BY

Krishna Kishore Kotnana

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

July 2003

UNIVERSITY OF MINNESOTA

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

Krishna Kishore Kotnana

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 Advisor

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Acknowledgements

I am deeply grateful to my advisor, Dr. Hudson Turner, for all the guidance, support and encouragement he provided. He played the principal role in making this work possible for me.

I am also thankful to Dr. Joe Gallian and Dr. Doug Dunham for their patient examination of the thesis and their valuable feedback.

I am grateful to the Department of Computer Science at the University of Minnesota Duluth, for all the support it provided.

I am grateful to all my friends who were, and continue to be, a constant source of encouragement.

Finally, I would like to express my gratitude to my parents and to my sisters, who made it possible for me to come this far.

Abstract

In *conformant planning*, we are given a description of an action domain—defining the possible states of the world and the possible effects of actions on those states—along with a description of the initial state and a description of the goal. The action domain may be nondeterministic, so that we cannot always know what the outcome of a given action in a given state will be. Similarly, the description of the initial state may be incomplete, so that we do not know everything about the initial state. A *plan* is a finite sequence of sets of actions, with each set of actions to be executed (concurrently) in order of appearance in the sequence. A plan is *valid* if it has the following two properties. (1) It is *executable*. That is, there is at least one possible initial state, and for each proper prefix of the plan, it is possible to execute the next set of actions in any of the states that can be reached by executing the prefix (starting in any possible initial state). (2) It is *sufficient*. That is, any state that can be reached by executing the full plan (starting in any possible initial state) satisfies the goal description. The objective is to find a valid plan.

In this thesis, we use a Quantified Boolean Formula (QBF) to represent a conformant planning problem, which can then be solved by a standard QBF solver. More specifically, we construct a formula in propositional logic that encodes all possible histories of a given length k in the action domain, i.e., all sequences of $k + 1$ states interleaved with k (sets of) actions that, according to the action description, can take the world from one state to the next. Using this formula, along with formulas that encode the initial state description and the goal description, we construct a QBF that encodes all valid plans of length k . That is, such a plan exists if and only if the QBF is satisfiable. A QBF solver can be used to decide this question of satisfiability, and, furthermore, if there is a valid plan of length k , it is easily extracted from the output of the QBF solver.

This conformant planning method is notable for its generality and mathematical elegance.

Contents

1	Introduction	1
1.1	Classical Planning	2
1.2	Conformant Planning	4
1.3	Conditional Planning	7
1.4	Contribution of the Thesis	7
1.5	Outline of the Thesis	9
2	Basic Framework	12
2.1	Action Domains as Transition Systems	12
2.1.1	Plan Validity in Terms of Transition Systems	14
2.2	Propositional Logic	16
2.2.1	Syntax	17
2.2.2	Semantics	17
2.2.3	Useful Terminology, Notation Conventions and Facts	18
2.3	Representing Transition Systems Using Propositional Logic	20
2.3.1	Characterization of Plan Validity in Propositional Logic	23
2.4	Classical Planning as Propositional Satisfiability	25
2.4.1	Using CCALC for Classical Planning	26
2.5	Second-Order Propositional Logic	27
2.5.1	Syntax	27
2.5.2	Semantics	28

2.5.3	Useful Terminology, Notation Conventions and Facts	28
2.6	Conformant Planning in Second-Order Propositional Logic	32
2.7	Formulas Produced by CCALC	34
3	Approach	37
3.1	From CCALC Formulas to a QBF in Prenex Normal Form	38
3.2	A Standard Algorithm	41
3.2.1	Problems with this Method	43
3.3	Further Improvements	44
3.3.1	METHOD1: Clausification with New Atoms	44
3.3.2	METHOD2: Exploiting the Formula Structure	45
3.3.3	METHOD3: Fine Tuning	47
3.3.4	A Natural Way of Quantification	48
4	Experiments	50
5	Other Conformant Planning Methods	64
6	Conclusions	68
6.1	Summary	68
6.2	Future work	69
7	Appendix	70

List of Figures

2.1	Transition system for classical version of Bomb in Toilet problem	13
2.2	Transition system for standard Bomb in Toilet with 2 packages	13
2.3	Transition system for Bomb in Toilet with nondeterministic clogging and two packages	14

List of Tables

4.1	BT Family - Original Encoding - 1 to 8 toilets	54
4.2	BT Family - Original Encoding - 9 to 16 toilets	55
4.3	BT Family - Original Encoding - 17 to 24 toilets	56
4.4	BT Family - Original Encoding - 25 to 30 toilets	57
4.5	BT Family - Natural Encoding - 1 to 8 toilets	58
4.6	BT Family - Natural Encoding - 9 to 16 toilets	59
4.7	BT Family - Natural Encoding - 17 to 24 toilets	60
4.8	BT Family - Natural Encoding - 25 to 32 toilets	61
4.9	BTC Family - Natural Encoding - 1 to 8 toilets	62
4.10	BTC Family - Natural Encoding - 9 to 14 toilets	62
4.11	BTNC Family - Natural Encoding - 1 to 8 toilets	63
4.12	BTNC Family - Natural Encoding - 9 to 13 toilets	63

Chapter 1

Introduction

Planning is one of the oldest problems in the field of Artificial Intelligence. Problems in planning have been studied in various forms and varied techniques have been used to approach planning problems. A planning problem is usually defined by three basic components, namely,

- a description of the action domain—that is, the possible states of the world and how actions may affect them,
- a description of the initial state,
- a description of the goal.

Essentially, the problem is to find a plan that will always succeed in achieving the goal (that is, reaching a state that satisfies the goal description) starting in any of the possible initial states (that is, when the initial state is a state that satisfies the initial state description).

In this thesis, we will be concerned with plans that can be expressed as a finite sequence of sets of actions. Each element of the sequence is a set of actions to be performed concurrently, with each set of actions to be performed in order of appearance in the sequence. Roughly speaking, we call such a plan “valid” if it is guaranteed to be executable and, moreover, every possible execution is guaranteed to achieve the goal. We’ll call these two properties of a valid plan “executability” and “sufficiency.” A planning problem is solved

by either finding a valid plan or determining that none exists. In this thesis, we focus on the special case of finding a valid plan of a given length k .

There are two sources of uncertainty in planning problems. First, it may be that the initial state description does not completely determine the identity of the initial state. Second, the action domain itself may be nondeterministic. That is, there may be more than one state that can result from concurrently performing a given set of actions in a given state. In this thesis, we will be concerned with planning in the presence of both sources of uncertainty.

1.1 Classical Planning

In classical planning, the initial state is completely known and the action domain is deterministic. Consider the following example.

Let there be a toilet and a package that contains a bomb. The bomb can be disarmed by dunking the package in the toilet. Suppose that the bomb is initially armed and our goal is to disarm the bomb.

A state in an action domain is described by a set of symbols called fluents. Intuitively, each fluent stands for a property of the world that may change from state to state.

There is only one fluent in this example domain:

- *armed*, to say that the bomb is armed.

We represent a state by a subset of the set of fluents: intuitively, those fluents that stand for properties that are “true” in that state. In general, we do not assume that every subset of the set of fluents is a state; intuitively, some such subsets may not correspond to states of the world.

The states in this action domain are:

1. $\{\}$,
2. $\{armed\}$.

The only state that satisfies the initial state condition is $\{armed\}$. Since in this case there is exactly one state that satisfies the initial conditions, we say that knowledge about the initial state is complete.

Each action in an action domain is represented by a symbol. For convenience, we use the word action to refer not only to the action itself but also to the symbol that represents it. (We assume that the set of actions is disjoint from the set of fluents.)

The only action in this domain is:

1. $dunk$.

In general, we are interested in concurrent execution of actions. For convenience, we introduce the term *event* to stand for a set of actions. So each step in a plan corresponds to the execution of an event. The number of such steps in a plan is the length of the plan.

In this example the events are:

1. $\{\}$,
2. $\{dunk\}$.

The event $\{\}$ can be executed from either of the states and will result in the same state. The event $\{dunk\}$ can be executed in either of the states and in each case results in state $\{\}$.

Such action domains, where executing an event in any state results in exactly one state, are called deterministic action domains, and actions in such domains are referred to as deterministic actions. (Note that determinism does not guarantee that every event is executable in every state.)

A plan is a finite sequence of events. Such a plan is executable if, starting from the initial state, the execution of each event in the sequence results in a state in which the next event is executable. This simple characterization of executability depends on the fact that this planning problem is classical, so that there is exactly one possible initial state, and the execution of each event in the plan leads to a unique resulting state. We will see that in conformant planning the executability condition is more complicated. If such an execution

of a plan results in a goal state (i.e., a state that satisfies the goal) then the plan is sufficient. Again, this characterization of sufficiency depends on the fact that the planning problem is classical. More generally, a plan is sufficient if *every* possible execution of the plan from an initial state results in a goal state. A plan that is both executable and sufficient is valid.

In the current example, the sequence $\langle \{dunk\} \rangle$ is a valid plan of length 1.

In the next section we will look at elaborations of this example—involving uncertainty about the initial state and about the effects of actions—in which the planning problem is more complicated.

1.2 Conformant Planning

Let us modify the example discussed in the previous section by adding another package. There is still only one bomb, and we assume that it is contained in one of the packages but we do not know which one. Only one package can be dunked in the toilet at a time. Initially, the bomb is armed. Other conditions remain the same.

Let us call the packages $p1$ and $p2$, and the fluents for this action domain are:

- $armed$,
- $in(p1)$, to say that the bomb is in package $p1$,
- $in(p2)$, to say that the bomb is in package $p2$.

Since exactly one of $in(p1)$ and $in(p2)$ can be “true” in a given state, we have only 2^2 states.

Notice that now we have multiple possible initial states. They are:

1. $\{in(p1), armed\}$,
2. $\{in(p2), armed\}$.

The actions are:

1. $dunk(p1)$,

2. $dunk(p2)$.

So the events are:

1. $\{\}$,

2. $\{dunk(p1)\}$,

3. $\{dunk(p2)\}$,

4. $\{dunk(p1), dunk(p2)\}$.

A valid plan in this case is $\langle\{dunk(p1)\}, \{dunk(p2)\}\rangle$. Notice that it is executable from either of the possible initial states and each such execution results in a goal state.

Planning in such conditions, where the initial state may be incompletely specified but actions are deterministic, is sometimes called *deterministic* conformant planning. The example described here is a standard one, called the “bomb in the toilet problem”, introduced in [11].

Now let us make the problem a little more interesting. Suppose that dunking a package clogs the toilet, and that a package cannot be dunked in a clogged toilet. Let us add a flush action that unclogs the toilet. We will stipulate that it is not possible to flush and dunk a package at the same time. Assume that initially the toilet is not clogged. Otherwise what we know about the initial state remains the same, and as usual the goal is to disarm the bomb.

So now there is a new fluent (*clogged*), and a new action (*flush*). There are 2^3 states. The possible initial states are:

1. $\{in(p1), armed\}$,

2. $\{in(p2), armed\}$.

A valid plan in this case would be $\langle \{dunk(p1)\}, \{flush\}, \{dunk(p2)\} \rangle$.

This example falls in the same category as the last one (deterministic conformant), but it prepares us for a further elaboration that illustrates conformant planning in its full generality, with uncertainty about the effects of actions in addition to uncertainty about the initial state.

Suppose that dunking a package does not always clog the toilet. With this modification, the action domain becomes nondeterministic. For example, performing the set of actions $\{dunk(p1)\}$ in the state $\{in(p1), armed\}$ can result in either of the states $\{in(p1), clogged\}, \{in(p1)\}$.

With a little bit of thinking, we can see that the solutions to this version of the problem are the same as in the previous version, but the *reasons* are a bit more complicated. Consider, for instance, the plan $\langle \{dunk(p1)\}, \{dunk(p2)\} \rangle$. In a very real sense, this plan can be executed in either of the possible initial states. For instance, starting in state $\{in(p1), armed\}$, doing $\{dunk(p1)\}$ can lead to state $\{in(p1)\}$. And in this state, one can execute $\{dunk(p2)\}$. Similar observations hold for the other possible initial state state $\{in(p2), armed\}$. Moreover, it is surely the case that whenever the plan has been (“fully”) executed starting from either of the two possible initial states, the result will be a state that satisfies the goal. So why is $\langle \{dunk(p1)\}, \{dunk(p2)\} \rangle$ not a valid plan? The short answer is: while it is sufficient, it is not (guaranteed to be) executable. It fails the executability condition because, for example, executing $\{dunk(p1)\}$ in state $\{in(p1), armed\}$ can result in the state $\{in(p1), clogged\}$, at which point it is not possible to execute $\{dunk(p2)\}$.

What we see is that executability requires that the result of executing any proper prefix of the plan, starting in any possible initial state, must be a state in which the next event in the plan can also be executed. (Also, of course, there must be at least one possible initial state.)

As we have seen, conformant planning allows uncertainty about the initial state and about the effects of actions, but it considers only plans that can be represented as a sequence of events. In the next section, we briefly consider the significance of this limitation.

1.3 Conditional Planning

Suppose that in the “bomb in toilet problem with non-deterministic clogging” we cannot flush, but we can—during plan execution—test the packages to determine which one has the bomb. Notice that in this case there is no conformant plan that works! In such cases, a plan may need to have a more complex structure in which the action(s) to be performed at each step in the execution of the plan can depend on what is observed about the current state and/or the previous execution history. For instance, in the example at hand, one could first observe which package has the bomb and then dunk that package. This is a simple plan, but it is conditioned on observations that are to be made during plan execution.

This is an example of conditional planning. Clearly conditional planning problems need more involved treatment than conformant and classical planning problems. In general, a conditional plan must map the observable portion of each possible partial execution of the plan to the next action(s) to be executed. Such plans are clearly rather complex, and there are very many of them, and they may grow quite large. In fact, plans involving k or fewer execution steps may have size exponential in k .

In the remainder of the thesis, our primary interest will be conformant planning, but we will have several occasions to consider the special case of classical planning.

1.4 Contribution of the Thesis

This thesis contributes to the development of a general, mathematically elegant approach to conformant planning, based on the use of quantified Boolean formulas (QBF).

Over the last decade, *satisfiability planning* [9] has been a highly influential approach to classical planning. In satisfiability planning, the action domain is described by a formula of classical propositional logic. More precisely, all possible histories of a given length k are described by such a formula, in the sense that the models of the formula correspond to the possible histories of length k . (This will be explained more fully in the thesis.) In addition, there is a formula describing the initial state and a formula describing the goal. A valid plan

is found by submitting these formulas to a standard satisfiability solver—a general-purpose tool that decides whether the formulas are satisfiable (i.e. can be made true), and, if so, returns a model. A plan is then (easily) extracted from the model. The correctness of this approach depends on both assumptions of classical planning: the action domain must be deterministic and the initial state must be completely known.

The success of this approach is attributable not only to its mathematical elegance but also to the rapid improvement of satisfiability solvers, which are themselves an object of intense study.

This thesis works out some details that can allow us to nicely extend the satisfiability planning approach to the more complex and computationally demanding problem of conformant planning. We begin with the same fundamental building blocks: formulas describing the action domain, what is known about the initial state, and the goal. Using these three formulas, we construct a QBF that encodes all valid conformant plans of a given length. Finally we submit this QBF to a standard QBF solver to check for satisfiability. From the output of the solver we can (easily) either extract a valid plan of the given length or determine that there is none.

This approach to conformant planning is not new. It can be traced at least back to work by Rintanen published in 1999 [12], and was given the more general formulation used here by Turner [14]. This thesis works out some of the practical details needed to implement Turner’s formulation.

Some additional complications addressed in this thesis are due to the desire to adapt this approach for use in conjunction with the Causal Calculator (CCALC): an implemented system for reasoning, about actions, publically available at the following url: www.cs.utexas.edu/users/tag/cc/. The primary input to CCALC is an action domain description expressed in a “high-level” logic-based action language. The details of this input language are not essential to this thesis. What is interesting for our purposes is that CCALC compiles this high-level action description into a formula of classical propositional logic, and then reasons about this formula by means of calls to a standard

satisfiability solver. In particular then, CCALC can perform satisfiability planning using an unusually expressive action description language. The work described in this thesis lays the foundation for extending CCALC to perform conformant planning.

