LOGICAL FILTERING AND LEARNING IN PARTIALLY OBSERVABLE WORLDS

BY

DAFNA SHAHAF

B.Sc., Tel-Aviv University, 2005

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Agents often need to act intelligently under uncertainty. Two main sources of uncertainty are partial observability and unknown world dynamics: in partially observable domains, agents do not know the complete state of the world (for example, they can only observe their immediate surroundings). In partially known worlds, the agents do not know how their actions affect the world.

We present algorithms for tracking the state of the world and learning its dynamics. Our algorithms derive actions' effects and preconditions in partially observable, relational domains. They have two unique features: an expressive relational language, and an exact tractable computation. An action-schema language that we present permits learning of preconditions and effects that include implicit objects and unstated relationships between objects. For example, we can learn that replacing a blown fuse turns on all the lights whose switch is set to on.

The algorithms maintain and output a relational-logical representation of all possible action-schema models after a sequence of executed actions and partial observations. Importantly, our algorithms take polynomial time in the number of time steps and predicates. Time dependence on other domain parameters varies with the action-schema language.

Our tractability result is interesting because it contrasts sharply with intractability results for structured stochastic domains. The key to this advance lies in using *logical circuits* to represent belief states; this structure can be updated efficiently and preserves compactness of representation. We also report on a reasoning algorithm (answering propositional questions) for our circuits, which can handle questions about past time steps (*smoothing*).

We evaluate our algorithms extensively on popular AI domains. Our experiments show that the circuit structure is indeed more compact than the previous representations. Moreover, the relational structure speeds up both learning and generalization, and outperforms propositional learning methods, sometimes by orders of magnitude.

To my parents, for their absolute confidence in me.

Hopeful parents

Acknowledgments

Many people have been a part of my graduate education, as friends, teachers, and colleagues. I would like to take the opportunity and thank them all.

First and foremost, I would like to thank my thesis advisor, **Eyal Amir**, who has been nothing short of amazing. I appreciate his many useful comments on this work, but even more so, I appreciate his friendship, his support (and his confidence that I should not go to New Mexico). I know of no advisor as dedicated and committed to his students.

A special thanks goes to my boyfriend, **Rodrigo de Salvo Braz**, who made life so much more enjoyable. I thank him for everything he has done for me, especially snow fights, crunches and endless nitpicking. Trust me, it is a rare combination.

I have been very fortunate to gain many friendships at Urbana. I especially wish to thank **Yoav Sharon**, for making me smile in Kickapoo; **Yaniv Eytani**, for perspective; **Deepak Ramachandran**, for tirelessly trying to bridge culture gaps; **Alok Baikadi**, for Amish farms and giant George; **Arie Ratinov**, for flashlight stories; and my research group, for many fruitful discussions and one axe.

I also wish to thank all my friends in Israel, who miraculously managed to keep in touch with me. Special thanks to **Eyal Ron** and **Tal Galili**; I miss you, you know.

I have received support from many other people, both in UIUC and in Israel. In particular, I wish to thank **Sariel Har-Peled**, **Dan Roth**, **Steve LaValle**, **Nachum Dershowitz**, **Gal Kaminka**, and **Shay Bushinsky**, for words of wisdom any time I needed them (even in India).

Last, but not least, I thank my family for being behind me. This thesis is dedicated to you.

[It would be a long list to mention all the other friends I am indebted to. I gratefully thank all of them. If your name was not listed, rest assured that my gratitude is not less than for those listed above.]

For financial support, I thank the Department of Computer Science and the Siebel Scholars Foundation. This work was supported by a Defense Advanced Research Projects Agency (DARPA) grant HR0011-05-1-0040, a DAF Air Force Research Laboratory Award FA8750-04-2-0222 (DARPA REAL program), and the National Science Foundation CAREER award grant IIS 05-46663.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SLAF	Simultaneous Learning and Filtering
CNF	Conjunctive Normal Form
DNF	Disjunctive Normal Form
NNF	Negation Normal Form
DBN	Dynamic Bayesian Network
BDD	Binary Decision Diagram
CP	Conformant Planning
BMC	Bounded Model Checking
LTL	Linear Temporal Logic
CTL	Computation Tree Logic

List of Symbols

Chapter 1

Introduction

Soon her eye fell on a little glass box that was lying under the table: she opened it, and found in it a very small cake, on which the words "EAT ME" were beautifully marked in currants. "Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door: so either way I'll get into the garden, and I don't care which happens!"

Lewis Carroll, Alice's Adventures in Wonderland

Reasoning about dynamic systems is fundamental to many applications in Artificial Intelligence (AI). Much work in Planning, Verification, and Learning applies system models whose state changes over time. Applications use these dynamic-system models to diagnose past observations, predict future behavior, and make decisions. Example domains are active Web crawlers (that perform actions on pages), robots that explore buildings, and agents in rich virtual worlds.

In this thesis we are interested in fundamental tasks and structures in partially observable domains. In those domains the initial state of the system is not known and the state is not observed fully. We investigate and present theoretical results and algorithms for two reasoning tasks that are fundamental in such domains: *Filtering* (tracking the state of the world) and *Learning* (of domain dynamics).

Filtering *Filtering* is the task of finding the set of states possible (*belief state*) after a sequence of observations and actions, starting from an initial belief state. Agents that operate in partially observable domains can act intelligently if they can detect possible current states of the world, on the basis of the past.

Logical Filtering in large deterministic domains is important and difficult. Planning, monitoring, diagnosis, and others in partially observable deterministic domains estimate the belief state (e.g., (Biere et al., 1999; Cimatti and Roveri, 2000; Bertoli et al., 2001a; Petrick and Bacchus, 2004)) as part of performing other computations. This estimation is difficult because the number of states in a belief state is exponential in the number of propositional features defining the domain.

Several approaches were developed that represent belief states compactly in logic (e.g., BDDs (Bryant, 1992), Logical Filter, and database progression (Winslett, 1990; Lin and Reiter, 1997)) and update this representation. However, none of them guarantees compact representation, even for simple domains. (Amir and Russell, 2003), for instance,

guarantees compactness and tractability only for sequences of STRIPS actions whose preconditions are known to hold. Most importantly, the straightforward approach to Logical Filtering (deciding if a clause should be in the belief state representation of time $t+1$, based on the belief state of time t) was shown to be coNP-complete (Liberatore, 1997b).

In this thesis we show that solving the update problem in its entirety is easier (done in poly-time) than creating the new belief state piecemeal. We present *C-Filter*– the first exact, tractable Logical Filtering algorithm that can handle any deterministic domain. Importantly, both time (to compute a belief state) and space (to represent it) do not depend on the domain size. Furthermore, the number of variables in the resulting formula is at most n, the number of state features (compare this with $n \cdot t$, the number of variables used in Bounded Model Checking (Clarke et al., 2001)).

We extend *C-Filter* to NNF Circuits (no internal negation nodes), and show that similar space and time bounds hold for this more restricted representation. We further show how to reason with an output circuit, including *smoothing* (queries about the past). Our results are also useful from the perspective of representation-space complexity; they sidestep previous negative results about belief-state representation (Section 2.5.4) and intractability results for stochastic domains.

The key to our advance lies in using logical circuits to represent belief states instead of traditional formulas. We show that updating a logical circuit formula amounts to adding a few internal connectives to the original formula. We take advantage of determinism: a feature is true after an action iff the action made it true or it was already true and the action did not change that. Interestingly, our empirical examination suggests that other graphical representations (e.g., BDDs) do not maintain compact representation with such updates.

C-Filter applies to many problems that require belief-state update, such as Bounded Model Checking and planning with partial observability. The attractive nature of this approach is that we can apply standard planning techniques without fear of reaching belief states that are too large to represent.

Learning *Filtering* assumes a *known* environment. That is, the agents know the action model that governs the world. In unknown, partially observable domains the problem becomes a lot more challenging: not only do the agents need to keep track of the state of the world, but also to learn the dynamics of the world.

Learning domain dynamics is difficult in general partially observable domains. An agent must learn how its actions affect the world as the world state changes and it is unsure about the exact state before or after the action. Moreover, since the immediate effects cannot be observed, actions' effects tend to mix with each other.

Current methods are successful, but assume full observability (e.g., learning planning operators (Gil, 1994; Wang, 1995; Pasula et al., 2004) and reinforcement learning (Sutton and Barto, 1998)), or do not scale to large domains (reinforcement learning in POMDPs (Jaakkola et al., 1994; Littman, 1996; Even-Dar et al., 2005)), or approximate the problem (Wu et al., 2005).

In this thesis we present a relational-logical approach to scaling up action learning in deterministic partially observable domains. Focusing on deterministic domains and the relational approach yields a strong result. The algorithm that we present learns relational schema representations that are rich and surpass much of PDDL (Ghallab et al., 1998). Many of the benefits of the relational approach hold here, including faster convergence of learning, faster computation, and generalization from objects to classes.

Like our Filtering algorithm, our learning algorithm uses a boolean-circuit formula representation for possible transition models and world states (*transition belief states*). The learning algorithm is given a sequence of executed actions and perceived observations together with a formula representing the initial transition belief state. It updates this formula with every action and observation in the sequence in an online fashion. This update makes sure that the new formula represents exactly all the transition relations that are consistent with the actions and observations. The formula returned at the end includes all consistent models, which can be retrieved then with additional processing.

We show that updating such formulas using actions and observations takes polynomial time, is exact (it includes all consistent models and only them), and increases the formula size by at most a constant additive (without increasing the number of state variables). We do so by updating a directed acyclic graph (DAG) representation of the formula. We conclude that the overall exact learning problem is tractable, when there are no stochastic interferences; it takes time $O(t \cdot p^{k+1})$, for t time steps, p predicates, and k the maximal precondition length. Thus, this is the first tractable relational learning algorithm for partially observable relational domains.

These results are useful in deterministic domains that involve many objects, relations, and actions, e.g., Web mining, learning planning operator schemas from partially observed sequences, and exploration agents in virtual domains. In those domains, our algorithm determines how actions affect the world, and also which objects are affected by actions on other objects (e.g., associating light bulbs with their switches). The understanding developed in this work is also promising for relational structure in real-world partially observed stochastic domains. It might also help enabling reinforcement learning research to extend its reach beyond explicit or very simply structured state spaces.

1.1 How to Read this Thesis

The sections in this paper are mostly self contained. A reader that is only interested in applying some of the algorithms can go safely to their respective section.

The rest of this work is organized as follows. Chapter 2 presents the Filtering problem (tracking the state of the world) formally, our logical language and circuit representation (Section 2.3.1). It also presents an algorithm, *C-Filter* , for updating belief-states as circuits (Section 2.3.2). It also discusses query answering with circuits and the non-deterministic case.

Chapter 3 extends the problem to unknown domains. It presents $\mathcal{SLAF}$ (Simultaneous Learning and Filtering), and an efficient meta-algorithm that uses our circuit representation. To turn the meta-algorithm into an algorithm family, we present several logical languages to represent transition relations; we pay special attention to the relational case (Section 3.4.3).

We describe two applications in Chapter 4: Conformant Planning and Bounded Model Checking. Each chapter includes analysis, empirical evaluation and conclusions. Chapter 5 concludes the whole thesis.

Chapter 2

Logical Filtering

2.1 Introduction

Much work in AI applies system models whose state changes over time (e.g. Planning and Verification). Applications use these dynamic-system models to diagnose past observations, predict future behavior, and make decisions. Those applications must consider multiple possible states when the initial state of the system is not known, and when the state is not observed fully at every time step. One fundamental reasoning task in such domains is *Logical Filtering* (Amir and Russell, 2003). It is the task of finding the set of states possible (belief state) after a sequence of observations and actions, starting from an initial belief state. Filtering allows us to answer queries about the current state: could we have reached a bad state? Are we necessarily in a goal state?

Logical Filtering in large deterministic domains is important and difficult. The main computational difficulties with Filtering are 1) the time needed to update the belief state, and 2) the space required to represent it. These depend on the nature of the transition model, which describes how the environment evolves over time, the observation model, which describes the way in which the environment generates observations, and the family of representations used to denote belief states.

In this chapter we show that solving the update problem in its entirety is easier (done in poly-time) than creating the new belief state piecemeal. We present *C-Filter* – the first exact, tractable Logical Filtering algorithm that can handle any deterministic domain. Importantly, both time (to compute a belief state) and space (to represent it) do not depend on the domain size. Furthermore, the number of variables in the resulting formula is at most n, the number of state features (compare this with $n \cdot t$, the number of variables used in Bounded Model Checking (Clarke et al., 2001)).

2.1.1 Related Work

This chapter is based on (Shahaf and Amir, 2007).

Research related to filtering in logical contexts is divided into logical reasoning methods for dynamic systems and logical representation languages.

(Amir and Russell, 2003) is the closest to our approach. This paper presents classes of logical languages for which

Filtering remains compact.

(Kumar and Russell, 2006) showed efficient filtering indefinitely for actions whose effects are representable as *connected row-convex constraints* (CRC).

Early on, it was pointed out that logical filtering is easy for deterministic systems with a known initial state (Fikes et al., 1972; Lin and Reiter, 1997). Filtering in nondeterministic domains is more difficult. In particular, the related problem of temporal projection is coNP-hard when the initial state is not fully known, or when actions have nondeterministic effects (Eiter and Gottlob, 1992; Nebel and Bäckström, 1992; Liberatore, 1997b; Amir, 2002; Baral et al., 2000).

Thus, computational approaches to logical filtering enumerate states, apply standard logical reasoning, or modify logical formulae that represent states. We survey these approaches now.

The simplest approaches enumerate the world states possible in every belief state and update each of those states separately, together generating the updated belief state (Kaelbling et al., 1998; Ferraris and Giunchiglia, 2000; Cimatti and Roveri, 2000; Bertoli et al., 2001b). This has the benefit of simplicity, but it is impractical when there are too many possible worlds, e.g., when the domain includes more than a few dozens of fluents and there are more than 2^{40} possible states.