The preliminary experiments reported in this thesis do not look very encouraging from the point of view of computation times and the ability to solve relatively large problems; the results are not competitive with current state-of-the-art conformant planning systems. Nonetheless, there are at least two reasons that the work described in the thesis may prove interesting in the longer run. First, this approach is more general than others. For instance, it can be applied to the expressive action language used by CCALC, rather than the relatively restricted action languages used by existing conformant planning systems. Second, this approach can benefit from improvements in general-purpose QBF solvers, which are themselves currently an object of study for a number of computer scientists.

1.5 Outline of the Thesis

In Chapter 2, we develop the basic framework needed in order to talk about conformant planning problems. In Section 2.1, we give an account of how “transition systems” can be used for a simple and convenient representation of action domains. Here we show how states and effects of actions in an action domain can be represented using transition systems and then give a characterization of plan validity in terms of the transition system, for both classical planning and conformant planning. Although transition systems provide a convenient way to talk precisely about action domains, for purposes of solving planning problems as satisfiability, we are interested in logic-based representations. (Because we want to be able to write a formula such that the problem of satisfiability of this formula is the same as, or has a clear correspondence to, the planning problem we want to solve.) As a first step towards this end, in Section 2.2, we give an account of the basic syntax and semantics of propositional logic along with some notational conveniences. Here we also present some facts that are useful for the representation of action domains we use. Then, based on

the ground work done, we show how information about action domains can be expressed in propositional logic. In particular we will form propositional formulas that encode the same kind of knowledge expressed by our transition systems representation. Here we also present formulas to represent the initial states and the goal. Using these formulas we give an equivalent (to the transition systems version) characterization of validity of classical and conformant plans. Then in Section 2.4 we show how classical planning problems can be solved as problems of propositional satisfiability using the propositional logic characterization of a valid classical plan. We also observe that propositional logic falls short in terms of expressability when it comes to conformant planning and that we need a more expressive logic. In Section 2.5 we present the basic syntax and semantics of the more expressive logic we will use—second-order propositional logic. (Second order propositional logic is not an entirely different logic. It can be considered an extension of propositional logic and we take this standpoint while defining the syntax and semantics.) Here again we introduce some notational conveniences and facts that prove useful in later sections. Then, in Section 2.6, we show how the propositional formulas used in classical planning can be used to build a QBF that encodes all valid conformant plans. In our approach we obtain these propositional formulas from the planning tool CCALC and then produce the QBF needed, in a form that is acceptable by a standard QBF solver. We end Chapter 2 with a description of how the planning tool CCALC works to produce the propositional logic formulas that we need and point out that these formulas are actually slightly different, and that as a result there are complications in the construction of the QBF.

In Chapter 3, we take up the task of building a QBF that encodes all valid conformant plans from the information we obtain from CCALC. Standard QBF solvers accept QBF's in conjunctive normal form, a special case of prenex normal form. As a first step in this direction, we show an equivalent form of the QBF we need that is in prenex normal form in Section 3.1. Then our goal is to find an equivalent to the propositional part of the prenex QBF in conjunctive normal form. In Section 3.2 we give an account of a standard method for finding equivalents in conjunctive normal form and give reasons why such methods are

not acceptable for us. Then, in Section 3.3, we observe that we do not absolutely require an equivalent formula in conjunctive normal form but instead can use a formula with a slightly weaker property. This change of stance does introduce some complications to the QBF formulation that we use. On the other hand, it produces a smaller and more manageable formulas. The rest of the section works out the details that are involved in dealing with these complications.

In Chapter 4, we present the results of experiments conducted on our method for conformant planning. Our experiments are on the three families of bomb and toilet problems already discussed. We observe that the results are not competitive with other state-of-the-art planners. We explain in brief some factors that could affect the performance of our planner.

In Chapter 5 we discuss in brief some of the recent conformant planning methods. We conclude with comments on our approach in relation to the other conformant planners.

Chapter 2

Basic Framework

2.1 Action Domains as Transition Systems

A *transition system* is a directed graph whose nodes represent states and whose edges represent transitions between the states. Transition systems are convenient as a precise, general representation for action domains.

In our transition systems, the nodes are the states in the action domain, and there is a directed edge from state a to state b if and only if there is an event (a set of actions executed concurrently) that can be executed in state a and such an execution can possibly lead to state b . The edges are labelled with the events associated with them. We denote an edge from a to b with event e associated with it, by the triple $\langle a, e, b \rangle$.

Let us consider an action domain with set F of fluents and a set A of actions. (F and A are assumed to be disjoint.) Let $S \subseteq 2^F$ be the set of states. The set of edges is as follows:

$$E = \{ \langle a, e, b \rangle \in S \times 2^A \times S \mid e \text{ is executable in } a \text{ and can result in } b \}.$$

Given the above representation, describing an action domain is equivalent to defining the node and edge sets of the associated transition system. Note that in a transition system associated with a non-deterministic domain there can be multiple edges labelled with the same event all originating from the same node.

For fairly simple action domains, we can conveniently draw the transition system associated with it. For example, the transition system associated with the action domain in our classical planning version of the bomb in the toilet problem (with only one package) is as shown in figure 2.1.

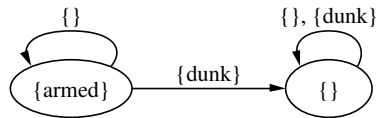


Figure 2.1: Transition system for classical version of Bomb in Toilet problem

The transition system for the bomb in the toilet problem with two packages is shown in figure 2.2. Although the planning problem is conformant, the action domain is deterministic (as is clear from the transition system). The uncertainty in this particular problem is in the initial state description (which is part of the planning problem but not a part of the action domain).

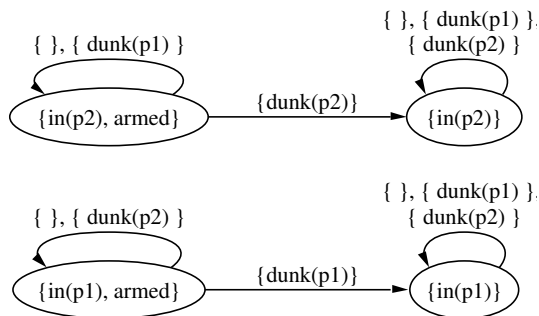


Figure 2.2: Transition system for standard Bomb in Toilet with 2 packages

There is another interesting thing in this transition system. There are two components (disconnected subgraphs) in the transition system. Within a component either all the states contain $in(p1)$ or none does, and similarly for $in(p2)$. Intuitively, this means that actions in the domain have no effect on these fluents. We call such fluents *rigid*. (We will later find it useful to distinguish between rigid and non-rigid fluents.)

For sake of completeness, let us also look at an example of a transition system for a non-deterministic domain. Figure 2.3 shows the transition system for the bomb in the

toilet problem with nondeterministic clogging. Notice that there are multiple edges with the same label from the same node. For example, from the state $\{in(p1), armed\}$, there are two edges with the label $\{dunk(p1)\}$.

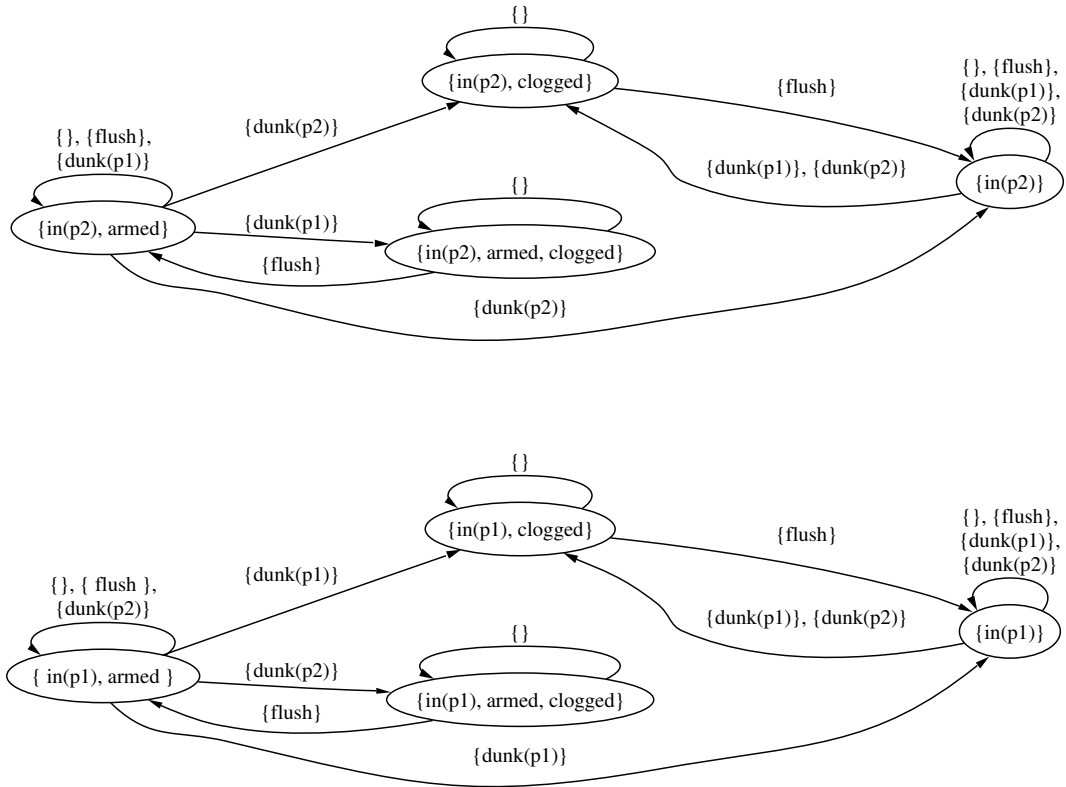


Figure 2.3: Transition system for Bomb in Toilet with nondeterministic clogging and two packages

2.1.1 Plan Validity in Terms of Transition Systems

Given the transition system for an action domain, a planning problem description can be completed by specifying the sets of initial and goal states. Thus, we can identify a planning problem with the triple $\langle I, D, G \rangle$, where I is the set of possible initial states, D is the transition system associated with the action domain and G is the set of goal states.

A plan of length k is simply a sequence

$$\langle e_0, e_1, \dots, e_{k-1} \rangle$$

of events (subsets of A).

In a classical planning problem, there is only one possible initial state and, due to the determinism of the action domain, a plan can involve at most one execution path, or “possible history”. (There will be exactly one possible history if the plan is indeed executable starting in the initial state.) Thus, the valid plans correspond to the paths connecting the initial state to one of the goal states. That is, a valid plan of length k is simply a sequence $\langle e_0, e_1, \dots, e_{k-1} \rangle$ of events for which there are states s_0, \dots, s_k such that $\langle s_0, e_0, s_1, e_1, \dots, s_{k-1}, e_{k-1}, s_k \rangle$ is a path in the transition system D with $s_0 \in I$ and $s_k \in G$.

In deterministic conformant planning, the situation is more complicated, because there may be more than one possible initial state. (There must be at least one, otherwise no plan would be executable!) So even with a deterministic action domain there may be several possible histories associated with a given plan. Still, in this case, it is not difficult to characterize the valid plans. If there is at least one possible initial state, then a plan is valid if and only if it is valid for each of the classical planning problems obtained by restricting consideration to each of the possible initial states.

In the most general case, when the action domain may also be non-deterministic, executability is a bit harder to characterize.

A plan $E = \langle e_0, e_1, \dots, e_{k-1} \rangle$ for the problem $\langle I, D, G \rangle$, of length k is *valid* if:

1. $I \neq \{\}$,
2. for all $i \in \{0, \dots, k-1\}$, for all states s_0, \dots, s_i , if
 - (a) $s_0 \in I$ and
 - (b) $\langle s_0, e_0, \dots, s_{i-1}, e_{i-1}, s_i \rangle$ is a path in D ,

then there is a state s_{i+1} such that $\langle s_i, e_i, s_{i+1} \rangle$ is a transition in D ,

3. for all states s_0, \dots, s_k , if
- (a) $s_0 \in I$ and
 - (b) $\langle s_0, e_0, s_1, e_1, \dots, s_{k-1}, e_{k-1}, s_k \rangle$ is a path in D ,
- then $s_k \in G$.

Intuitively, condition (1) guarantees the existence of an initial state, condition (2) guarantees that, for each i ($0 \leq i < k$) the event e_i is executable from each of the possible states that could result after the execution of the (previous) events e_0, \dots, e_{i-1} in that order, and finally condition (3) guarantees that any state that results after execution of a valid plan satisfies the goal. Conditions (1) and (2) together guarantee executability and condition (3) guarantees sufficiency.

For example, given the planning problem with the transition system from figure 2.3, and with initial states $\{in(p1), armed\}, \{in(p2), armed\}$, where the goal is to disarm the bomb, the valid plan $\langle \{dunk(p1)\}, \{flush\}, \{dunk(p2)\} \rangle$ corresponds to four different possible histories.

While transition systems provide a simple and general representation for action domains, and make it reasonably straightforward to define validity for conformant plans, we will be primarily interested in a logic-based representation of action domains, planning problems and plan validity. As a first step, we will show how the same kind of knowledge represented by a transition system can instead be represented using propositional formulas. Let us now do some groundwork about the syntax and semantics of propositional logic.

2.2 Propositional Logic

We begin with a standard account of the syntax and semantics of classical propositional logic, and then describe additional terminology, notation conventions and useful facts.

2.2.1 Syntax

A *signature* is a set of symbols, each of which is called an *atom*. The set of logical symbols used in propositional logic is

$$\{\perp, \top, \vee, \wedge, \neg, \equiv, \supset, \cdot, \cdot, \cdot\}.$$

\top (top) and \perp (bottom) are zero-place connectives, \neg (negation) is a unary connective, and \vee (disjunction), \wedge (conjunction), \equiv (equivalence), \supset (implication) are binary connectives. We assume that the set of logical symbols and the signature are disjoint.

The set of *formulas* of propositional logic (with respect to a given signature) is defined recursively as follows:

- The zero-place connectives and the atoms are formulas.
- If F is a formula $\neg F$ is a formula.
- If F and G are formulas then $(F \odot G)$ is a formula where \odot is a binary connective.

2.2.2 Semantics

A *truth-valued* function is any function whose codomain is the set $\{\mathbf{t}, \mathbf{f}\}$, where \mathbf{t} and \mathbf{f} are called *truth values*. An *interpretation* (of a propositional signature) is a truth-valued function over the signature. If a is an atom, the truth value assigned to a by the interpretation I is of course denoted by $I(a)$.

An interpretation I assigns to each formula F over a signature a truth value F^I which is defined recursively as follows:

- If F is an atom, $F^I = I(F)$.
- $\perp^I = \mathbf{f}$ and $\top^I = \mathbf{t}$.
- $(\neg F)^I = \begin{cases} \mathbf{t} & \text{if } F^I = \mathbf{f} \\ \mathbf{f} & \text{otherwise.} \end{cases}$

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

An interpretation I is said to *satisfy* a formula F (written as $I \models F$) if and only if $F^I = \mathbf{t}$. We will refer to an interpretation that satisfies a formula as a *model* of the formula. Similarly, an interpretation satisfies a set of formulas if it satisfies each of the formulas in the set. We use the notation $I \models F$ to say that I satisfies the formula F . Similarly, if S is a set of formulas, we write $I \models S$ to denote that I satisfies each formula in S .

A formula F *entails* a formula G if the set of models of G is a subset of the set of models of F . That is for any interpretation I , whenever $F^I = \mathbf{t}$, then $G^I = \mathbf{t}$. We write $F \models G$ with the meaning that F entails G .

Two formulas F and G are equivalent if and only if $F \models G$ and $G \models F$. We use the notation $F \leftrightarrow G$ with the meaning that F is equivalent to G .