The second set of approaches list the sequence of actions and observations (Reiter, 2001; Sandewall, 1994; Lifschitz, 2000) and prove queries on the updated belief state (e.g., by SAT solving (Selman et al., 1992; Moskewicz et al., 2001)). These have the benefit that they can be used for complex systems (Clarke et al., 2001), but they are impractical when the number of time steps is too long (e.g., more than 100 actions).

The third set of approaches is closest ours, namely, updating logical formulae. These approaches update a belief state representation as a propositional (or other) formula. They do so backward in time by regression (from query about time t to a different query about time 0) (Reiter, 1991; Reiter, 2001), or progress a belief-state formula from time 0 to time t (Fagin et al., 1983; Val and Shoham, 1994; Val, 1992; Liberatore, 1997a; Lin and Reiter, 1995; Lin and Reiter, 1997).

Results on regression for deterministic systems show that a step of regression is doable in polynomial time (e.g., (Reiter, 2001)), but multiple steps take exponential time and space due to the growth in formula size with every regression step.

Progression is done with Ordered Binary Decision Diagrams (OBDDs) (Bryant, 1992; Cimatti et al., 2002a) most times. These approaches are impractical in general because the formula sizes involved explode with the number of action steps. Furthermore, they are hard to compute in general (e.g, for OBDDs the update time is sometimes exponential in the number of state variables), and little success was reported on finding tractable classes or better representations.

Further approaches approximate the belief state representation (Son and Baral, 2001; Williams and Nayak, 1996; Doucet et al., 2000; Bertsekas and Tsitsiklis, 1996; Kearns et al., 2000). These approaches are natural directions of research, and bounding or quantifying the approximation is the main difficulty with them. Without such quantification these approaches may yield unacceptable mistakes in applications. Another difficulty with current approaches is the demand for an approximation that fits the given problem (if one exists).

Filtering probabilistic systems (e.g., (Doucet et al., 2000)) is the problem of determining the posterior distribution over the current state, given all the evidence to date. *Logical filtering* is the corresponding problem in logical systems. Probabilistic filtering is more difficult, and has received much attention in control theory in the past 50 years, as described above.

Logic-based languages for representing actions and change focus on expressivity and natural specification. Examples include Action Languages (Giunchiglia and Lifschitz, 1998; Son and Baral, 2001)), *Situation Calculus* (McCarthy and Hayes, 1969; Reiter, 2001) and others (Miller and Shanahan, 1999; Thielscher, 1998; Sandewall, 1994).

In this respect, our work assumes a specification in a simple propositional language similar to (Winslett, 1990). Our choice of this language is motivated by its proximity to propositional logic (thus, easy to analyze). It is not restrictive in principle because specifications in other languages can be translated into ours. Nonetheless, our tractability results describe restrictions to this language, and these restrictions would likely reflect differently in other languages.

This chapter is structured as follows: we present the problem of Logical Filtering formally in Section 2.2. We proceed to describe our representation and algorithm in Section 2.3 and give an extended example of our method. We evaluate our algorithm analytically (Section 2.5) and experimentally (Section 2.6). Finally, we conclude.

2.2 Logical Filtering

We proceed to describe the problem of Logical Filtering (tracking the state of the world), hereby referred to as Filtering. Imagine an assembly robot that can put items together in order to construct some machine. The parts are randomly oriented, and they must be brought to goal orientations for assembly. At the beginning, the parts are located on a conveyor belt. Each part drifts until it hits a fence perpendicular to the belt, and then rotates so one of its edges is aligned against the fence (see Figure 2.1). A sensor measures that edge, providing partial information about the part's orientation (partial, because the part can have some edges of equal length, and the sensor might be noisy). The robot can then rotate a part (by a certain, discrete amount) and place it back on the belt, or pick it up and try assembling it. We now define the problem formally (borrowed from (Amir, 2005)).

Definition 2.2.1 (Deterministic Transition System) *A* transition system *is a tuple* $\langle P, S, A, R \rangle$. *$P$ is a finite set of propositional fluents, $S \subseteq Pow(P)$ is the set of world states. A state contains exactly the fluents that are true in it. A*

is a finite set of actions, and R*:* $S \times A \rightarrow S$ *is the transition function.*

Executing action a in state s results in state $R(s, a)$. R may be partial. In our example,

$$P = \{\ \textit{OnBelt(part1)}, \textit{PartOfAssembly(part1)}, \textit{Touch(part1-e1)}, \textit{Touch(part1-e2)}, \ldots\ \},$$
$$A = \{\ \textit{PickUp(part1)}, \textit{Assemble(part1)}, \textit{Rotate(part1,90)}, \ldots\ \}$$

In the sequel we assume an implicit transition system.

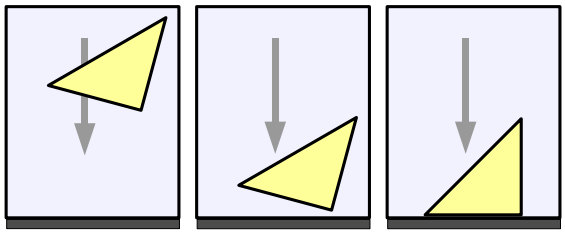


Figure 2.1: A conveyor belt: the triangle drifts down, hits the fence and rotates. The edge touching the fence is then measured.

Our robot tries to keep track of the state of the world, but it cannot observe it completely. One possible solution is to maintain a *belief state*– a set of possible world states. Every $\rho \subseteq S$ is a belief state. The robot updates its belief state as the result of performing actions and receiving observations. We now define the semantics of Filtering.

Definition 2.2.2 (Filtering Semantics (Amir and Russell, 2003)) $\rho \subseteq S$*, the states that the robot considers possible,* $a_i \in A$*. We assume that observations* o_i *are logical sentences over* P*.*

1. $\mathcal{F}\textit{ilter}[\epsilon](\rho) = \rho$ *(*ϵ*: an empty sequence)*
2. $\mathcal{F}\textit{ilter}[a](\rho) = \{s' \mid s' = R(s, a), s \in \rho\}$
3. $\mathcal{F}\textit{ilter}[o](\rho) = \{s \in \rho \mid o \textit{ is true in } s\}$
4. $\mathcal{F}\textit{ilter}[\langle a_j, o_j\rangle_{i \leq j \leq t}](\rho) =$
   $\mathcal{F}\textit{ilter}[\langle a_j, o_j\rangle_{i < j \leq t}]\ (\mathcal{F}\textit{ilter}[o_i](\mathcal{F}\textit{ilter}[a_i](\rho)))$

We call step 2 progression with a *and step 3 filtering with* o*.*

Every state s in ρ becomes $s' = R(s, a)$ after performing action a. After receiving an observation, we eliminate every state that is not consistent with it.

Figure 2.2 illustrates a belief-state update. Imagine that the robot has an isosceles right triangle (edges of size 1,1,$\sqrt{2}$), and one of the 1-edges is currently touching the fence. There are two possible orientations ((a) and (b), left part). After rotating the triangle 90 degrees, each possible state is updated (middle part). If the world state was (a), we should see a 1-edge again. Otherwise, we expect to see the $\sqrt{2}$-edge. After observing a 1-edge (right part), the robot eliminates (b) from his belief state, leaving him only with (a). That is, the robot knows the orientation of the triangle.

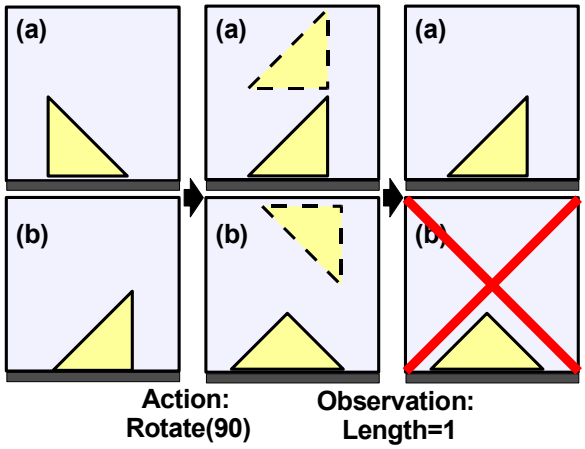


Figure 2.2: A belief-state update with a 1,1,$\sqrt{2}$ triangle. Left: Possible initial states. Middle: Progressing with *Rotate(90)*– rotating the triangle by 90° and putting it on the belt again. Right: After observing length=1, state (b) is eliminated.

2.3 Circuit Filtering

Filtering is a hard problem: there are $2^{2^{|P|}}$ belief states, so naïve methods (such as enumeration) are intractable for large domains. Following (Amir and Russell, 2003), we represent belief states in logic. Their solution provides the foundations for our research, but it guarantees an exact and compact representation only for a few classes of models (e.g. restricted action models, belief-states in a canonical form). We use logical circuits (not flat formulas) in order to extend their results to all deterministic domains. In this section we describe our representation and explain how to update it with an action-observation sequence, and how to reason with the result.

2.3.1 Representation

A belief-state ρ can be represented as a logical formula φ over some $P' \supseteq P$: a state s is in ρ iff it satisfies φ ($s \wedge \varphi$ is *satisfiable*). We call φ a *belief-state formula*. We represent our belief state formulas as circuits.

Definition 2.3.1 (Logical Circuits) Logical Circuits *are directed acyclic graphs. The leaves represent variables, and the internal nodes are assigned a logical connective. Each node represents a formula– the one that we get by applying the connective to the node's children.*

We allow the connectives $\wedge, \vee, \neg$. $\neg$ nodes should have exactly one child, while $\wedge, \vee$ can have many. In Corollary 2.5.6 we explain how to avoid internal $\neg$ nodes (for NNF).

We use logic to represent R, too: a *domain description* D is a finite set of *effect rules* of the form "a **causes** F **if** G", for a an action, F and G propositional formulas over P. W.l.g., F is a conjunction of literals. The semantics of these rules is as follows: after performing a in state s, iterate through its rules. If the rule's precondition G holds in

s, its effect F will hold in $R(s, a)$. If this leads to a contradiction, a is not possible to execute. The rest of the fluents stay the same; if no preconditions hold, the state does not change (we can also make action failure lead to a *sink state*).

Consider the triangle in Figure 2.2. If the triangle is on the belt, action $a =$ *Rotate(90)* will rotate it, so the touching edge will change: *e1* to *e2*, *e2* to *e3*, *e3* to *e1*. a's effect rules are:

"a **causes** *Touch(e2)* $\wedge \neg$*Touch(e1)* **if** *OnBelt()* $\wedge$ *Touch(e1)*"

"a **causes** *Touch(e3)* $\wedge \neg$*Touch(e2)* **if** *OnBelt()* $\wedge$ *Touch(e2)*"

"a **causes** *Touch(e1)* $\wedge \neg$*Touch(e3)* **if** *OnBelt()* $\wedge$ *Touch(e3)*"

2.3.2 *C-Filter*

We described how domains and belief-states are represented; we can now present our Circuit-Filtering algorithm, *C-Filter* .

Algorithm Overview

C-Filter is presented in Figure 2.3 and demonstrated in Section 2.4. It receives an action-observation sequence, an initial belief state formula, φ, over P, and a domain description D. It outputs the filtered belief state as a logical circuit.

The algorithm maintains a circuit data structure, and pointers to some of its nodes. A pointer is a variable that holds the address of some node; a pointer to a node represents the formula which is rooted in that node (and they will be used interchangeably). We maintain pointers to the following formulas: (1) A formula *cb* (constraint base) – the knowledge obtained so far from the sequence (receiving observations and knowing that actions were possible to execute constrains our belief state). (2) For every fluent $f \in P$, a formula $expl_f$; this formula explains why f should be true now (in Figure 2.4, the node marked *e(Tch1)* is the explanation formula of *Touch(e1)* at time t, and the root node is the explanation at time $t + 1$).

We keep the number of variables in our representation small ($|P|$) by allowing those formulas to involve *only* fluents of time 0. In a way, this is similar to regression: ***expl***$_f$ **expresses the value of fluent** f **as a function of the initial world state, and *cb* gives the constraints on the initial state.**

The belief state is always $cb \wedge \bigwedge_{f \in P}(f \leftrightarrow expl_f)$. In other words, a possible model should satisfy *cb*, and each fluent f can be replaced with the formula $expl_f$.

In the preprocessing phase, we extract data from the domain description (Procedure *C-Filter* , line 1). We then create a node for each fluent, and initialize the $expl_f$ pointers to them. We also create a circuit for the initial belief-state, φ (using the $expl_f$ nodes), and set the *cb* pointer to it (lines 2-3). Then we iterate through the sequence, update

[1] *Cause(a,f)* represents the conditions for a to cause f, extracted from the domain description (See (*))

PROCEDURE ***C-Filter***$(\langle a_i, o_i\rangle_{0<i\leq t}, \varphi, D)$
$\{a_i$ actions, o_i observations, φ an initial belief state over P,
D domain description$\}$

Preprocess D, φ

1: ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
2: **for** $f \in P$ **do** $expl_f :=$ a new proposition f_0
3: $cb :=$ ***Time0***(φ)

Process sequence

4: **for** i = 1 **to** t **do**
5: ***ProgressAction***(a_i)
6: ***FilterObservation***(o_i)
7: **return** $cb \wedge \bigwedge_{f\in P}(f \leftrightarrow expl_f)$

PROCEDURE ***ProgressAction***(a)
$\{a$ an action$\}$

Update *cb*: a executed, thus was possible.
Get f's next-value explanation:
a caused f, or f held and a did not cause $\neg f$

1: **for** $f \in$ *Effects(a)* **do**
2: $cb := cb \wedge$ ***Time0***(*Poss(a,f)*)
3: $expl'_f :=$ ***Time0***(*NextVal(a,f)*)
4: **for** $f \in$ *Effects(a)* **do** $expl_f := expl'_f$

PROCEDURE ***FilterObservation***(o)
$\{o$ an observation over $P\}$
1: $cb := cb \wedge$ ***Time0***(o)

PROCEDURE ***Time0***(ψ)
$\{\psi$ a formula$\}$

Return an equivalent circuit over time 0

1: **for** $f \in P$ in ψ **do** replace f with the node pointed by $expl_f$
2: **return** ψ

PROCEDURE ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
$\{D$ a domain description, a_i actions$\}$

Extract "Next Value" and "Possible" Formulas

1: **for** $f \in P, a \in \langle a_i\rangle$ **do**
2: *NextVal(a,f)* := *Cause(a,f)* $\vee$ ($f \wedge \neg$*Cause(a,¬f)*) [1]
3: *Poss(a,f)* := $\neg$(*Cause(a,f)* $\wedge$ *Cause(a,¬f)*)
4: (optional: Simplify Formulas)

Figure 2.3: *C-Filter* Algorithm

the circuit and the pointers with every time step (lines 4-6, see below), and finally return the updated belief state.