2.2.3 Useful Terminology, Notation Conventions and Facts

Often we abbreviate a formula $(F \odot G)$ by removing the outermost parentheses. Another convenient abbreviation is $F_1 \vee F_2 \vee \cdots \vee F_n$ which will stand for $(\cdots ((F_1 \vee F_2) \vee F_3) \cdots \vee F_n)$, for $n \geq 2$. We will use a similar abbreviation for conjunction.

The *negation* of a formula F is the formula $\neg F$. A *literal* is an atom or its negation.

A formula of the form $L_1 \vee L_2 \vee \cdots \vee L_n$, where $n \geq 1$ and L_1, \dots, L_n are literals, is called a *simple disjunction*. A simple disjunction is also called a *clause*. Similarly a

formula of the form $L_1 \wedge L_2 \wedge \cdots \wedge L_n$ is called a *simple conjunction*.

A formula is in *conjunctive normal form* if it is of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where $n \geq 1$ and C_1, \dots, C_n are simple disjunctions. Dually, a formula is in *disjunctive normal form* if it is of the form $D_1 \vee D_2 \vee \cdots \vee D_n$, where $n \geq 1$ and D_1, \dots, D_n are simple conjunctions.

As mentioned previously, a satisfiability solver takes as input a formula and checks if it is satisfiable, and, if so, returns a model of the formula. Conjunctive normal form is a standard form in which most satisfiability solvers accept their input. So, it becomes more or less essential to replace a propositional formula with its equivalent in conjunctive normal form.

Fact 1 *Every formula over a non-empty signature is equivalent to a formula in conjunctive normal form and to a formula in disjunctive normal form.*

In general, conversion of a formula into conjunctive normal form may cause an exponential increase in the size of the formula. Fortunately, it is possible to avoid this exponential cost, at the cost of extending the signature with additional atoms. Thus we have a strong interest in the next definition.

Let F be a formula over the signature σ . A formula G over a signature σ' , with $\sigma \subseteq \sigma'$, is a *conservative extension* of F if, for all interpretations I of σ , $I \models F$ if and only if there is an interpretation I' of σ' that extends I and satisfies G .

We will see that, for any formula F , we can efficiently compute a conservative extension G whose size is linear in F . Such a possibility is crucial to our approach. In fact, we will eventually introduce special purpose algorithms for this purpose, tailored to the formulas we use.

We will need to be able to, so to speak, transpose a formula from one signature to another. Recall that we will be interested in formulas that describe the transitions of a transition system. We will wish to use such formulas in turn to describe all paths of length k in that system, and we will do so, roughly speaking, by making k copies of the transition formula. Following is notation related to this need.

If σ_1 and σ_2 are two ordered sets of atoms of the same size, and $F(\sigma_1)$ is a formula, then $F(\sigma_2)$ is the formula obtained by replacing simultaneously each occurrence of an atom from σ_1 in $F(\sigma_1)$ by its corresponding atom in σ_2 . Similarly, if $F(\sigma_1, \sigma_2, \dots, \sigma_n)$ is a formula, then $F(\gamma_1, \gamma_2, \dots, \gamma_n)$ is the formula obtained by replacing simultaneously every occurrence of an atom from $\cup_i \sigma_i$ in $F(\sigma_1, \sigma_2, \dots, \sigma_n)$ with the corresponding atom from $\cup_i \gamma_i$. (Here we assume that the σ_i are pairwise disjoint.)

Finally, we wish to observe that it is indeed possible to represent any finite transition system by a formula. To this end, notice that a formula can also be seen as a truth-valued function over the set of all interpretations of its signature.

Fact 2 *For any set L of interpretations of a finite signature, there exists a formula F such that $L = \{I : F^I = \mathbf{t}\}$.*

Moreover, when convenient, we can identify an interpretation with the set of atoms it makes true. In this way, when the set of fluents is finite, it is always possible to represent the set of states by a formula—one whose models are (or correspond to) the states. Similarly, each event in a finite transition system can be represented by a formula (whose signature is the set of action symbols). Shortly we will extend this idea to allow representation of the set of transitions in a finite transition system.

2.3 Representing Transition Systems Using Propositional Logic

In this section, we will see how a transition system can be represented in propositional logic, using one formula to represent the set of states and another formula to represent the set of transitions. Recall that transitions are triples of the form $\langle a, e, b \rangle$, where a and b are states and e is an event.

From now on, we will assume that both the set of fluents and the set of actions are finite. As we saw in the previous section, under this assumption Fact 2 guarantees that the

set of states can be represented by a formula of propositional logic whose atoms are taken from the set of fluents. Similarly, any given event e can be represented by a formula whose atoms are taken from the set of actions.

In order to represent transitions though, we need to be able to encode two states: intuitively, an initial state (the state before an event is executed) and a resulting state (a state that may result from that execution). For this purpose, we will need two copies of the set of fluents. Or, more accurately, we will need two copies of the non-rigid fluents. (By the definition of rigid, the initial and resulting states of a transition cannot differ on the truth value of a rigid fluent, so we will not need two copies of the rigid fluents.)

Let R be the set of rigid fluents, let F be the set of non-rigid fluents, and let A be the set of actions. Let $state(F)$ be a propositional formula over $R \cup F$ that represents the set of states. That is, the interpretations of $R \cup F$ that satisfy $state(F)$ correspond exactly to the states of the transition system.

For example, in the standard bomb in the toilet domain with two packages, the set R is $\{in(p1), in(p2)\}$, the set F is $\{armed\}$ and the formula $in(p1) \equiv \neg in(p2)$ is a formula whose models are precisely the states in the domain. Intuitively, the formula expresses that in any state of the domain, there is exactly one package with the bomb.

Let F_0 and F_1 be “copies” of the set F obtained by subscripting each atom of F with 0 and 1 respectively. (We assume of course that F is disjoint from F_0 and F_1 . Similar disjointness assumptions will be left unstated from now on.) Then, in the obvious way, a state can be represented by a subset (or interpretation) of $R \cup F_0$ and another state can be represented by a subset (or interpretation) of $R \cup F_1$. And a subset (or interpretation) of $R \cup F_0 \cup F_1$ can be understood to represent a pair of states that agree on the truth value of each of the rigid fluents. Similarly then, a transition—which is a triple $\langle a, e, b \rangle$, where a and b are states that agree on rigid fluents, and e is an event—can be represented by a subset (or interpretation) of $R \cup F_0 \cup A \cup F_1$.

Hence, the set of transitions can be represented by a formula $tr(F_0, A, F_1)$ over the signature $R \cup F_0 \cup A \cup F_1$ whose models correspond to the edges of the transition system.

Note that this correspondence depends not only on Fact 2 but also on the fact that the fluents in R are rigid.

Before we consider an example, we introduce some notation that will prove convenient. If s is a subset (or interpretation) of $R \cup F$, then for any natural number n , let $s(n)$ denote the corresponding interpretation of $R \cup F_n$. Then, for any interpretation s of $R \cup F$ and any natural number n , s is a state if and only if $s(n) \models \text{state}(F_n)$. In a similar manner, we will eventually have use for “copies” of the set of actions A . So, for any event e , $e(n)$ will denote the corresponding interpretation of signature A_n (obtained by subscripting all atoms in A with n).

Let $\langle a, e, b \rangle$ be a transition. Now $a(0)$ and $b(1)$ are interpretations of $R \cup F_0$ and $R \cup F_1$ corresponding to states a and b . Moreover, we can safely write $a(0) \cup e \cup b(1)$ to denote the interpretation of $R \cup F_0 \cup A \cup F_1$ that corresponds to transition $\langle a, e, b \rangle$. (By the definition of rigid, we know that $a(0)$ and $b(1)$ agree on the truth values for all atoms in R .)

For an example of a formula representing a set of transitions, consider again the standard bomb in the toilet domain with two packages. We have $R = \{in(p1), in(p2)\}$, $F_0 = \{armed_0\}$, $F_1 = \{armed_1\}$ and $A = \{dunk(p1), dunk(p2)\}$. For $tr(F_0, A, F_1)$ we can use

$$\begin{aligned}
& (in(p1) \equiv \neg in(p2)) \\
& \wedge (\neg dunk(p1) \vee \neg dunk(p2)) \\
& \wedge (((dunk(p1) \wedge in(p1)) \vee (dunk(p2) \wedge in(p2))) \supset \neg armed_1) \\
& \wedge (\neg((dunk(p1) \wedge in(p1)) \vee (dunk(p2) \wedge in(p2))) \supset (armed_1 \equiv armed_0))
\end{aligned} \tag{2.1}$$

Notice that $tr(F_0, A, F_1) \models \text{state}(F_0)$ and $tr(F_0, A, F_1) \models \text{state}(F_1)$, as expected. After all, if $I \models tr(F_0, A, F_1)$ then I corresponds to a transition $\langle a, e, b \rangle$ (in which a and b are by definition states). In fact, I can be written $a(0) \cup e \cup b(1)$, and we have $a(0) \models \text{state}(F_0)$ and $b(1) \models \text{state}(F_1)$.

2.3.1 Characterization of Plan Validity in Propositional Logic

We have already characterized valid plans (both in the case of classical and in the case of conformant planning) in terms of transition systems. In this section we will see an equivalent characterization in propositional logic.

Again by Fact 2, the set of possible initial states and the set of goal states can be encoded by formulas $init(F)$ and $goal(F)$ over the signature $R \cup F$. It is convenient to require that $init(F)$ correspond exactly to the set of possible initial states, in which case we will also have $init(F) \models state(F)$. On the other hand, for our purposes it is convenient to allow $goal(F)$ to be weaker, so that it is in fact the formula $state(F) \wedge goal(F)$ that corresponds to the set of goal states.

Observe that, with respect to a transition system, the possible histories of length k are the sequences

$$\langle s_0, e_0, s_1, e_1, \dots, s_{k-1}, e_{k-1}, s_k \rangle$$

that correspond to an execution path. Notice that such a sequence corresponds to the interpretation

$$s_0(0) \cup e_0(0) \cup s_1(1) \cup e_1(1) \cup \dots \cup s_{k-1}(k-1) \cup e_{k-1}(k-1) \cup s_k(k)$$

of the signature

$$R \cup \bigcup_{i=0}^{k-1} A_i \cup \bigcup_{i=0}^k F_i.$$

In fact, the possible histories of length k correspond exactly to the models of

$$\bigwedge_{i=0}^{k-1} tr(F_i, A_i, F_{i+1}).$$

Moreover, each such model can be written in the form

$$\bigcup_{i=0}^k s_i(i) \cup \bigcup_{i=0}^{k-1} e_i(i)$$

where each of the s_i 's agrees on all elements of R . Whenever a set of interpretations of $R \cup F$ agree in this way, we will say that they are *R-consistent*.

Let us first deal with plan validity for the special case of classical planning. From our transition systems definition of validity, we know a sequence $\langle e_0, e_1, \dots, e_{k-1} \rangle$ is a valid plan if and only if there are states s_0, \dots, s_k such that $\langle s_0, e_0, s_1, e_1, \dots, s_{k-1}, e_{k-1}, s_k \rangle$ is a path in the transition system and $s_0 \in I$ and $s_k \in G$. Of course in every such instance, the s_i 's are R -consistent. So we can express plan validity for classical planning problems as follows.

For a classical planning problem, a plan $\langle e_0, e_1, \dots, e_{k-1} \rangle$ is valid iff there is an R -consistent set $\{s_0, \dots, s_k\}$ of interpretations of $R \cup F$ such that

1. $s_0(0) \models \text{init}(F_0)$,

- 2.

$$\bigcup_{i=0}^k s_i(i) \cup \bigcup_{i=0}^{k-1} e_i(i) \models \bigwedge_{i=0}^{k-1} \text{tr}(F_i, A_i, F_{i+1}),$$

3. $s_k(k) \models \text{goal}(F_k)$.

Notice that although the goal states are characterized by the conjunction of $\text{goal}(F)$ and $\text{state}(F)$, we don't need to mention $\text{state}(F)$ in the third condition because whenever the first two conditions are satisfied, we know that $s_k(k) \models \text{state}(F_k)$.

From this characterization of a classical plan it is more or less clear how to express the problem of finding a classical plan of a fixed length as a problem of satisfiability of a propositional formula. We will discuss this in a little more detail in the next section.

For a conformant planning problem, a plan $\langle e_0, e_1, \dots, e_{k-1} \rangle$ is valid iff the following hold:

1. $\text{init}(F_0)$ is satisfiable,

2. for all $i \in \{0, \dots, k-1\}$, for all R -consistent sets $\{s_0, \dots, s_i\}$ of interpretations of $R \cup F$, if

$$\bigcup_{n=0}^i s_n(n) \cup \bigcup_{n=0}^{i-1} e_n(n) \models \text{init}(F_0) \wedge \bigwedge_{n=0}^{i-1} \text{tr}(F_n, A_n, F_{n+1}),$$

then there is an interpretation s_{i+1} of $R \cup F$ such that $\{s_0, \dots, s_i, s_{i+1}\}$ is R -consistent and

$$s_i(i) \cup e_i(i) \cup s_{i+1}(i+1) \models tr(F_i, A_i, F_{i+1}),$$

3. for all R -consistent sets $\{s_0, \dots, s_k\}$ of interpretations of $R \cup F$, if

$$\bigcup_{i=0}^k s_i(i) \cup \bigcup_{i=0}^{k-1} e_i(i) \models init(F_0) \wedge \bigwedge_{i=0}^{k-1} tr(F_i, A_i, F_{i+1}),$$

then

$$s_k \models goal(F_k).$$

As before, conditions (1) and (2) together guarantee executability and condition (3) guarantees sufficiency.

We have shown how to represent transition systems in propositional logic and have characterized plan validity both in the case of classical and in the case of conformant planning. In the next section we will observe that the simpler characterization of plan validity that is applicable in the case of classical planning makes it easy to formulate classical planning, for plans of a fixed length, in terms of satisfiability.

2.4 Classical Planning as Propositional Satisfiability

A classical planning problem can be translated into a problem of satisfiability (finding a model) of a propositional formula. As mentioned previously, this approach to solving classical planning problems is called satisfiability planning [9].

For any given k , the set of all R -consistent sets $\{s_0, \dots, s_k\}$ of interpretations of $R \cup F$ has a natural one-to-one correspondence with the set of all interpretations of $R \cup \bigcup_{i=0}^k F_i$. According to the characterization of classical plan validity in the previous section, a plan $\langle e_0, e_1, \dots, e_{k-1} \rangle$ is valid iff there is an R -consistent set $\{s_0, \dots, s_k\}$ of interpretations of $R \cup F$ such that

$$\bigcup_{i=0}^k s_i(i) \cup \bigcup_{i=0}^{k-1} e_i(i) \models init(F_0) \wedge \bigwedge_{i=0}^{k-1} tr(F_i, A_i, F_{i+1}) \wedge goal(F_k).$$

Consequently, the valid plans of length k for a classical planning problem are exactly those which correspond to the models of

$$init(F_0) \wedge \bigwedge_{i=0}^{k-1} tr(F_i, A_i, F_{i+1}) \wedge goal(F_k).$$

That is, each model of this formula can be written in the form $\bigcup_{i=0}^k s_i(i) \cup \bigcup_{i=0}^{k-1} e_i(i)$ where $\langle s_0, e_0, s_1, \dots, e_{k-1}, s_k \rangle$ is a possible history such that $s_0 \models init(F)$ and $s_k \models goal(F)$. Thus, for a classical planning problem, it follows that $\langle e_0, \dots, e_{k-1} \rangle$ is a valid plan of length k . Moreover, every valid plan of length k can be obtained in this manner.

Let us call the above formula $plan(k)$.

2.4.1 Using CCALC for Classical Planning

The Causal Calculator (CCALC) can be used to perform classical planning in a manner similar to the one described above. Essentially, it works by forming a conservative extension of the formula $plan(k)$ and then calling a standard satisfiability solver to find a model. One the advantages of CCALC is that it accepts the action domain description (more specifically a definition of the transition system associated with it) in a very expressive action representation language \mathcal{C} . \mathcal{C} provides a convenient way to express facts about actions and it also handles non-determinism in the domain in a seamless manner.

Given a description of the action domain in \mathcal{C} , CCALC translates the description into a formula in conjunctive normal form that represents the transition system associated with the domain. This process is explained in detail in [7]. A separate plan file can then be provided with formulas expressing the initial conditions and the goal, and specifying the length of the plan. CCALC takes these descriptions as input and forms a propositional formula, also in conjunctive normal form, which is a conservative extension of $plan(k)$, where k is the length of the plan, and then submits this formula to a satisfiability solver to find a model. If a model is found, then the actions that occur in it form the plan. CCALC also provides an option to incrementally vary the plan length and find the shortest plan.

For our purpose of constructing the QBF representing a valid conformant plan, we will

be interested in the propositional formulas constructed by CCALC that encode the transition system, the knowledge of the initial state, and the goal. In summary, the approach discussed in this thesis obtains the following formulas from CCALC:

1. a conservative extension of $tr(F_0, A_0, F_1)$ in conjunctive normal form,
2. a conservative extension of $init(F_0)$ in conjunctive normal form, and
3. a conservative extension of $goal(F_0)$ in conjunctive normal form.

Propositional logic provides a natural way to represent classical planning problems. However, for conformant planning we need a more expressive logic, which we consider next.

2.5 Second-Order Propositional Logic

Second-order propositional logic is an extension of propositional logic which uses quantifiers to increase the expressiveness of the language.

2.5.1 Syntax

As before, a *signature* is a set of symbols called *atoms*. The set of logical symbols includes two quantifiers \forall and \exists , in addition to the propositional connectives \neg , \vee , \wedge , \supset , \equiv , \top , \perp and the left and right parentheses (and).

The set of *formulas* in second-order propositional logic is defined recursively just as for propositional logic, except that we add one more recursive case as follows.

- For any quantifier K and any atom v , if F is a formula then $K v F$ is a formula.

Let $F(v)$ be a formula, with v an atom (that may or may not occur in $F(v)$). For any formula G , we define the result of substituting G for v in $F(v)$, denoted $F(G)$, as follows.

- If $F(v) = v$ then $F(G) = G$.

- If $F(v)$ is an atom other than v or $F(v)$ is a 0-place connective, then $F(G) = F(v)$.
- If $F(v) = \neg F'(v)$ for some formula $F'(v)$, then $F(G) = \neg F'(G)$.
- If $F(v) = (F'(v) \odot F''(v))$ for some formulas $F'(v)$ and $F''(v)$ and binary connective \odot , then $F(G) = (F'(G) \odot F''(G))$.
- If $F(v) = K v F'(v)$ for some quantifier K and formula $F'(v)$, then $F(G) = F(v)$.
- If $F(v) = K w F'(v)$ for some quantifier K , some atom w other than v , and some formula $F'(v)$, then $F(G) = K w F'(G)$.

2.5.2 Semantics

The definition of an interpretation in second-order propositional logic remains the same as an interpretation in propositional logic.

The truth value F^I of a second-order propositional formula F with respect to the interpretation I is defined recursively just as it was in propositional logic, except for the following cases dealing with the quantifiers.

- $\forall w F(w)^I = \mathbf{t}$ if and only if $F(\perp)^I = \mathbf{t}$ and $F(\top)^I = \mathbf{t}$
- $\exists w F(w)^I = \mathbf{t}$ if and only if $F(\perp)^I = \mathbf{t}$ or $F(\top)^I = \mathbf{t}$

The definitions of satisfiability, entailment, equivalence and conservative extension are the same as in propositional logic.

2.5.3 Useful Terminology, Notation Conventions and Facts

It will be useful to note that if F is a conservative extension of G and n_1, \dots, n_k are exactly the new atoms in the extended signature, then $\exists n_1 \dots \exists n_k F$ is equivalent to G . This is a direct consequence of the definition of a conservative extension.

We will be interested in formulas of a special form. So it helps to introduce a few more definitions. A formula is in *prenex normal form* if it is of the form $K_1 a_1 K_2 a_2 \dots K_n a_n F$

where K_1, \dots, K_n are quantifiers, a_1, \dots, a_n are atoms and F is a propositional formula (that is, a formula without quantifiers). A formula in prenex normal form is in *conjunctive normal form* if the propositional part of the formula is in conjunctive normal form (as a propositional formula).

Eventually we will need to find models for a formula. For this purpose we use a solver, which is similar to a propositional satisfiability solvers, except that it finds models for second-order propositional formulas. Most QBF solvers accept formulas only in conjunctive normal form. Therefore the following fact is of interest to us.

Fact 3 *Every formula in second-order logic has an equivalent formula that is in prenex normal form and, hence, an equivalent formula in conjunctive normal form.*

Given the above fact we should be able to use a solver that accepts formulas in conjunctive normal form to find models for any arbitrary formula. In fact, a main contribution of this thesis is a novel method for obtaining, in conjunctive normal form, a conservative extension for the specific kind of QBF that interests us—one that encodes the valid conformant plans of length k .

Next, we define some abbreviations that add convenience to the discussion to follow. We will abbreviate a formula of the form $\exists x \exists y F$ with $\exists xy F$ and $\forall x \forall y F$ with $\forall xy F$. Also if X is a set $\{x_1, \dots, x_n\}$ of atoms, then we write $\exists X$ to stand for $\exists x_1 \dots x_n$ and similarly $\forall X$ to stand for $\forall x_1 \dots x_n$.

We observe some facts that are a direct consequence of the definition of semantics. We will find them useful later in the thesis when we find equivalent forms of a QBF.

Fact 4 *For any formula F ,*

1. $\exists x \exists y F \leftrightarrow \exists y \exists x F$,
2. $\forall x \forall y F \leftrightarrow \forall y \forall x F$.

Roughly speaking, what we observe here is that, whenever we find a group of atoms quantified with the same quantifier together, we can change their order and still preserve

equivalence. We will find this fact useful when we reorder quantifiers in our formulation of the QBF representing a conformant plan.

Fact 5 For any two formulas F and G , $\forall x F \wedge \forall x G \leftrightarrow \forall x (F \wedge G)$.

Intuitively this means that the universal quantifier can be distributed over conjunctions. Before presenting some more useful facts we need to introduce a new definition.

The set of *free* atoms of a formula F is defined recursively as follows:

- If F is an atom then F is free in F .
- 0-place connectives do not have any free atoms.
- The free atoms of the formula $\neg F$ are exactly the same as the free atoms of F .
- The free atoms of $(F \odot G)$ are the free atoms of F plus the free atoms of G .
- For any quantifier K , the free atoms of $K v F$ are all the free atoms of F except v .

In propositional logic we saw that every formula can be seen as a truth-valued function over the set of interpretations. More precisely they can be seen as truth-valued functions over the set of interpretations of atoms that occur in the formula. Similarly, formulas in second-order propositional logic can be seen as truth-valued functions over the set of interpretations of the free atoms that occur in it. It also helps to think of models of a formula as just the interpretation of the free atoms of the formula. (Observe that interpretations which do not differ in the truth values of free atoms of a formula F , satisfy F on an all or none basis.)

The definition of free atoms immediately leads to some more observations.

Fact 6 For any two formulas F and G , if x is not a free atom of G then,

1. $\exists x F(x) \supset G \leftrightarrow \forall x (F(x) \supset G)$,
2. $G \supset \exists x F(x) \leftrightarrow \exists x (G \supset F(x))$.

Proof: Notice that, because x is not free in G , any substitution for x in G is equal to G . The rest of the proof just uses the cases of quantifiers in the definition of semantics, and a property of propositional logic.

$$\begin{aligned}\forall x (F(x) \supset G) &\leftrightarrow (F(\top) \supset G) \wedge (F(\perp) \supset G) \\ &\leftrightarrow (F(\top) \vee F(\perp)) \supset G \\ &\leftrightarrow \exists x F(x) \supset G\end{aligned}$$

The proof for the second part is similar.

Observe that this fact gives us the ability to move existential quantifiers outside of an implication, adjusting the quantification if needed. This will prove to be very useful for us.

Fact 7 For any two formulas F and G , if x is not a free atom of G , then

$$\exists x F(x) \wedge G \leftrightarrow \exists x (F(x) \wedge G).$$

Proof: Here again, we make use of the fact that substituting for an atom that is not free does not affect the formula.

$$\begin{aligned}\exists x (F(x) \wedge G) &\leftrightarrow (F(\top) \wedge G) \vee (F(\perp) \wedge G) \\ &\leftrightarrow (F(\top) \vee F(\perp)) \wedge G \\ &\leftrightarrow \exists x F(x) \wedge G\end{aligned}$$

This fact allows us to move an existential quantifier out over conjunction. Facts 5–7 together will be helpful in obtaining equivalent QBF's in prenex normal form.

Second-order propositional logic is also known as *quantified boolean logic* and the formulas are referred to as quantified boolean formulas or simply *QBF*.

2.6 Conformant Planning in Second-Order Propositional Logic

In Section 2.1, we used the transition system representation of an action domain to characterize a valid conformant plan. Then we expressed the same knowledge in terms of formulas in propositional logic. In this section we will see the later definition of a valid plan extends naturally to a QBF encoding of a conformant planning problem as described in [14].

For convenience we restate the characterization of validity of a plan (in terms of propositional logic). For a conformant planning problem, a plan $\langle e_0, e_1, \dots, e_{k-1} \rangle$ is valid iff the following hold:

1. $init(F)$ is satisfiable,
2. for all $i \in \{0, \dots, k-1\}$, for all R -consistent sets $\{s_0, \dots, s_i\}$ of interpretations of $R \cup F$, if

$$\bigcup_{n=0}^i s_n(n) \cup \bigcup_{n=0}^{i-1} e_n(n) \models init(F_0) \wedge \bigwedge_{n=0}^{i-1} tr(F_n, A_n, F_{n+1}),$$

then there is an interpretation s_{i+1} of $R \cup F$ such that $\{s_0, \dots, s_i, s_{i+1}\}$ is R -consistent and

$$s_i(i) \cup e_i(i) \cup s_{i+1}(i+1) \models tr(F_i, A_i, F_{i+1}),$$

3. for all R -consistent sets $\{s_0, \dots, s_k\}$ of interpretations of $R \cup F$, if

$$\bigcup_{i=0}^k s_i(i) \cup \bigcup_{i=0}^{k-1} e_i(i) \models init(F_0) \wedge \bigwedge_{i=0}^{k-1} tr(F_i, A_i, F_{i+1}),$$

then

$$s_k \models goal(F_k).$$

Each of these three conditions can be nicely expressed as a QBF.

Condition (1) is expressed by the QBF

$$\exists R \exists F_0 \, init(F_0). \tag{2.2}$$

The rest of the discussion will assume that formula (2.2) is satisfiable. (This is easy enough to check with a sat solver.)

Condition (3) is expressed by

$$\forall R \forall F_0 \cdots \forall F_k \left(\left(\text{init}(F_0) \wedge \bigwedge_{i=0}^{k-1} \text{tr}(F_i, A_i, F_{i+1}) \right) \supset \text{goal}(F_k) \right). \quad (2.3)$$

The free atoms in formula (2.3) are $\bigcup_{i=0}^{k-1} A_i$, and the models of (2.3) are exactly the interpretations of these atoms that correspond to the plans of length k that satisfy condition (3). Notice that the universal quantification over R, F_0, \dots, F_k in formula (2.3) exactly captures the corresponding quantification over all R -consistent sets $\{s_0, \dots, s_k\}$ in condition (3). The if-then structure of condition (3) is expressed by implication (\supset) in (2.3).

Similarly, condition (2) is expressed by a family of QBFs. For each $i \in \{0, \dots, k-1\}$,

$$\forall R \forall F_0 \cdots \forall F_i \left(\left(\text{init}(F_0) \wedge \bigwedge_{n=0}^{i-1} \text{tr}(F_n, A_n, F_{n+1}) \right) \supset \exists F_{i+1} \text{tr}(F_i, A_i, F_{i+1}) \right) \quad (2.4)$$

We would like to have a single QBF that we can submit to the QBF solver. To this end we note that condition (1) just guarantees that the set of initial states is not empty. This is necessary for plan validity. But on the other hand this formula does not affect the plan, when indeed there is a possible initial state. So it is enough to check if formula (2.2) is equivalent to \top , and, if it is, then find a model the conjunction of the formulas (2.4) and formula (2.3).

For later convenience, we note that the conjunction of formulas (2.4) and (2.3) can be

written as follows.

$$\begin{aligned}
& \forall R \forall F_0 (init(F_0) \supset \exists F_1 tr(F_0, A_0, F_1)) \\
\wedge & \forall R \forall F_0 \forall F_1 ((init(F_0) \wedge tr(F_0, A_0, F_1)) \supset \exists F_2 tr(F_1, A_1, F_2)) \\
\wedge & \forall R \forall F_0 \forall F_1 \forall F_2 ((init(F_0) \wedge tr(F_0, A_0, F_1) \wedge tr(F_1, A_1, F_2)) \supset \\
& \qquad \qquad \qquad \exists F_3 tr(F_2, A_2, F_3)) \\
& \vdots \\
\wedge & \forall R \forall F_0 \forall F_1 \cdots \forall F_{k-1} ((init(F_0) \wedge tr(F_0, A_0, F_1) \wedge \cdots \\
& \qquad \qquad \qquad \wedge tr(F_{k-2}, A_{k-2}, F_{k-1})) \supset \exists tr(F_{k-1}, A_{k-1}, F_k)) \\
\wedge & \forall R \forall F_0 \forall F_1 \cdots \forall F_k ((init(F_0) \wedge tr(F_0, A_0, F_1) \wedge \cdots \\
& \qquad \qquad \qquad \wedge tr(F_{k-1}, A_{k-1}, F_k)) \supset goal(k)).
\end{aligned} \tag{2.5}$$

If the formula (2.5) is satisfiable then the actions that occur in the model constitute a valid plan.

In the next section we will discuss in more detail the formulas produced by CCALC that we plan to use as building blocks in the approach presented in the thesis. Then we show how we can build a conservative extension of the above QBF using the formulas produced by CCALC. Our goal then is to present the QBF in a form accepted by the QBF solver.

2.7 Formulas Produced by CCALC

As we have already discussed, CCALC can be used to do satisfiability planning in a manner similar to that described in Section 2.3. We have also remarked that CCALC does not produce the formulas $tr(F_0, A, F_1)$, $init(F)$ and $goal(F)$; instead it produces conservative extensions of them. In this section we will describe this conservative extension complication in sufficient detail in order to prepare for the approach presented in this thesis.

To start with, CCALC accepts a description of the action domain in action language \mathcal{C} . Similarly, it accepts descriptions of the possible initial states and the goal in a plan file which also specifies a length k for the plan. CCALC's objective then is to construct the formula $plan(k)$ as described in Section 2.3 and then present it in a form acceptable

to a propositional satisfiability solver. But standard propositional satisfiability solvers accept formulas only in conjunctive normal form, whereas $plan(k)$ could be an arbitrary propositional formula. So CCALC has to find a formula equivalent to $plan(k)$ which is in conjunctive normal form. Standard methods to find an equivalent of a propositional formula in conjunctive normal form have a worst-case exponential cost in terms of the size of the formula produced. This is undesirable because it could affect the time taken by the solver to search for a satisfying interpretation.

Fortunately, there are well-known algorithms for taking an arbitrary formula and producing a conservative extension of the formula, in conjunctive normal form, whose size is linear in the input. CCALC uses a variant of one of these algorithms to produce a conservative extension of $plan(k)$ in conjunctive normal form.