A Closer Look

The circuit is constructed as follows. In the preprocessing stage, we extract some useful formulas from the domain description. Let *Effects(a)* be the set of fluents that action a might affect. For each f in this set, we need to know how a can affect f. Let *Cause(a,f)* be a formula describing when a causes f to be true. It is simply the precondition of the rule of a causing f to hold (if there are several, take the disjunction; if there are none, set it to *FALSE*). That is, if $s \models$ *Cause(a,f)* and a is possible to execute, f will hold after it. *Cause(a,¬f)* is defined similarly.

For example, take $a =$ *Rotate(90)* (Section 2.3.1). Then

Effects(a)=$\{$*Touch(e1)*, *Touch(e2)*, *Touch(e3)*$\}$

Cause(a,Touch(e1)) $=$ *OnBelt()* $\wedge$ *Touch(e3)*

Cause(a,$\neg$ *Touch(e1))* $=$ *OnBelt()* $\wedge$ *Touch(e1)* (*)

Procedure *ProgressDomain* then constructs the formula *NextVal(a,f)*, which evaluates to *TRUE* iff f holds after a (given the previous state). Intuitively, either a caused it to hold, or it already held and a did not affect it. Similarly, the formula *Poss(a,f)* states that a was possible to execute regarding f, i.e. did not cause it to be true and false at the same time.

After preprocessing, we iterate through the sequence. Procedure *ProgressAction* uses those formulas to update the belief state: First, it constructs a circuit asserting the action was possible (corresponding to the *Poss* formula) and adds it to *cb* (line 2). Then, it builds a circuit for the *NextVal* formula. Procedure *Time0* ensures the circuit uses only time-0 fluents. When we construct a new *Poss* or *NextVal* circuit, its leafs represent fluents of the previous time step; *Time0* replaces them by their *equivalent* explanation nodes. Our circuit implementation is crucial for the efficiency of this replacement. Instead of copying the whole formula, we only need to update edges in the graph (using the pointers). This way, we can share formulas recursively, and maintain compactness.

After all the new circuits were built, the explanation pointers are updated (line 4); the new explanation is the root of the corresponding *NextVal* circuit, built earlier (line 3; see also Section 2.5.1). Then we deal with the observation (Procedure *FilterObservation*): similarly, we use *Time0* to get a time-0 formula, and simply add it to *cb*.

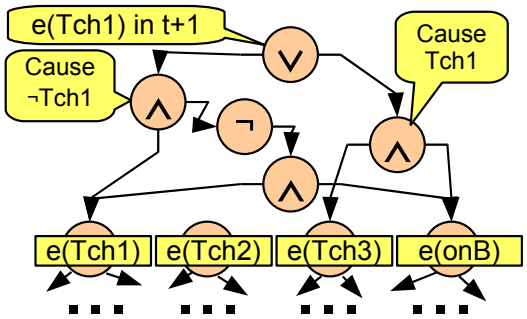


Figure 2.4: Updating the explanation of *Touch(e1)* after *Rotate(90)*

Example: Figure 2.4 shows an update of the explanation of *Touch(e1)* after the action *Rotate(90)*. Rectangles (on the bottom nodes) represent the explanation pointers of time t (before the action). The circuit in the image is the *NextVal* formula, after Procedure *Time0* replaced its fluents by the corresponding explanation nodes.

The $\vee$ node is the root of the graph representing state of *Touch(e1)* after the action: the right branch describes the case that the action caused it to hold, and the left branch is the case that it held, and the action did not falsify it. In the next iteration, the pointer of *Touch(e1)* will point at this node.

Note the re-use of some time-t explanation nodes; they are internal nodes, possibly representing large subformulas.

2.3.3 Query Answering with a Belief-State Formula

C-Filter returns an updated belief state φ^t, represented as a logical circuit. We are interested in satisfiability queries ($\varphi^t \wedge \psi$ satisfiable) and entailment queries ($\varphi^t \models \psi$, or $\varphi^t \wedge \neg\psi$ unsatisfiable). In the following, we construct a circuit corresponding to the query and run inference on it.

Query Circuits

Let ψ be an arbitrary propositional query formula; we want to check whether $\varphi^t \wedge \psi$ is satisfiable. Very similarly to an observation, we add ψ to *cb*, and replace the fluents for their explanations. The new *cb* is our query circuit. Queries are usually about time t, but the circuit structure allows more interesting queries, in particular *smoothing*– queries that refer to the past (e.g., did f change its value in the last 5 steps? Could the initial value of g be TRUE?). Note that every fluent in every time step has a corresponding node. If we keep track of those nodes, we can replace fluents from any time step by their explanations. If the queries are given in advance, this does not change the complexity. Otherwise, finding a past-explanation node might take $O(\log t)$ time. Note that the same mechanism (tracking previous explanations) has many interesting applications, such as filtering in non-Markovian domains.

Using the model: Satisfiability for Circuits

Our algorithm computes a solution to *Filter* as a logical formula; we can now use a SAT solver in order to answer queries about the world state. Most current SAT solvers require CNF encoding, so we may consider conversion to CNF. Conversion by flattening down the circuit is often exponential in the circuit size, thus impractical. Assigning a variable to each internal node will not result in a blow-up, but it suffers from several drawbacks: for instance, the number of variables is no longer fixed, but grows with k. Also, the structural information of the circuit is lost. We explain those in detail and describe the approach that we take.

First we consider the number of variables in the formula. SAT is exponential in this number, so adding variables should be handled with care. Maintain the size of the state space by preventing the DPLL search from branching on any internal-node variable was shown to be non-robust and to entail a reduction in the power of the proof system it implements (Järvisalo et al., 2004).

Furthermore, several works show that structural information can be used to improve performance of SAT solvers significantly. For example, the Lsat solver (Ostrowski et al., 2002) and NoClause (Thiffault et al., 2004a) extracts and exploit extensive structural information, solving a number of important benchmarks easily.

Finally, every clausal DPLL search procedure is inherently limited by the power of resolution. Circuit DPLL solvers, on the other hand, can employ complex gates and special purpose propagators, that in certain cases can circumvent resolution limitations. Since our problem is already encoded in a rich non-clausal structure, it seems that

Circuit SAT solvers could show a great promise in our case.