From the action domain description, CCALC forms a conservative extension of the formula $tr(F_0, A, F_1)$ in conjunctive normal form. In the process it may introduce some new atoms. (We later present an algorithm similar to the one used by CCALC for this purpose.) Let V be the set of new atoms introduced, and let $tr(F_0, A, V, F_1)$ be the formula that is produced. Notice that while $tr(F_0, A, F_1)$ is a formula over the signature $R \cup F_0 \cup A \cup F_1$, $tr(F_0, A, V, F_1)$ is a formula over the extended signature $R \cup F_0 \cup A \cup V \cup F_1$. We know from the definition of a conservative extension that the models of $tr(F_0, A, V, F_1)$ correspond to the models (over of $R \cup F_0 \cup A \cup F_1$) of $tr(F_0, A, F_1)$.

CCALC can be also given descriptions of the possible initial states and the goal, along with the length k of the plan sought. In a similar manner, CCALC produces conservative extensions of $init(F)$ and $goal(F)$ in conjunctive normal form. Again new atoms could be introduced in each case and let I and G be the sets of new atoms introduced. Let $init(F, I)$ and $goal(F, G)$ be the formulas produced. Then CCALC makes $k + 1$ copies of $tr(F_0, A, V, F_1)$ and constructs the following conservative extension of $plan(k)$:

$$init(F_0, I_0) \wedge \bigwedge_{i=0}^{k-1} tr(F_i, A_i, V_i, F_{i+1}) \wedge goal(F_k, G_k).$$

Then this formula is submitted to the solver to find a model. Notice that the models of this

formula are interpretations over the signature

$$\bigcup_{i=0}^k F_i \cup \bigcup_{i=0}^{k-1} A_i \cup \bigcup_{i=0}^{k-1} V_i.$$

A valid plan is then extracted from the model.

Our interest in CCALC is the three formulas $tr(F_0, A, V, F_1)$, $init(F, I)$ and $goal(F, G)$. We obtain these formulas from CCALC along with details about the sets of new atoms introduced and the set of rigid atoms. Then we use this information to construct a QBF in conjunctive normal form that encodes the valid conformant plans. Notice that, here, the new atoms introduced did not present any inconvenience. This is due to the fact that classical planning is a rather special case of planning. But when we try to use these formulas with new atoms to build a QBF that encodes a conformant plan, there are some complications and we work out the details of the process in Chapter 3.

Chapter 3

Approach

We have already seen how to use CCALC to obtain the formulas $tr(F_0, A, V, F_1)$, $init(F, I)$ and $goal(F, G)$. In this chapter we work out the details involved in using these formulas along with the other information obtained from CCALC to produce a QBF in conjunctive normal form that represents all valid conformant plans of a given length k . Section 3.1 shows how a QBF in prenex normal form can be formed using the above formulas and related information obtained from CCALC. Then our goal would be to convert the propositional part into conjunctive normal form. Section 3.2 gives an overview of a standard method to find an equivalent of a propositional formula in conjunctive normal form and explains why such a method proves infeasible for us. Then in Section 3.3 we relax our constraints from requiring an equivalent form to requiring a conservative extension. Again we show a standard method which is not very effective and proceed to present the approach taken in this thesis, one which takes advantage of the structure of the formulas we have at hand.

3.1 From CCALC Formulas to a QBF in Prenex Normal Form

In Section 2.6, we showed a QBF (2.5) that represents all valid conformant plans (of length k) for a given problem. Our goal is to find a model of this formula. It would be fairly straightforward to build this formula if we had the components that occur in it—the formulas $init(F_i)$, $tr(F_i, A_i, F_{i+1})$ and $goal(F_i)$. But the formulas we obtain from CCALC— $tr(F_0, A, V, F_1)$, $init(F, I)$ and $goal(F, G)$ —are their respective conservative extensions. So we are left with the task of finding a formula equivalent to formula (2.5) that can be constructed with the formulas we have from CCALC. More specifically, all we need is a conservative extension of formula (2.5). Strictly speaking, we will find a conservative extension over an extended signature. But the set of free atoms in these two formulas remains the same, and therefore as formulas in the extended signature they are equivalent.

Since we use the formulas obtained from CCALC extensively, it is convenient to introduce a shorthand notation for each of these. In all the discussion to follow we will use the following abbreviations for each i :

$$\begin{aligned} in_i & \text{ for } init(F_i, I_i), \\ tr_{i,i+1} & \text{ for } tr(F_i, A_i, V_{i+1}, F_{i+1}), \\ gl_i & \text{ for } goal(F_i, G_i). \end{aligned}$$

From the definition of a conservation extension the following equivalences follow. For each i :

$$\begin{aligned} init(F_i) & \leftrightarrow \exists I_i in_i \\ tr(F_i, A_i, F_{i+1}) & \leftrightarrow \exists V_{i+1} tr_{i,i+1} \\ goal(F_i) & \leftrightarrow \exists G_i gl_i \end{aligned}$$

Replacing parts of formula (2.5) with the corresponding equivalent formulas from

above, we get an equivalent of formula (2.5).

$$\begin{aligned}
& \forall R \forall F_0 (\exists I_0 in_0 \supset \exists V_1 F_1 tr_{0,1}) \\
\wedge & \forall R \forall F_0 \forall F_1 ((\exists I_0 in_0 \wedge \exists V_1 tr_{0,1}) \supset \exists F_2 \exists V_2 tr_{1,2}) \\
\wedge & \forall R \forall F_0 \forall F_1 \forall F_2 ((\exists I_0 in_0 \wedge \exists V_1 tr_{0,1} \wedge \exists V_2 tr_{1,2}) \supset \exists F_3 \exists V_3 tr_{2,3}) \\
& \vdots \\
\wedge & \forall R \forall F_0 \forall F_1 \cdots \forall F_{k-1} ((\exists I_0 in_0 \wedge \exists V_1 tr_{0,1} \wedge \cdots \wedge \exists V_{k-1} tr_{k-2,k-1}) \supset \exists F_k \exists V_k tr_{k-1,k}) \\
\wedge & \forall R \forall F_0 \forall F_1 \cdots \forall F_k ((\exists I_0 in_0 \wedge \exists V_1 tr_{0,1} \wedge \cdots \wedge \exists V_k tr_{k-1,k}) \supset \exists G_k gl_k)
\end{aligned} \tag{3.1}$$

At this point, we have a QBF that is equivalent to formula (2.5). (Notice that this formula is over an extended signature.) Also it is clear that this QBF can be constructed easily using the formulas from CCALC. But our goal is to find a model of this QBF and therefore we need to present this to a QBF solver. In order to use the QBF solver, we need to construct an equivalent to formula (3.1), or a conservative extension of (3.1), in conjunctive normal form. We will find an equivalent formula over an extended signature. We split this process into two tasks. First we find an equivalent in prenex normal form and then find an equivalent of the propositional part in conjunctive normal form.

The first task, which is to find a prenex equivalent to formula (3.1), will be done in a series of steps. At each step we preserve equivalence.

In formula (3.1), there are several implications. The antecedent of each implication is a conjunction and each of the conjuncts is existentially quantified. We first pull these existential quantifiers out over conjunction using Fact 7 and then further move the existential quantifiers out over implication using Fact 6 to get the following equivalent formula:

$$\begin{aligned}
& \forall R \forall F_0 \forall I_0 (in_0 \supset \exists F_1 \exists V_1 tr_{0,1}) \\
\wedge & \forall R \forall F_0 \forall F_1 \forall I_0 \forall V_1 ((in_0 \wedge tr_{0,1}) \supset \exists F_2 \exists V_2 tr_{1,2}) \\
\wedge & \forall R \forall F_0 \forall F_1 \forall F_2 \forall I_0 \forall V_1 \forall V_2 ((in_0 \wedge tr_{0,1} \wedge tr_{1,2}) \supset \exists F_3 \exists V_3 tr_{2,3}) \\
& \vdots \\
\wedge & \forall R \forall F_0 \forall F_1 \cdots \forall F_{k-1} \forall I_0 \forall V_1 \forall V_2 \cdots \forall V_{k-1} ((in_0 \wedge tr_{0,1} \wedge \cdots \\
& \qquad \qquad \qquad \cdots \wedge tr_{k-2,k-1}) \supset \exists F_k \exists V_k tr_{k-1,k}) \\
\wedge & \forall R \forall F_0 \forall F_1 \cdots \forall F_k \forall I_0 \forall V_1 \forall V_2 \cdots \forall V_k ((in_0 \wedge tr_{0,1} \wedge \cdots \wedge tr_{k-1,k}) \supset \exists G_k gl_k)
\end{aligned} \tag{3.2}$$

Now we move all the universal quantifiers outside, using Fact 5. To avoid later “clashes” between names of atoms, we also replace, for each i , $\exists F_{i+1} \exists V_{i+1} tr_{i,i+1}$ with $\exists F'_{i+1} \exists V'_{i+1} tr'_{i,i+1}$ where $tr'_{i,i+1}$ stands for $tr(F_i, A_i, V_{i+1}, F'_{i+1})$, and F'_{i+1} and V'_{i+1} are primed copies of F_{i+1} and V_{i+1} .

$$\begin{aligned}
\forall R \forall F_0 \cdots \forall F_k \forall I_0 \forall V_0 \cdots \forall V_k & \left((in_0 \supset \exists F'_1 \exists V'_1 tr'_{0,1}) \right. \\
& \wedge ((in_0 \wedge tr_{0,1}) \supset \exists F'_2 \exists V'_2 tr'_{1,2}) \\
& \wedge ((in_0 \wedge tr_{0,1} \wedge tr_{1,2}) \supset \exists F'_3 \exists V'_3 tr'_{2,3}) \\
& \vdots \\
& \wedge ((in_0 \wedge tr_{0,1} \wedge \cdots \wedge tr_{k-2,k-1}) \supset \exists F'_k \exists V'_k tr'_{k-1,k}) \\
& \left. \wedge ((in_0 \wedge tr_{0,1} \wedge \cdots \wedge tr_{k-1,k}) \supset \exists G_k gl_k) \right)
\end{aligned} \tag{3.3}$$

Using Fact 6, we move the existential quantifiers in the consequent of each implication out of the implication to obtain the following formula.

$$\begin{aligned}
& \forall R \forall F_0 \dots \forall F_k \forall I_0 \forall V_0 \dots \forall V_k \left(\exists F'_1 \exists V'_1 (in_0 \supset tr'_{0,1}) \right. \\
& \quad \wedge \exists F'_2 \exists V'_2 ((in_0 \wedge tr_{0,1}) \supset tr'_{1,2}) \\
& \quad \wedge \exists F'_3 \exists V'_3 ((in_0 \wedge tr_{0,1} \wedge tr_{1,2}) \supset tr'_{2,3}) \\
& \quad \vdots \\
& \quad \wedge \exists F'_k \exists V'_k ((in_0 \wedge tr_{0,1} \wedge \dots \wedge tr_{k-2,k-1}) \supset tr'_{k-1,k}) \\
& \quad \left. \wedge \exists G_k ((in_0 \wedge tr_{0,1} \wedge \dots \wedge tr_{k-1,k}) \supset gl_k) \right)
\end{aligned} \tag{3.4}$$

Finally, we use Fact 7 again to move the existential quantifiers out over the conjunctions to get the following formula in prenex normal form, which is equivalent to formula (3.1) and hence to formula (2.5).

$$\begin{aligned}
& \forall R \forall F_0 \dots \forall F_k \forall I_0 \forall V_0 \dots \forall V_k \exists F'_1 \dots \exists F'_k \exists V'_1 \dots \exists V'_k \exists G_k \\
& \quad \left((in_0 \supset tr'_{0,1}) \right. \\
& \quad \wedge ((in_0 \wedge tr_{0,1}) \supset tr'_{1,2}) \\
& \quad \wedge ((in_0 \wedge tr_{0,1} \wedge tr_{1,2}) \supset tr'_{2,3}) \\
& \quad \vdots \\
& \quad \wedge ((in_0 \wedge tr_{0,1} \wedge \dots \wedge tr_{k-2,k-1}) \supset tr'_{k-1,k}) \\
& \quad \left. \wedge ((in_0 \wedge tr_{0,1} \wedge \dots \wedge tr_{k-1,k}) \supset gl_k) \right)
\end{aligned} \tag{3.5}$$

The propositional part of the formula we have obtained is not in conjunctive normal form. But we require this in order to be able to present the QBF to the solver. The rest of the chapter will address this issue.

3.2 A Standard Algorithm

In this section we present a standard algorithm that converts an arbitrary propositional formula into conjunctive normal form, i.e., finds an equivalent propositional formula that

is in conjunctive normal form, and then explain why such a conversion is infeasible for our purposes. We will use the term *clause* to refer to a simple disjunction.

Given two formulas A and B in conjunctive normal form, $\text{DIS}(A, B)$ returns a formula equivalent to $A \vee B$ in conjunctive normal form.

$\text{DIS}(A, B)$

1. Let A_1, \dots, A_n be the clauses of A and let B_1, \dots, B_m be the clauses of B .
2. Return

$$\bigwedge_{i=1}^n \bigwedge_{j=1}^m (A_i \vee B_j)$$

For any atom a , let $\bar{a} = \neg a$ and let $\overline{\bar{a}} = a$.

Given a formula A in conjunctive normal form, $\text{NEG}(A)$ returns a formula equivalent to $\neg A$ in conjunctive normal form.

$\text{NEG}(A)$

1. If $A = \bigvee_{i=1}^n L_i$ where L_i are literals, return $\bigwedge_{i=1}^n \overline{L_i}$.
2. Let $A = B \wedge C$ where B is a clause.
3. Return $\text{DIS}(\text{NEG}(B), \text{NEG}(C))$.

Given two formulas A and B in conjunctive normal form, $\text{IMP}(A, B)$ returns a formula equivalent to $A \supset B$ in conjunctive normal form.

$\text{IMP}(A, B)$

1. Return $\text{DIS}(\text{NEG}(A), B)$.

Given two formulas A and B in conjunctive normal form, $\text{EQUIV}(A, B)$ returns a formula equivalent to $A \equiv B$ in conjunctive normal form.

$\text{EQUIV}(A, B)$

1. Return $\text{IMP}(A, B) \wedge \text{IMP}(B, A)$.

Now we are ready to describe the method that converts an arbitrary propositional formula to conjunctive normal form.

$\text{CLAUSIFY}(F)$

1. If F is an atom return F .
2. If $F = \neg G$ return $\text{NEG}(\text{CLAUSIFY}(G))$.
3. If $F = (G \vee H)$ return $\text{DIS}(\text{CLAUSIFY}(G), \text{CLAUSIFY}(H))$.
4. If $F = (G \wedge H)$ return $\text{CLAUSIFY}(G) \wedge \text{CLAUSIFY}(H)$.
5. If $F = (G \supset H)$ return $\text{IMP}(\text{CLAUSIFY}(G), \text{CLAUSIFY}(H))$.
6. If $F = (G \equiv H)$ return $\text{EQUIV}(\text{CLAUSIFY}(G), \text{CLAUSIFY}(H))$.

3.2.1 Problems with this Method

Let us investigate the nature of the procedure $\text{NEG}(A)$. If A has n clauses each having m literals in it. Then $\text{NEG}(A)$ has m^n clauses each of length n . For example, a formula with 10 clauses each with 5 literals in it will produce a formula with 5^{10} clauses each of length 10. This method is used whenever there is any connective other than conjunction. So even in the case of a formula of reasonable size, if there are connectives other than conjunction, the method CLAUSIFY could return a very large formula, which in many cases would be difficult for a QBF solver to handle.

Clearly the size of the formula produced grows exponentially and the time complexity cannot be any better. As a result we have two problems with this method which make it infeasible even in very small action domains.

First the algorithm takes exponential time to terminate, which is highly undesirable because we want to construct this formula in negligible time (linear time is okay) when compared to time taken by the solver to decide if the QBF is satisfiable.

The second and probably more important problem is that the formula so produced is of exponential size. This essentially means that there is no hope of running the QBF solver efficiently on the resultant formula.

We will discuss subsequent improvements to this method which present a way to construct in linear time a QBF representing the valid conformant plans. The formula produced is also of linear size.

3.3 Further Improvements

The method presented in the previous section finds an equivalent to any arbitrary propositional formula. On the other hand we do not absolutely need to find an equivalent formula. A conservative extension is enough for our purpose. Then we can handle it very much the same way we treated the conservative extensions obtained from CCALC.

3.3.1 METHOD1: Clausification with New Atoms

The method METHOD1 presented here is not one that finds an equivalent formula, instead it is a standard algorithm that finds a conservative extension of an arbitrary propositional formula in conjunctive normal form. In the process it introduces some new atoms.

CLAUSIFY^{*}(F)

1. If F is a conjunction of clauses, return F .
2. Let G be a minimal non-literal subformula of F .
3. Let u be a new atom¹.
4. Let F' be the result of replacing all occurrences of G in F by u .

¹Here and elsewhere in the thesis, it is assumed that there is a single new atom generator that generates a new atom on demand, never runs out of new atoms and these atoms are different from the atoms used in other formulas.

5. Return $\text{CLAUSIFY}^*(F') \wedge \text{CLAUSIFY}(u \equiv G)$

In fact, CCALC uses an algorithm similar to METHOD1 to hold down the size of the formulas it produces. The algorithm described above produces a conservative extension of linear size in linear time. Nevertheless, METHOD1 has a disadvantage. The formula it forms introduces a new atom for (almost) every binary connective in the original formula. In a fairly large formula this can result in the introduction of too many new atoms. Also METHOD1, being a general purpose algorithm, does not take advantage of the structure of the formula we have at hand. We present an alternative that will also produce a conservative extension but with many fewer atoms. As a result, the size of the resulting formula is also much less than what METHOD1 would produce (for the formulas we deal with).

3.3.2 METHOD2: Exploiting the Formula Structure

We first present some auxiliary methods that are helpful in describing our main method.

Given a formula B in conjunctive normal form and an atom a , the following method produces a conservative extension of $B \supset a$ in conjunctive normal form.

$\text{IMP}^*(B, a)$

1. Let B be of the form $\bigwedge_{i=1}^n \left(\bigvee_{j=1}^{k_i} L_{ij} \right)$, where the L_{ij} are literals.
2. Let c_1, c_2, \dots, c_n be new atoms.
3. Return

$$\left(\bigvee_{i=1}^n \neg c_i \vee a \right) \wedge \bigwedge_{i=1}^n \bigwedge_{j=1}^{k_i} (\neg L_{ij} \vee c_i)$$

The following algorithm produces a conservative extension of $B \equiv a$ in conjunctive normal form, where B and a are as above.

$\text{EQUIV}^*(a, B)$

1. Return $\text{IMP}^*(B, a) \wedge \text{IMP}(a, B)$

Our main method uses the structure of the propositional part of formula (3.5) and hence is only applicable to formulas that have the same form. For convenience, we restate it (only the propositional part) using the shorthands we have already introduced.

$$\begin{aligned}
& (in_0 \supset tr'_{0,1}) \\
\wedge & \left((in_0 \wedge tr_{0,1}) \supset tr'_{1,2} \right) \\
\wedge & \left((in_0 \wedge tr_{0,1} \wedge tr_{1,2}) \supset tr'_{2,3} \right) \\
& \vdots \\
\wedge & \left((in_0 \wedge tr_{0,1} \wedge \cdots \wedge tr_{k-2,k-1}) \supset tr'_{k-1,k} \right) \\
\wedge & \left((in_0 \wedge tr_{0,1} \wedge \cdots \wedge tr_{k-1,k}) \supset gl_k \right)
\end{aligned} \tag{3.6}$$

If we call the above formula F , the following algorithm finds a conservative extension of F in conjunctive normal form. The idea, essentially, is to replace the antecedent and consequent in each of the implications with an equivalent “name”.

METHOD2(F)

1. Let a_0, a_1, \dots, a_k and c_0, c_1, \dots, c_k be new atoms.
2. Return

$$\begin{aligned}
& \text{EQUIV}^*(a_0, in_0) \\
\wedge & \bigwedge_{i=1}^k \text{EQUIV}^*(a_i, a_{i-1} \wedge tr_{i-1,i}) \\
\wedge & \bigwedge_{i=0}^{k-1} \text{EQUIV}^*(c_i, tr'_{i,i+1}) \\
\wedge & \text{EQUIV}^*(c_k, gl_k) \\
\wedge & \bigwedge_{i=0}^k (a_i \supset c_i)
\end{aligned}$$

Let G be the formula obtained from METHOD2(F) and N be the set of new atoms produced in the process. Then

$$F \leftrightarrow \exists N G \tag{3.7}$$

and therefore formula (3.5) is equivalent to

$$\forall R \forall F_0 \cdots \forall F_k \forall I_0 \forall V_0 \cdots \forall V_k \exists F'_1 \cdots \exists F'_k \exists V'_1 \cdots \exists V'_k \exists N G. \tag{3.8}$$

Notice that the signature of formula (3.8) contains new atoms also. But formula (3.5) is also a formula in this extended signature. (It just does not use the new atoms.) When we say that these two formulas are equivalent, we implicitly mean that the underlying signature is the extended one.

We can submit formula (3.8) to our QBF solver and the action atoms that are true in the model found will form a plan (solution) for our conformant problem.

3.3.3 METHOD3: Fine Tuning

By making slight modifications to the above algorithm, we can further reduce the size of the resulting formula. The following revision to METHOD2 shows the changes.

If we call formula (3.6) F , the following algorithm finds a conservative extension of F in conjunctive normal form.

METHOD3(F)

1. Let m_0, m_1, \dots, m_k be new atoms.
2. Return

$$\begin{aligned}
& \text{IMP}^*(in_0, m_0) \\
\wedge & \bigwedge_{i=1}^k \text{IMP}^*(m_{i-1} \wedge tr_{i-1,i}, m_i) \\
\wedge & \bigwedge_{i=0}^{k-1} \text{IMP}(m_i, tr'_{i,i+1}) \\
\wedge & \text{IMP}(m_k, gl_k)
\end{aligned}$$

Let G be the formula obtained from METHOD3(F). For each $i \in \{0, \dots, k\}$, let M_i be the set of new atoms introduced by the above call to IMP^* in which the second argument is m_i , and then take $N_i = \{m_i\} \cup M_i$. Now by property of conservative extension we get an equivalent to formula (3.5) as:

$$\forall R \forall F_0 \dots \forall F_k \forall I_0 \forall V_0 \dots \forall V_k \exists F'_1 \dots \exists F'_k \exists V'_1 \dots \exists V'_k \exists G_k \exists N_0 \dots \exists N_k G. \quad (3.9)$$

Formula (3.9) can be submitted to a QBF solver to obtain a model. Recall that a model contains the interpretation of the free atoms that satisfies the QBF. In our case the free

atoms are exactly the action atoms. So from the output of the solver, a valid conformant plan can be extracted easily.

3.3.4 A Natural Way of Quantification

We tried an equivalent encoding which retains a more “natural” way of quantification. We explain this equivalent form briefly.

Starting from formula (2.5), before moving any quantifiers out, if we replace the parts of the formula with their conservative extensions, essentially as computed in METHOD3, and then move the quantifiers out (over implication and conjunction) in a different order, we can get the following equivalent formula.

$$\begin{aligned}
& \forall R \forall F_0 \forall I_0 \exists F'_1 \exists V'_1 \exists N_0 \\
& \forall F_1 \forall V_1 \exists F'_2 \exists V'_2 \exists N_1 \\
& \forall F_2 \forall V_2 \exists F'_3 \exists V'_3 \exists N_2 \\
& \vdots \\
& \forall F_{k-1} \forall V_{k-1} \exists F'_k \exists V'_k \exists N_{k-1} \\
& \forall F_k \forall V_k \exists N_k \exists G_k \quad \mathbf{G}
\end{aligned} \tag{3.10}$$

where \mathbf{G} is same as above and the N_i s have the same meaning as above. This formula has an interesting property. The quantification for each set of atoms has been moved as far inside as possible. This depicts the dependency relation between the sets of atoms more naturally. For example, the choice of truth values for $F'_1 \cup V'_1 \cup N_0$ depends on the choice of truth values for $R \cup F_0 \cup I_0$ but not on the choice of truth values for $\bigcup_{i=1}^k (F_i \cup V_i)$. That is, once we have truth values for $R \cup F_0 \cup I_0$ we can immediately answer the question whether, given those truth values, it is possible to find a satisfying interpretation for the part of the formula involving $F'_1 \cup V'_1 \cup N_0$. In terms of the definition of a valid plan, this is roughly analagous to the observation that, in order to determine whether the next action in a plan can be executed, what is important is just the previous execution history.

In spite of the fact that this is a simple change from the previous encoding, this change resulted in significant improvements to the performance in terms of search times of the

solver. This also suggests that there could be still better ways of quantification, although at this point, it is not clear if there are any. Notice that the propositional part of the formula is still the same and hence the size of the formula.

In the next chapter, we present results obtained from our experiments and comment on our observations and conclusions.

Chapter 4

Experiments

The approach described in this thesis (using the natural encoding) was implemented as a program in Perl. It takes as input the formulas corresponding to the set of possible initial states, the transitions of the transition system and the goal. It also takes as input the number of rigid and non-rigid fluents in the domain, the number of actions and the number of new atoms introduced by CCALC. (These atoms are assumed to constitute a prefix of the positive integers, and the classes of atoms are assumed to occur in a fixed order in the sequence. Thus, the identity of the classes of atoms can be determined from the counts given as input.) The program assumes for simplicity that the number of new atoms introduced in the initial state formula and the goal formula is zero. The length of plan sought is also given as input. The implementation program then forms a QBF in conjunctive normal form that encodes all valid plans of the given length, as described in the previous chapter. This formula is then presented to QUBE [8], a QBF satisfiability solver publicly available at <http://www.mrg.dist.unige.it/star/qube>, to find a model. Each model corresponds to a valid plan in the straightforward manner we have described.

We conducted experiments on three variations of the bomb in the toilet problem—Standard Bomb in Toilet (BT), Bomb in Toilet with Deterministic Clogging (BTC) and Bomb in Toilet with Non-deterministic Clogging (BTNC). In all the examples we discussed so far, there was only one toilet. As a result, only one action could be performed at a given

time. In other words, there was no concurrency. We conducted experiments with multiple toilets as well. In this case, we allow dunking in multiple toilets at the same time. As a result, for a given number of packages, the shortest valid plans are shorter when there are more toilets available for dunking. While introducing multiple toilets complicates the domain, it can affect the search for a valid plan positively. Shortly, we will see why.

There are two parameters associated with each member of these families, the first one is number of toilets and the second is the number of packages. For example, BTNC(4, 2) refers to the Bomb in Toilet problem with Non-Deterministic Clogging with 4 toilets and 2 packages. In each case, the QBF was built for the optimal length plan. The times recorded are search times reported by the solver (cpu times). The searches were limited to a maximum of 1200 seconds, after which the solver would timeout. All experiments were run on a PC running Linux.

The tables are presented at the end of this chapter. The row header in the table shows the number of packages and the column header shows the number of toilets. The tables are named after the problem and the QBF encoding used. We call the encoding as in formula (3.9) the “original” encoding and the one in formula (3.10) “natural” encoding.

For example, in the table “BT - Family - Original Encoding”, if we pick an element from the table whose row header is 7 and column header is 3, then it is the search time for solving the BT(3, 7) problem using the original encoding. Also a ‘-’ entry in the table indicates that search took more than 1200 seconds. We experimented with the original encoding only on the BT family. The natural encoding was experimented on all three families.

As has already been remarked, the search times reported here are not competitive with the state-of-the-art conformant planning methods. Nevertheless, some conclusions can be drawn from the results obtained.

Of the three families of problems we experimented with, BT is the simplest. BTC is more complex than BT because of the fact that it has more actions and more fluents. But the most important of all differences is that length of the plan in case of BTC is roughly twice that of the corresponding BT problem. This makes a significant difference as formu-

las that encode longer histories are significantly larger. For example, with 1 toilet and 2 packages the number of atoms in the formulas for BT, BTC and BTNC are 37, 84 and 99 respectively. And the corresponding numbers of clauses are 89, 203 and 269. The BTNC family introduces non-determinism into the action domain. This could increase the search time as now the planner should consider all outcomes of an action before it can come up with the plan. Notice that valid plans for the BTC family work for BTNC and vice versa. So overall we expected BT to perform the best, BTC to be the next best and BTNC the worst. The results obtained clearly indicate that this is the case. So a general conclusion is that search times deteriorate with increase in the length of the plan and also with the introduction of non-determinism, just as one would expect.

Of the two encodings we used, the natural encoding outperformed the original encoding in every comparison conducted. As we have already mentioned, this could be because of fact that the way the quantifiers were ordered better reflected the dependencies between the atoms. This suggests that there could be a “best form” for a QBF that the QBF solver performs best on. But it is not very clear how such factors affect the search times, and such factors may in fact affect different solvers in different ways.

Since the natural encoding consistently outperformed the original encoding in the case of BT, we did not test the original encoding on the other two sets of problems.

Within a family of problems, if we keep the number of toilets fixed, the search times, in general, increase with increase in the number of the packages. The difference in search times was more significant when the increase in the number of packages resulted in an increase in the length of the plan sought. This again reinforces the observation that the length of the histories that the QBF represents has an impact on the search time. It increases the size of the formula, and also increases the number of plans that, in some sense, must be considered by the solver. This again was as expected.

On the other hand, varying the number of toilets for a fixed number of packages shows some interesting results. There seem to be two factors that affect search times in this case. First, with the introduction of more toilets there is lot of concurrency (more packages can

be dunked at the same time) and hence the plans get shorter. Also this could give the solver a lot of latitude in finding a model because the number of valid plans increases with the introduction of more toilets. On the other hand, increasing the number of toilets increases the size of the formula, and it could get more and more difficult for the solver. Our experimental results indicate that search times initially decrease with increase in toilets but eventually start deteriorating. For example the number of atoms in each of the formulas corresponding to the $BT(1, 6)$, $BT(2, 6)$, $BT(3, 6)$ and $BT(6, 6)$ is 481, 424, 417 and 446 respectively. The number of clauses in each is 1095, 954, 943, 1040 respectively. The sizes of these formulas are roughly at the same level. But the corresponding search times are 1165.799, 20.635, 5.488 and 0.010. The decrease in length of the plan appears to compensate for any effects introduced by the increase in the number of fluents and actions in the domain. Also, as the length of the plan comes down the search times improve. For $BT(12, 6)$, $BT(18, 6)$ and $BT(24, 6)$ the numbers of atoms are 1058, 1886, 2930, and the number of clauses are 2624, 4856 and 7736. Notice that here the length remains constant and only the number of toilets increase, and as a result, the size of the formulas increases with the increase in the number of fluents and actions. The search times for these problems increases as well slowly from 0.037 for $BT(12, 6)$ to 0.096 for $BT(18, 6)$ and 0.271 for $BT(24, 6)$. We observed the same kind of trend throughout our experiments. This is a good indication that while concurrency has the effect of increasing the formula sizes, it helps the search for the plan.

In general, introduction of non-determinism and increase in the length of the plan seem to affect the search times negatively and concurrency seems to have a positive effect.