We implemented our own circuit SAT solver, *C-DPLL* . It is a generalization of DPLL: every iteration, an uninstantiated variable f is chosen, and set to *TRUE*. The truth value is then propagated as far as possible, resulting in a smaller circuit (for example, if f had an OR parent, it will be set to *TRUE* as well). Then, *C-DPLL* is called recursively. If no satisfying assignment was found, it backtracks and tries *f=FALSE*. If no assignment is found again, return UNSAT. *C-DPLL* takes $O(|E| \cdot 2^l)$ time and $O(|E|)$ space for a circuit with $|E|$ edges and l leaves. In addition, we use off-the-shelf circuit SAT solvers, such as NoClause (Thiffault et al., 2004b).

Note that the number of variables in the formula is independent of the length of the sequence. Therefore, we can use SAT solvers for very long sequences. We can also use bias (McCarthy, 1986) to find minimal models. Preferential bias is well studied and fits easily with logical formula.

2.4 Extended Example

We now give a detailed example of the whole process. Interestingly, this example demonstrates how logical circuits can represent compactly a belief state that one *cannot* represent compactly using CNF formulas over the same variables.

Our domain includes fluents $\{p_1, ..., p_n, odd\}$. The following sequence of actions makes the fluent *odd* equal to $p_1 \oplus p_2 \oplus ...p_n$, the parity of the other fluents. Our actions $a_1, ..., a_{n-1}$ are defined such that a_1 sets $odd := p_1 \oplus p_2$, and any other a_i sets $odd := odd \oplus p_{i+1}$. Formally:

"a_1 **causes** *odd* **if** $(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)$"

"a_1 **causes** $\neg odd$ **if** $\neg[(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)]$"

"a_i **causes** *odd* **if** $(odd \wedge \neg p_{i+1}) \vee (\neg odd \wedge p_{i+1})$"

"a_i **causes** $\neg odd$ **if** $\neg[(odd \wedge \neg p_{i+1}) \vee (\neg odd \wedge p_{i+1})]$"

Applying the sequence $a_1, ..., a_{n-1}$ sets $odd = p_1 \oplus ...p_n$. We now show how our algorithm maintains the belief state throughout the sequence.

Preprocessing the Domain:

In this phase we extract the *Poss* and *NextVal* formulas. We examine the action specifications: the only fluent which is affected is *odd*. a_1 is executable when it does not cause both *odd*, $\neg odd$.

Cause(a_1,*odd*) $= (p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)$

Cause(a_1,$\neg$*odd*) $= \neg[(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)]$

Poss(a_1,*odd*) $= \neg[$*Cause*(a_1,*odd*) $\wedge$ *Cause*(a_1,$\neg$*odd*)$]$

It is easy to see that both cannot hold simultaneously, and the formula can be simplified to *TRUE*: indeed, a_1 is always executable. Similarly, all of our actions are always possible to execute, so the *Poss* formulas are all equal *TRUE*.

Now, the *NextVal* formulas. After executing a_1, *odd* will be set to *Cause(*a_1*,odd)* $\vee$ [*odd* $\wedge \neg$*Cause(*a_1*,*$\neg$*odd)*].

This is equivalent to *Cause(*a_1*,odd)*. In other words, *odd* will be set to $p_1 \oplus p_2$. Similarly, after action a_i *odd* will be set to $p_{i+1} \oplus$ *odd*. Note, simplifying the formulas is not mandatory; the representation will be compact without it, too.

Executing the Actions:

Imagine that we receive the (arbitrary) sequence $a_1, a_2, ..., a_{n-1}$, *odd* $\wedge \neg p_n$ (performing n actions and receiving an observation). Figure 2.5 describes how the algorithm updates the belief-state with this sequence. At time 0 (2.5a) we create a node for every fluent, and another for *TRUE*. The nodes represent the value of the fluent at time 0. We set a pointer (the rectangles) for each formula that we want to maintain: the formula for *cb* (constraints) is set to *TRUE* because we do not have any initial knowledge. The explanation formula of each fluent is set to the corresponding node.

We then execute a_1, arriving at time 1 (2.5b). No constraint was added to *cb*, since the action is always executable. No explanation formula of p_i changed, since a_1 does not affect them. The only thing that changed is the state of *odd*: its new explanation is $p_1 \oplus p_2$. We construct the graph for this formula, and update the explanation pointer to its root node.

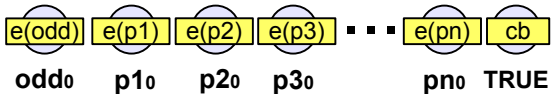
NOTE: the image shows xor gates just for the sake of clarity. In fact, each of them should be replaced by five gates, as depicted in 2.5b.

Executing a_2 is similar (time 2, 2.5c). We construct the graph for *odd*'s new value, *odd* $\oplus\ p_3$. Note that we substitute the fluents in this formula (*odd*, p_3) by their explanations in time 1, i.e. the pointers of the previous time step.

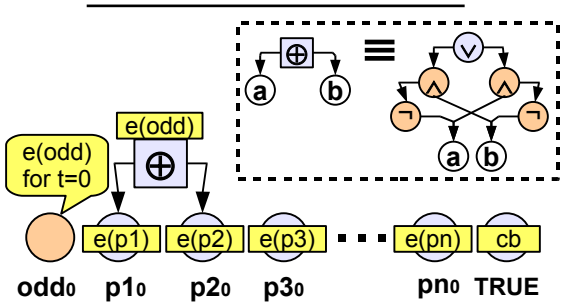
We execute $a_3, ..., a_{n-1}$, and then observe *odd* $\wedge \neg p_n$ (2.5d. This is just an example observation; alternatively, you can think of it as querying whether it is possible that *odd* $\wedge \neg p_n$ holds now). First, we process the actions and update the explanation of *odd*. Then we add *odd* $\wedge \neg p_n$ to our constraints, creating a new *cb* circuit and updating the pointer. Finally, we return the circuit in 2.5d, along with the pointers. This is our updated belief state.

Answering Queries:

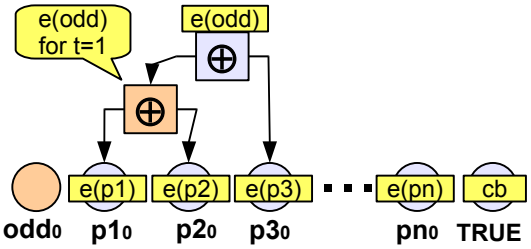
In 2.5e we show an example of truth-value propagation: if we assume that at time 0 p_1=*TRUE* and the rest are set to *FALSE*, those values are propagated up and result in *cb*=*TRUE*. That is, this assignment is consistent with our



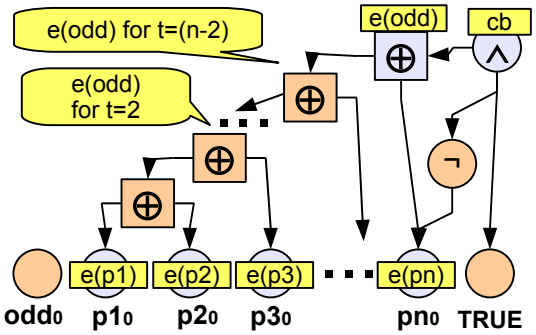
(a) At time t=0: initial belief state $\varphi = TRUE$



(b) Time t=1: after performing a_1



(c) Time t=2: after performing a_1, a_2



(d) Time t=(n-1): after performing $a_1, .., a_{n-1}$ and observing $(odd \wedge \neg p_n)$

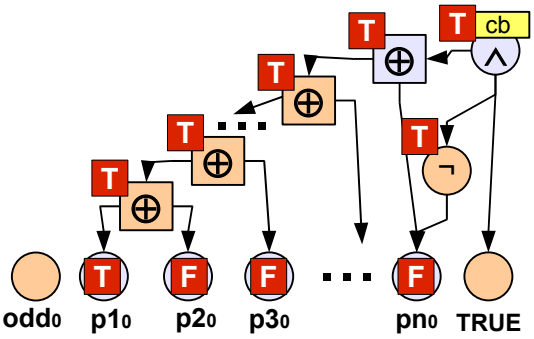


Figure 2.5: (e) Propagating truth-values

sequence.

2.5 Analysis and Complexity

2.5.1 Correctness

Theorem 2.5.1 *C-Filter is correct. For any formula φ and a sequence of actions and observations $\langle a_i, o_i\rangle_{0<i\leq t}$,*

$$\{\, s \in S \;\; \textit{that satisfy C-Filter } (\langle a_i, o_i\rangle_{0<i\leq t}, \varphi)\} =$$
$$\mathit{Filter}[\langle a_i, o_i\rangle_{0<i\leq t}](\{s \in S \textit{ that satisfy } \varphi\}).$$

Recall that a state s satisfies formula φ if $s \wedge \varphi$ is *satisfiable* (Section 2.3.1). s is used as a formula and as a state.

PROOF SKETCH We present an effect model, and show how to update a belief-state (flat) formula with this model. We show that the Filtering definition in Section 2.2 can be reduced to consequence finding (in a restricted language) with this formula. Then, we show that *C-Filter* computes exactly those consequences.

Definition 2.5.2 ***Effect Model**:*

*For an action a, define the **effect model** of a at time t to be:*

$T_{\mathit{eff}}(a,t) = a_t \rightarrow$

$\quad \bigwedge_{f\in P} \mathit{Poss}(a,f,t) \;\wedge\; (f_{t+1} \leftrightarrow \mathit{NextVal}(a,f,t))$

$\mathit{Poss}(a,f,t) = \neg(\mathit{Cause(a,f)}_t \wedge \mathit{Cause(a,\neg f)}_t)$

$\mathit{NextVal}(a,f,t) = \mathit{Cause(a,f)}_t \vee (f_t \wedge \neg \mathit{Cause(a,\neg f)}_t)$

a_t asserts that action a occurred at time t, and f_{t+1} means that f after performing a. ψ_t is the result of adding a subscript $_t$ to every fluent in formula ψ (see Section 2.3.2 for definition of *Cause(a,f)*). The effect model corresponds to effect axioms and explanation closure axioms from Situation Calculus (Reiter, 1991). If the robot can recognize an impossible action, we can drop the assumption that actions are possible, and adopt a slightly different effect model.

Recall that $\mathit{Filter}[](\cdot)$ was defined over a set of states. We now define its analogue *L-Filter*, which handles belief-state logical formulas.

Definition 2.5.3 (L-Filter) *Let φ a belief-state formula.*

- *L-Filter*$[a](\varphi) = Cn^{L_{t+1}}(\varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t))$
- *L-Filter*$[o](\varphi) = \varphi \wedge o$

where $Cn^L(\psi)$ are the consequences of ψ in vocabulary L. $L_{t+1} = (L(\varphi_t) \cup P_{t+1}) \setminus P_t$, for $P_t = \{f_t \mid f \in P\}$ and $L(\varphi_t)$ the language of φ_t; i.e., L_{t+1} does not allow fluents with subscript t.

Lemma 2.5.4 *The result of applying L-Filter*$[a]$ *for* $a \in A$ *is a formula representing exactly the set of states* $\mathcal{F}ilter[a]$*. More formally, let* φ *be a belief state formula.*

$$\mathcal{F}ilter[a](\{s \in S \mid s \textit{ satisfies } \varphi\}) =$$
$$\{s \in S \mid s \textit{ satisfies L-Filter}[a](\varphi)\}$$

PROOF SKETCH:

We show that the two sets of world states have the same elements. We show first that the left-hand side of the equality is contained in the right-hand side.

Take $s' \in \mathcal{F}ilter[a](\{s \in S \mid s \text{ satisfies } \varphi\})$. We show that s' satisfies *L-Filter*$[a](\varphi)$. From the Filtering definition there is $s \in S$ such that $s \in \{s \in S \mid s \text{ satisfies } \varphi\}$ and $R(s,a) = s'$. In other words, there is $s \in S$ such that s satisfies φ and $R(s,a) = s'$.

To prove that s' satisfies *L-Filter*$[a](\varphi)$ we need to show that $\varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t)$ together with the truth assignment s' to time $t+1$ is satisfiable. We show that the truth assignment s to time t satisfies this formula together with the truth assignment s' to time $t+1$ and $a_t = \mathit{TRUE}$. It is not satisfying this formula only if one of the conjuncts $\mathit{Poss}(a,f,t) \wedge (f_{t+1} \leftrightarrow \mathit{NextVal}(a,f,t))$ or φ is falsified. This cannot be the case for φ by our choice of s.

Assume by contradiction that this is the case for some conjunct. If it is the case that $\neg \mathit{Poss}(a,f,t)$, then $(\mathit{Cause(a,f)}_t \wedge \mathit{Cause(a,\neg f)}_t)$. In other words, s satisfies *Cause(a,f)* and *Cause(a,¬f)*. According to our domain-description semantics, $R(s,a)$ should satisfy f and $\neg f$. This cannot be the case, since we only address executable actions.

If it is the case that $\neg f_{t+1} \wedge \mathit{NextVal}(a,f,t))$, then $\neg f_{t+1} \wedge \mathit{Cause(a,f)}_t \vee (f_t \wedge \neg \mathit{Cause(a,\neg f)}_t)$. That is, s satisfies *Cause(a,f)*, or it satisfies f and $\neg$*Cause(a,¬f)*. According to our domain-description semantics, s' has to satisfy f. Therefore, this cannot be the case as well.

Thus, there is no such conjunct and the truth assignment s, s' satisfies this formula. From the definition of *L-Filter*$[a](\varphi)$ and Craig's interpolation theorem for propositional logic we get that s' satisfies *L-Filter*$[a](\varphi)$.

For the opposite direction (showing the right-hand side is contained in the left-hand side), take $s' \in S$ that satisfies *L-Filter*$[a](\varphi)$. We show that $s' \in \mathcal{F}ilter[a](\{s \in S \mid s \text{ satisfies } \varphi\})$. From Craig's interpolation theorem for propositional logic we get that there is a truth assignment s for $\mathcal{P}$ such that the truth assignment s, s' for times $t, t+1$, respectively, together satisfy $\varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t)$ (otherwise, there is no such truth assignment, and *L-Filter*$[a](\varphi)$ is not satisfiable; in particular, s' does not satisfy it). In a manner similar to the first part of this proof (observing the way $\mathfrak{R}$ is defined) we can show that $\mathfrak{R}(s,a) = s'$ and the second part is done. ■

That is, both definitions are equivalent. As a result, we can compute $\mathcal{F}ilter$ using a consequence finder in a restricted language. However, this does not guarantee tractability. Instead of using a consequence-finder, we show that *C-Filter*

computes exactly those consequences.