Table 4.1: BT Family - Original Encoding - 1 to 8 toilets

	1	2	3	4	5	6	7	8
2	0.002	0.000	0.000	0.002	0.002	0.000	0.000	0.002
3	0.244	0.053	0.002	0.000	0.000	0.002	0.002	0.002
4	12.076	12.518	10.607	0.002	0.002	0.004	0.004	0.006
5	-	899.891	-	-	0.014	0.008	0.008	0.012
6	-	-	-	-	-	0.010	0.016	0.020
7	-	-	-	-	-	-	0.084	0.027
8	-	-	-	-	-	-	-	0.045

Table 4.2: BT Family - Original Encoding - 9 to 16 toilets

	9	10	11	12	13	14	15	16
2	0.002	0.002	0.002	0.002	0.002	0.002	0.004	0.004
3	0.002	0.004	0.006	0.004	0.004	0.008	0.008	0.010
4	0.008	0.008	0.012	0.012	0.016	0.016	0.020	0.021
5	0.014	0.016	0.018	0.020	0.027	0.031	0.035	0.041
6	0.023	0.027	0.031	0.039	0.043	0.053	0.062	0.070
7	0.033	0.041	0.051	0.062	0.070	0.084	0.100	0.117
8	0.055	0.064	0.078	0.096	0.109	0.131	0.154	0.184
9	0.541	0.096	0.115	0.143	0.162	0.199	0.271	0.314
10	-	0.137	0.166	0.211	0.238	0.287	0.363	0.434
11	-	-	2.289	0.281	0.385	0.396	0.465	0.621
12	-	-	-	0.391	0.486	0.604	0.744	0.898
13	-	-	-	-	8.211	0.844	0.926	1.229
14	-	-	-	-	-	1.162	1.543	1.812
15	-	-	-	-	-	-	24.660	2.580
16	-	-	-	-	-	-	-	3.506

Table 4.3: BT Family - Original Encoding - 17 to 24 toilets

	17	18	19	20	21	22	23	24
2	0.004	0.006	0.004	0.006	0.006	0.008	0.008	0.008
3	0.012	0.012	0.014	0.018	0.018	0.020	0.021	0.025
4	0.023	0.025	0.031	0.037	0.041	0.047	0.051	0.053
5	0.045	0.053	0.064	0.070	0.076	0.086	0.098	0.119
6	0.082	0.094	0.111	0.125	0.141	0.160	0.199	0.223
7	0.133	0.176	0.178	0.201	0.242	0.332	0.312	0.396
8	0.238	0.238	0.289	0.348	0.395	0.516	0.486	0.656
9	0.340	0.398	0.408	0.510	0.623	0.836	0.850	1.266
10	0.531	0.611	0.791	0.830	1.014	1.256	1.404	1.703
11	0.689	0.867	1.094	1.365	1.604	2.029	2.238	2.414
12	1.033	1.377	1.693	2.207	2.285	2.600	3.186	3.553
13	1.703	2.084	2.268	2.650	3.396	3.611	4.291	4.438
14	2.332	2.494	3.277	3.678	4.252	4.986	5.219	5.959
15	3.084	3.676	4.340	4.881	5.582	6.109	6.857	7.627
16	4.100	4.559	5.320	6.025	6.982	7.674	8.572	9.414
17	66.746	6.051	6.510	7.555	8.465	9.547	10.584	11.639
18	-	7.033	8.312	9.344	10.389	11.518	12.668	14.076
19	-	-	162.928	11.199	12.271	13.805	15.135	16.686
20	-	-	-	13.279	14.887	16.518	18.199	19.967
21	-	-	-	-	363.062	19.283	21.318	23.408
22	-	-	-	-	-	22.893	25.160	27.705
23	-	-	-	-	-	-	768.504	32.158
24	-	-	-	-	-	-	-	36.992

Table 4.4: BT Family - Original Encoding - 25 to 30 toilets

	25	26	27	28	29	30
2	0.010	0.010	0.010	0.012	0.014	0.016
3	0.027	0.031	0.033	0.035	0.039	0.041
4	0.062	0.070	0.074	0.086	0.096	0.100
5	0.123	0.150	0.152	0.184	0.197	0.234
6	0.238	0.268	0.320	0.383	0.430	0.498
7	0.434	0.555	0.574	0.791	0.807	0.902
8	0.789	0.922	1.076	1.242	1.414	1.570
9	1.174	1.600	1.645	1.875	2.170	2.354
10	1.871	2.225	2.441	2.770	3.115	3.529
11	2.826	3.277	3.621	4.045	4.236	4.656
12	3.881	4.395	4.752	5.268	5.850	6.383
13	5.205	5.688	6.367	6.820	7.426	8.162
14	6.580	7.262	8.033	8.822	9.568	10.283
15	8.494	9.127	10.076	10.963	12.000	12.877
16	10.400	11.355	12.418	13.480	14.730	16.006
17	12.750	13.936	15.240	16.391	18.076	19.449
18	15.240	16.779	18.344	20.021	21.750	23.395
19	18.367	20.119	22.105	23.918	26.232	28.127
20	21.908	23.816	26.221	28.441	31.119	33.461
21	25.658	28.023	30.822	33.320	36.551	39.465
22	30.158	32.912	36.000	39.383	42.822	46.021
23	35.246	38.492	42.018	45.527	49.387	53.119
24	40.572	44.449	48.705	52.406	57.039	61.670

Table 4.5: BT Family - Natural Encoding - 1 to 8 toilets

	1	2	3	4	5	6	7	8
2	0.000	0.000	0.000	0.000	0.002	0.000	0.002	0.000
3	0.012	0.004	0.002	0.002	0.000	0.002	0.002	0.002
4	0.371	0.016	0.021	0.002	0.002	0.004	0.004	0.006
5	19.834	0.389	0.070	0.072	0.016	0.006	0.008	0.010
6	1165.799	20.635	5.488	0.283	0.279	0.010	0.014	0.016
7	-	988.285	90.127	18.975	0.857	0.811	0.119	0.027
8	-	-	-	-	69.207	2.482	2.264	0.045
9	-	-	-	-	-	229.918	6.695	5.701
10	-	-	-	-	-	-	575.516	15.203

Table 4.6: BT Family - Natural Encoding - 9 to 16 toilets

	9	10	11	12	13	14	15	16
2	0.000	0.002	0.002	0.002	0.002	0.002	0.002	0.004
3	0.004	0.004	0.004	0.004	0.006	0.006	0.008	0.008
4	0.006	0.008	0.010	0.010	0.012	0.014	0.018	0.020
5	0.014	0.016	0.018	0.021	0.023	0.029	0.033	0.039
6	0.021	0.025	0.031	0.037	0.045	0.051	0.062	0.070
7	0.035	0.043	0.053	0.061	0.074	0.086	0.098	0.119
8	0.055	0.068	0.080	0.096	0.113	0.135	0.160	0.230
9	0.754	0.102	0.119	0.146	0.172	0.207	0.262	0.354
10	12.455	0.150	0.178	0.217	0.252	0.324	0.348	0.465
11	33.250	26.525	3.215	0.307	0.377	0.498	0.533	0.639
12	-	69.809	56.658	0.426	0.535	0.699	0.703	1.178
13	-	-	128.303	105.135	10.508	0.980	1.180	1.404
14	-	-	-	273.795	204.633	1.531	1.803	2.002
15	-	-	-	-	512.084	372.100	36.701	2.977
16	-	-	-	-	-	-	-	4.109

Table 4.7: BT Family - Natural Encoding - 17 to 24 toilets

	17	18	19	20	21	22	23	24
2	0.004	0.004	0.006	0.004	0.004	0.006	0.008	0.006
3	0.008	0.012	0.014	0.014	0.016	0.016	0.020	0.021
4	0.021	0.023	0.029	0.033	0.035	0.039	0.045	0.051
5	0.047	0.047	0.057	0.066	0.074	0.084	0.094	0.104
6	0.080	0.096	0.104	0.119	0.133	0.152	0.172	0.271
7	0.133	0.154	0.176	0.203	0.270	0.250	0.342	0.354
8	0.217	0.248	0.289	0.389	0.477	0.477	0.578	0.682
9	0.318	0.451	0.539	0.666	0.748	0.955	0.986	1.211
10	0.541	0.582	0.682	1.045	1.100	1.412	1.580	1.725
11	0.857	1.121	1.162	1.523	1.920	2.256	2.385	2.871
12	1.133	1.658	2.119	2.373	2.688	3.199	3.746	4.059
13	2.094	2.268	2.529	3.035	3.561	4.436	5.057	5.568
14	2.480	3.402	3.926	4.723	5.635	6.072	6.613	7.684
15	3.582	4.010	5.383	6.104	6.879	7.637	8.834	9.535
16	4.932	6.113	6.775	7.598	8.736	9.949	10.656	12.287
17	101.125	7.727	9.014	10.082	11.047	12.303	13.740	15.119
18	-	9.537	10.947	12.348	13.861	15.307	16.727	18.596
19	-	-	241.607	14.971	16.383	18.652	20.441	22.441
20	-	-	-	18.111	20.078	22.330	24.471	27.096
21	-	-	-	-	519.338	26.496	29.131	32.264
22	-	-	-	-	-	31.471	34.576	37.871
23	-	-	-	-	-	-	1062.760	44.662
24	-	-	-	-	-	-	-	52.057

Table 4.8: BT Family - Natural Encoding - 25 to 32 toilets

	25	26	27	28	29	30	31	32
2	0.008	0.008	0.008	0.010	0.012	0.014	0.014	0.016
3	0.023	0.025	0.027	0.031	0.033	0.037	0.039	0.051
4	0.057	0.062	0.070	0.076	0.096	0.098	0.098	0.113
5	0.117	0.135	0.143	0.178	0.197	0.262	0.254	0.311
6	0.207	0.309	0.359	0.430	0.416	0.471	0.521	0.570
7	0.469	0.551	0.602	0.805	0.855	1.018	1.111	1.244
8	0.812	0.996	1.143	1.305	1.420	1.664	1.936	2.219
9	1.486	1.723	1.863	2.133	2.473	2.766	3.166	3.268
10	2.383	2.613	2.867	3.377	3.781	4.000	4.496	4.834
11	3.402	3.771	4.211	4.770	5.223	5.641	6.152	6.756
12	4.594	5.473	5.750	6.377	7.104	7.732	8.225	9.010
13	6.416	7.146	7.865	8.498	9.119	10.041	10.957	11.721
14	8.275	9.270	10.109	11.090	11.977	12.914	14.078	15.043
15	10.695	11.877	12.801	13.965	15.188	16.529	17.738	19.086
16	13.393	14.727	15.992	17.582	18.934	20.512	22.102	23.881
17	16.623	18.156	19.734	21.479	23.102	25.098	27.289	29.297
18	20.193	22.191	24.162	26.359	28.352	30.668	33.125	35.746
19	24.541	27.049	29.195	31.863	34.344	37.283	40.012	43.027
20	29.469	32.197	34.945	37.928	41.125	44.303	47.998	51.457
21	35.084	38.195	41.578	45.256	48.781	52.617	56.787	61.070
22	41.359	45.111	48.994	53.279	57.547	61.975	66.801	71.834
23	48.863	53.033	57.600	62.420	67.355	72.830	78.426	84.488
24	56.844	61.768	67.160	72.656	78.477	84.740	91.244	98.092

Table 4.9: BTC Family - Natural Encoding - 1 to 8 toilets

	1	2	3	4	5	6	7	8
2	0.012	0.002	0.008	0.029	0.092	0.279	0.832	2.451
3	0.496	1.414	0.035	0.072	0.283	0.924	2.830	8.523
4	482.996	5.930	140.090	0.361	1.094	4.379	14.059	41.537
5	-	-	360.646	-	7.389	10.287	39.609	123.939
6	-	-	846.211	-	-	69.861	90.775	350.869
7	-	-	-	-	-	-	-	793.623

Table 4.10: BTC Family - Natural Encoding - 9 to 14 toilets

	9	10	11	12	13	14
2	6.949	19.492	52.406	140.475	369.934	991.033
3	24.416	72.357	203.047	550.285	-	-
4	123.504	373.936	1107.650	-	-	-
5	409.857	1188.344	-	-	-	-
6	1088.637	-	-	-	-	-
7	-	-	-	-	-	-

Table 4.11: BTNC Family - Natural Encoding - 1 to 8 toilets

	1	2	3	4	5	6	7	8
2	0.014	0.004	0.014	0.043	0.141	0.436	1.314	3.854
3	2.742	4.135	0.160	0.453	1.365	3.768	12.311	37.105
4	-	14.418	871.611	1.246	4.711	14.773	44.412	135.164
5	-	-	-	-	29.164	50.293	151.256	395.018
6	-	-	-	-	-	360.732	457.229	-

Table 4.12: BTNC Family - Natural Encoding - 9 to 13 toilets

	9	10	11	12	13
2	10.887	30.266	83.184	223.018	636.324
3	126.207	457.551	-	-	-
4	398.094	1161.846	-	-	-
5	-	-	-	-	-
6	-	-	-	-	-

Chapter 5

Other Conformant Planning Methods

For a long time, classical planning was the only form of planning that received attention. But over the last ten years there was a lot of research on other forms of planning, such as conformant planning and conditional planning. In this section we briefly discuss some of the other conformant planning methods that appeared over the last decade.

Smith and Weld's conformant planning method, Conformant Graphplan (CGP) [13] is the first (non-probabilistic) conformant planning method that emerged. A couple of earlier conformant planning methods were reported in [13], but they were all probabilistic methods (Buridan, C-Buridan and UDTPOP), and their performance as conformant planners was extremely poor. CGP is based on a classical planning system called Graphplan (which, for a short time around 1995, was a state-of-the-art planner). Graphplan constructs a so-called plan graph, in polynomial time, that can sometimes show the impossibility of finding a plan of a given length, and also serves to restrict the search space somewhat when a valid plan of the given length exists. Within the remaining part of the search space, Graphplan's algorithm is rather slow (and of course the space is still exponential). CGP deals with uncertainty in the initial state by constructing a plan graph for each of the possible initial states, and then binding these graphs together during the plan extraction process, to find a plan that works in any of the possible initial states. CGP handles concurrent actions, as long as they do not interact at all. Smith and Weld [13] proposes a method to handle non-

deterministic actions also. However, CGP has some limitations. It uses a low-level action representation language (STRIPS), which makes representing non-determinism a cumbersome process. Also, CGP's performance degrades quickly with increasing uncertainty and plan length.

The next method to appear was Cimatti and Roveri's Conformant Model Based Planner (CMBP) [4]. CMBP is notable for its representation of the planning domain as a finite state automaton. It uses Binary Decision Diagrams to compactly represent and efficiently search the automaton. CMBP handles uncertainty both in the initial state and in the effects of actions. According to [4] CMBP performs much better than CGP in problems with long plans. Also CMBP is complete in the sense that it reports failure and terminates if there is no valid plan. An important limitation of CMBP is its inability to allow for concurrent actions.

A short while later, Bonet and Geffner came up with their method named GPT [2]. GPT constructs a "belief space" that represents the conformant planning problem and then conducts a heuristic search in the belief space to find a plan. They claim to have achieved better search times on the bomb and toilet benchmark than CMBP. GPT is not complete: it does not report failure on problems that do not have a valid plan. GPT, being a heuristic method, relies heavily on the cost of computing the heuristic and usefulness of such a heuristic. The action description language used by GPT is low-level (STRIPS-based). GPT does not allow concurrent execution of actions.

In 2000, Giunchiglia introduced \mathcal{C} -PLAN [6], which uses as a driving unit for the algorithm a satisfiability solver, which is used in two ways. First, it generates candidates for valid plans. So each candidate involves a call to the sat solver. After a candidate plan is generated, it is used in a second call to the solver to check that the plan is indeed valid. In general, the second call may involve an exponentially longer formula. The most important advantage of this system is that it uses the high-level action language \mathcal{C} . It allows for concurrency and deals with uncertainty both in the initial state and effects of actions. This method is also complete. Giunchiglia however does not report on the performance of this

general method. In a separate paper, Ferraris and Giunchiglia [5] report on a specialized version of \mathcal{C} -PLAN that achieves more acceptable performance by using a STRIPS-based input language with limited nondeterminism and limited concurrency. For this version of the system, they report performance roughly comparable to that of CGP.

In 2001, Shirley Liu introduced a new method for conformant planning in her Master's thesis [10], done under the supervision of Hudson Turner. Liu extended CCALC to handle uncertainty in the initial state (deterministic conformant planning) in a manner conceptually similar to CGP. That is, it “eliminates” the uncertainty in the initial state by creating a number of parallel worlds, one each for every possible initial state. At every step, the same actions must be executed in each of the worlds. This method is also notable for the use of the high-level action language \mathcal{C} and the application of a satisfiability solver to find a conformant plan. The experimental results reported for the bomb in the toilet problems with uncertainty only in the initial states were better than any published before. However this method does not account for non-deterministic actions.

At IJCAI 2001, Cimatti et al. published a report on their new method Heuristic Search with Symbolic Model Checking (HSCP) [1]. Their method was a combination of binary decision diagrams and a heuristic search. They claim that their search times are an improvement of up to three orders magnitude over CMBP[4], and up to five orders magnitude over GPT [2]. HSCP is complete but does not allow concurrent actions.

In 2002, Zhuo Chen, another student of Hudson Turner, introduced a method [3] which improves over Liu's method and also handles non-deterministic actions under certain conditions. For instance, it can handle the non-determinism in the BTNC problems. In comparison to Liu's method, Chen tries to avoid making complete copies of the world, and achieves an improvement by doing this. On the other hand, this approach relies on special properties of its STRIPS-based input language. This method produced much better search times than Liu's method, but could not compete with HSCP (which was also more general).

The conformant planning method presented in this thesis has the advantage of using \mathcal{C} as the action description language. It accounts for concurrency and non-determinism very

naturally. This method translates a conformant planning problem into a problem of satisfiability of a QBF. This is comparable to (propositional) satisfiability planning in the case of classical planning. This method does not perform as well as the other recent methods presented above in terms of search times. But it has the advantage that it treats conformant planning in its full generality and is also mathematically elegant. Also as QBF solvers continue to improve this method will perform better.

Chapter 6

Conclusions

Let us take a look at what we have accomplished in this thesis.

6.1 Summary

- This thesis works out some details involved in extending the satisfiability planning approach of CCALC to the case of conformant planning.
- This requires taking the propositional formulas produced by CCALC (for classical planning), and using them to construct a QBF that represents the set of all valid conformant plans (of a given length).
- It would be fairly straightforward at this point to implement an extension to CCALC that would solve conformant planning problems via a QBF solver.
- This approach is arguably more mathematically elegant than other conformant planning methods. It is also unusually general, in that it is applicable to the expressive high-level action language \mathcal{C} , which is convenient for describing action domains with concurrency and non-determinism.
- Our search times are not competitive with current state-of-the-art conformant planners. Nevertheless, the times we obtain are fairly competitive with planners from just

a few years ago. In particular, our method is not much worse than the planners CGP and \mathcal{C} -plan on the bomb in the toilet problems we experimented with.

- As has certainly proven to be the case with the satisfiability planning approach to classical planning, this method could benefit immensely from advances in the field of QBF satisfiability. As we remarked earlier, QBF satisfiability is an object of intense study for a number of researchers.

6.2 Future work

- We saw that there were quite a few complications involved in using the formulas that were produced by CCALC. If we can generate these formulas from action descriptions directly, the process of building the QBF could be significantly simpler. In the process we may be able to produce formulas that result in faster inference.
- It would be interesting to investigate how an extension of this approach might work for conditional planning, following the lead of Rintanen [12]. Note that the structure of plans would be significantly different, but the definitions of executability and sufficiency could remain quite similar.
- It would be interesting to explore state-of-the-art QBF solvers and find out if there are particular forms of QBFs that might be more suitable for them. In that case we can present our QBF in that form and maybe achieve better results. We already saw an example of this, when our “natural encoding” outperformed the equivalent “original” encoding.
- Another direction would be to investigate QBF solvers that would accept arbitrary QBFs, or at least arbitrary prenex QBFs. This way we could have a QBF that would be much more concise, because we would avoid the use of the new atoms that we introduce.

Chapter 7

Appendix

This appendix includes code for the perl program that implements the conformant planning approach described in the thesis.

```
#!/usr/bin/perl -w
# Function Name: shift_range
# Input: formula_string, shift_offset,
# range_low, range_high)
# Output : Shifted formula
sub shift_range{
    my $input = shift;
    my $shift_offset = shift;
    my $range_low = shift;
    my $range_high = shift;
    my $output = "";
    $input .= "\n";
    while ($input =~ m/.*?\n/g) {
        my @numbers = split /\s+/, $&;
        for (my $i = 0; $i <= $#numbers; $i++) {
            my $abs_value = abs($numbers[$i]);
```

```

        my $negative;
        if ($abs_value != $numbers[$i]) {
$negative = -1;
        } else {
$negative = 1;
        }
        if ($abs_value >= $range_low and $abs_value <= \
$range_high) {
$abs_value = $abs_value + $shift_offset;
        }
        my $new_num = $negative * $abs_value;
        $output .= "$new_num ";
    }
    $output .= "\n";
}
return $output;
}

```

```

# Function: shift_all
# Input : (Formula, Shift_Value)
# Output : Shifted Formula

```

```

sub shift_all_above{
    my $input = shift;
    my $shift_offset = shift;
    my $range_low = shift;
    my $output = "";
    $input .= "\n";
}

```

```

while ($input =~ m/.*?\n/g) {
    my @numbers = split /\s+/, $&;
    for (my $i = 0; $i <= $#numbers; $i++) {
        my $abs_value = abs($numbers[$i]);
        my $negative;
        if ($abs_value != $numbers[$i]) {
$negative = -1;
        } else {
$negative = 1;
        }
        if ($abs_value > $range_low) {
$abs_value = $abs_value + $shift_offset;
        }
        my $new_num = $negative * $abs_value;
        $output .= "$new_num ";

    }
    $output .= "\n";

}
return $output;
}
# Function: form_implication1
# Input: cnf formula, atom, num_clauses,
#         base_number_for_new_formula_new_atom
# Output: conservative extension
sub form_implication1{
    my $input = shift;

```

```

my $consequent_atom = shift;
my $num_clauses = shift;
my $base_number_for_new_formula = shift;
my $output = "";
$input .= "\n";
for (my $i = 1; $i <= $num_clauses; $i++) {
    $input =~ /.*\n/g;
    my $line = $&;
    my @atoms = split /\s+/, $line;
    ## Negation
    for (my $j = 0; $j <= $#atoms; $j++) {
        $atoms[$j] = 0 - $atoms[$j];
    }
    ##
    foreach my $atom ( @atoms ) {
        my $N = -($i + $base_number_for_new_formula);
        $output .= "$N $atom\n";
    }
}
for (my $i = 1; $i <= $num_clauses; $i++) {
    my $temp = ($i + $base_number_for_new_formula);
    $output .= "$temp ";
}
$output .= "$consequent_atom ";
$output .= "\n";
return $output;
}
# Function: form_implication2

```

```

# Input: (Formula, New atom)
# Output: Formula corresponding to Formula -> New atom
sub form_implication2{
    my $a = shift;
    my $b = shift;
    return (form_disjunction(-$a, $b));
}

# Function: form_disjunction
# Input : (atom, disjunction)
# Output : atom | disjunction
sub form_disjunction{

    my $a = shift;
    my $b = shift;
    my @arr = split(/\n/, $b);
    $b = "";
    foreach $element (@arr) {
        $b .= $element;
        $b .= " $a\n";
    }
    #$b =~ s/\n$//;
    return $b;
}

# Main Program

```

```

open(FILE, $ARGV[0]);
$num_clauses_in_init = 0;
while ( <FILE> ) {
    $init .= $_;
    $num_clauses_in_init ++;
}
close(FILE);

open(FILE, $ARGV[1]);
$num_clauses_in_transition = 0;
while ( <FILE> ) {
    $num_clauses_in_transition++;
    $tran .= $_;
}
close(FILE);

open(FILE, $ARGV[2]);
while ( <FILE> ) {
    $goal .= $_;
}
close(FILE);

$num_constant_atomic_formulas = $ARGV[3];
$num_fluents = $ARGV[4];
$num_actions = $ARGV[5];
$num_ccalc_new_atoms = $ARGV[6];
$plan_length = $ARGV[7];

```

```

$debug_flag = 0;
$num_atoms_in_transition = \
$num_constant_atomic_formulas
\+ 2*$num_fluents + $num_actions \
+ $num_ccalc_new_atoms;
$num_new_atoms = $num_clauses_in_transition;
$cycle_length = $num_atoms_in_transition - \
    $num_constant_atomic_formulas + \
    $num_clauses_in_transition + $num_ccalc_new_atoms;
$transition_span = $plan_length * $cycle_length + \
    $num_constant_atomic_formulas;
$shift_offset = $num_fluents + $num_ccalc_new_atoms + \
    $num_new_atoms;
$curr_name = $transition_span + $num_fluents + \
    $num_clauses_in_init + 1;
$formula = "";
$formula .= &form_implication1($init, $curr_name, \
$num_clauses_in_init, \
$transition_span + \
$num_fluents);
$range_low = $num_constant_atomic_formulas +
    $num_fluents + $num_actions + 1;
$range_high = $num_atoms_in_transition;

if ($range_low <= $range_high) {
    $curr_tran = &shift_range($tran, $shift_offset, \
        $range_low, $range_high);
} else {

```

```

    print "Exception\n";
}
$curr_tran_dash = $tran;
if ($debug_flag == 1) {
    print "Number of constant atomic formulas", \
        "= $num_constant_atomic_formulas \n";
    print "Number of fluents = $num_fluents\n";
    print "Number of actions = $num_actions \n";
    print "Number of ccalc_new_atoms = \
$num_ccalc_new_atoms \n";
    print "Plan length = $plan_length\n";
    print "Number of atoms in transition = \
$num_atoms_in_transition \n";
    print "Number of clauses in transition = \
$num_clauses_in_transition \n";
    print "Number of clauses in init = \
$num_clauses_in_init\n";
    print "Cycle length used for shifting = \
$cycle_length \n";
    print "Transition span = $transition_span\n";
    print "Shift offset = $shift_offset \n";
    print "Current name is $curr_name \n";
    print "Range low = $range_low \n";
    print "Range high = $range_high \n";
    print "Current transition is:\n";
    print "$curr_tran \n";
    print "Current transition dash is:\n";
    print "$curr_tran_dash\n";
}

```

```

}
$range_low = $num_constant_atomic_formulas;
if ($debug_flag ==1) {
    print "Formula before entering loop is\n";
    print $formula;
}
for ( $k = 0; $k < $plan_length; $k++ ) {
    $temp1 = &form_implication2($curr_name, $curr_tran_dash);
    if ($debug_flag ==1) {
        print "Formula during k = $k first is\n" ;
        print $temp1;
    }

    $formula .= $temp1;
    $new_name = $curr_name + 1;
    $new_atom_start = $k * $cycle_length + \
        $num_atoms_in_transition;
    $temp = &form_implication1( $curr_tran, $new_name, \
        $num_clauses_in_transition, \
        $new_atom_start);
    if ($debug_flag ==1) {
        print "Formula during k = $k second is\n";
        print $temp;
    }
    $temp1 = &form_implication2($curr_name, $temp);
    if ($debug_flag ==1) {
        print "Formula during k = $k third is\n" ;
        print $temp1;
    }
}

```

```

    }
    $formula .= $templ;
    $curr_name = $new_name;
    $curr_tran = &shift_all_above($curr_tran, \
$cycle_length, \
$range_low);
    $curr_tran_dash = &shift_all_above($curr_tran_dash, \
    $cycle_length, \
    $range_low);
}
$formula .= \
    form_implication2(
        $curr_name, \
        &shift_all_above($goal, \
        $transition_span-
        $num_constant_atomic_formulas, \
        $num_constant_atomic_formulas));
$total_atoms = $transition_span + $num_fluents + \
    $num_clauses_in_init + $plan_length + 1;
$formula_length = 0;
while ($formula =~ m/\n/g) {
    $formula_length++;
}
$formula =~ s/(.*)?\n/$1 0\n/g;
$formula =~ s/\n$//;
$first_existential_quantifier = "q ";
for ($i = 0; $i < $plan_length ; $i++ ) {

```

```

    $jstart = $num_constant_atomic_formulas + \
        $i * $cycle_length + $num_fluents + 1;
    $jend = $jstart + $num_actions;
    for ($j = $jstart ; $j < $jend; $j++) {
        $first_existential_quantifier .= "$j ";
    }
}

$first_existential_quantifier .= " 0\n";
$quantifiers[0] = $first_existential_quantifier;
$quantifiers[1] = "q";
for ( $i = 1; $i <= $num_constant_atomic_formulas + \
    $num_fluents; $i++ ) {
    $quantifiers[1] .= " $i";
}
$quantifiers[1] .= " 0\n";
for ( $i = 0; $i < $plan_length; $i++ ) {
    $forall = "q ";
    $exists = "q ";
    $jstart = $cycle_length * $i + \
        $num_constant_atomic_formulas + $num_fluents \
        + $num_actions;
    $jend = $jstart + $num_ccalc_new_atoms + $num_fluents;
    foreach $j( $jstart+1 .. $jend ) {
        $exists .= "$j ";
    }
    $jstart += $num_fluents + $num_ccalc_new_atoms \
        + $num_clauses_in_transition;
}

```

```

    $jend += $num_fluents + $num_ccalc_new_atoms \
        + $num_clauses_in_transition;
    foreach $j( $jstart+1 .. $jend ) {
        $forall .= "$j ";
    }
    $quantifiers[2 * $i + 2] = $exists;
    $quantifiers[2 * $i + 3] = "$forall 0\n";
}
$quantifiers[2*$plan_length + 2] = "q ";
$jstart = $transition_span + $num_fluents + 1;
$jend = $jstart + $num_clauses_in_init - 1;
foreach $j ( $jstart .. $jend ) {
    $quantifiers[2] .= "$j ";
}
$index = 4;
$jstart = $num_constant_atomic_formulas + 2 * \
    $num_fluents + $num_actions + $num_ccalc_new_atoms + 1;
$jend = $num_constant_atomic_formulas + $cycle_length \
    - $num_ccalc_new_atoms;

for ( $i = 0; $i <$plan_length; $i++ ) {
    foreach $j ( $jstart .. $jend ) {
        $quantifiers[$index] .= "$j ";
    }
    $index += 2;
    $jstart += $cycle_length;
    $jend += $cycle_length;
}

```

```

$jstart = $transition_span + $num_fluents + \
    $num_clauses_in_init + 1;
$jend = $jstart + $plan_length;
$index = 2;
foreach $j ( $jstart .. $jend ) {
    $quantifiers[$index] .= "$j 0\n";
    $index += 2;
}
$formula_header = \
"p cnf $total_atoms $formula_length\n";
$quantifier_lines = "";
foreach $j ( 0 .. $#quantifiers ) {
    $quantifier_lines .= $quantifiers[$j];
}
$final_formula = $formula_header.$quantifier_lines.$formula;
print $final_formula;

```

Bibliography

- [1] Piergiorgio Bertoli, Alessandro Cimatti, and Marco Roveri. Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI*, pages 467–472, 2001.
- [2] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. 5th International Conf. on Artificial Intelligence Planning and Scheduling*, pages 52–61, Breckenridge, Colorado, 2000. AAAI Press.
- [3] Z. Chen. Determinizing in conformant planning. Master’s thesis, University of Minnesota, Duluth, MN 55805, US, 2002.
- [4] Alessandro Cimatti and Marco Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.
- [5] P. Ferraris and E. Giunchiglia. Planning as satisfiability in nondeterministic domains. In *Proceedings of AAAI*, Austin, Texas, July 2000. AAAI Press.
- [6] E. Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency. In *Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR’00)*, 2000.
- [7] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. To appear in *Artificial Intelligence (AIJ)*, 2003.

- [8] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *IJCAR*, pages 364–369, 2001.
- [9] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of ECAI*, Vienna, Austria, 1992.
- [10] S. Liu. Deterministic conformant planning with the causal calculator. Master’s thesis, University of Minnesota, Duluth, MN 55805, US, 2001.
- [11] Drew McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–160, 1987.
- [12] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
- [13] David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference of Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 889–896, Menlo Park, California, July 1998. AAAI Press.
- [14] H. Turner. Polynomial-length planning spans the polynomial hierarchy. In *Logics in Artificial Intelligence: Proc. 8th European Conf.*, pages 111–124. Springer LNAI 2424, 2002.