Let $\psi := \varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t)$. According to our definition, *L-Filter*$[a](\varphi) = Cn^{L_{t+1}}(\psi)$. We now observe that consequence finding is easy if we keep φ_t in the following form:

$$\varphi_t = \mathit{cb} \wedge \textstyle\bigwedge_{f \in P}(f_t \leftrightarrow \mathit{expl}_f)$$

[s.t. *cb* and expl_f do not involve any fluent of time t]

We now show how to compute the consequences of such formulas. Furthermore, we show that the resulting formula maintains this form, so we only need to check the form of the initial belief-state. Luckily, this is not a problem; every initial belief-state can be converted to this form (in linear time) using new proposition symbols.

Let ψ be a formula in this form. ψ states that $(f_t \leftrightarrow \mathit{expl}_f)$ for every $f_t \in P_t$: we construct an equivalent formula, ψ', by replacing every $f_t \in P_t$ in $T_{\mathit{eff}}(a,t)$ with the formula expl_f.
Notation: $\psi' := \varphi_t \ \wedge a_t \ \wedge \ T_{\mathit{eff}}(a,t)[\mathit{expl}_f / f_t]$.

$$\psi \equiv \psi' \Rightarrow Cn^{L_{t+1}}(\psi) \equiv Cn^{L_{t+1}}(\psi')$$

Therefore, we can find the consequences of ψ' instead. Note that consequence finding in L_{t+1} is the same as using the Resolution algorithm to resolve fluents of P_t. We use this to compute $Cn^{L_{t+1}}(\psi')$:

$$Cn^{L_{t+1}}(\psi') \equiv \ \mathit{cb} \wedge \textstyle\bigwedge_{f \in P} (\mathit{Poss}(a,t,f)[\mathit{expl}_g / g_t]) \ \wedge$$
$$\textstyle\bigwedge_{f \in P}(f_{t+1} \leftrightarrow \mathit{NextVal}(a,t,f)[\mathit{expl}_g / g_t])$$

Let

$$\mathit{cb}' := \mathit{cb} \wedge \textstyle\bigwedge_{f \in P}(\mathit{Poss}(a,t,f)[\mathit{expl}_g / g_t])$$
$$\mathit{expl}'_f := \mathit{NextVal}(a,t,f)[\mathit{expl}_g / g_t].$$

The last formula can be re-written as

$$\mathit{cb}' \wedge \textstyle\bigwedge_{f \in P}(f_{t+1} \leftrightarrow \mathit{expl}'_f)$$

Now note that *C-Filter* maintains the belief-state formula exactly in that easy-to-compute form, namely $\mathit{cb} \wedge \bigwedge_{f \in P}(f_t \leftrightarrow \mathit{expl}_f)$, *cb* and expl_f involve only special propositions, representing time-0 (to avoid confusion, you might think of the new propositions in line 2 as f_{init}, not f_0).

Also, $\mathit{cb}', \mathit{expl}'_f$ are exactly the constraint-base and explanation formulas after *C-Filter* 's *ProgressAction*. That is, *C-Filter* correctly progresses the belief-state with actions. The proof for handling observations is similar. ■

2.5.2 *C-Filter* Complexity

Let φ^0 be the initial belief state, t the length of the action-observation sequence, and $|Obs|$ the total length of the observations in the sequence. Let *ActDesc* be the longest description of an action $a \in A$ (preconditions + effects).

Theorem 2.5.5 *Allowing preprocessing of $O(|P|)$ time and space (or, alternatively, using a hash table), C-Filter takes time $O(|\varphi^0| + |Obs| + t \cdot ActDesc)$. Its output is a circuit of the same size.*

If there are no preconditions, we can maintain a flat formula instead. If the observations are always conjunctions of literals, we can drop $|Obs|$ from the space complexity. In this case, we can move the pointers to the *TRUE* and *FALSE* nodes, resulting in shorter paths in the graph. Note that this *does not* depend on the domain size, $|P|$. *ActDesc* is usually small– especially if the actions in the domain affect a small number of fluents, and have simple preconditions.

PROOF SKETCH: Initializing *cb* takes $O(|\varphi^0|)$ time. Handling each action adds at most $O(ActDesc)$ nodes and edges to the graph, and takes the same time: in the worst case, assuming no simplifications were done, we need to construct a graph for each of the action's *Causes* formulas. Finally, each time we receive an observation o we add at most $O(|o|)$ nodes and edges, resulting in total $O(|Obs|)$.

Corollary 2.5.6 *We can maintain an NNF-Circuit (no negation nodes) with the same complexity.*

PROOF SKETCH The circuit's leaves represent *literals* (instead of propositions). We maintain explanation formulas for them. Since $(f \leftrightarrow expl_f) \Rightarrow (\neg f \leftrightarrow \neg expl_f)$, we can define $expl_{\neg f} := \neg expl_f$. We take the NNF-form of every formula we use (observations, explanations, etc.), and replace every literal by its explanation. Converting to NNF takes time linear in the formula's size; therefore, time and space complexities will not change (modulo a small constant). ■

The NNF algorithm is very similar to *C-Filter* . The main differences appear in Figure 2.6: formulas are converted to NNF form, there are pointers to positive and negative literals, and ***Time0*** takes care of both cases.

2.5.3 Projection

Projection is the problem of checking that a property holds after t action steps of an action system, starting from a belief state. Generalizations allow additional observations.

Our results from previous sections show that projection is doable by applying *C-Filter* (generating the belief state at time t, φ^t, adding the query and running our *C-DPLL* solver).

Let m be maximal length of a single observation plus *ActDesc*. Usually $m = O(1)$. Since φ^t includes $O(n)$ variables, and has overall size $O(n + mt)$, checking if a variable assignment is a model of φ^t takes time $O(n + mt)$. Thus, testing satisfiability is NP-Complete in n, instead of $n + mt$ (the size of the formula) or $n \cdot t$ (the number of

PROCEDURE ***NNF-C-Filter***$(\langle a_i, o_i\rangle_{0<i\leq t}, \varphi, D)$
$\{a_i$ actions, o_i observations, φ an initial belief state over P,
D domain description$\}$

Preprocess D, φ

1: ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
2: **for** $f \in P$ **do**
3: $\quad$ *expl*$_f$:= a new proposition f_0
4: $\quad$ *expl*$_{\neg f}$:= $\neg f_0$
5: *cb* := ***Time0***(*NNF(*φ*)* **)**

Process sequence

6: **for** i = 1 **to** t **do**
7: $\quad$ ***ProgressAction***(a_i)
8: $\quad$ ***FilterObservation***(o_i)
9: **return** $cb \wedge \bigwedge_{f\in P}(f \leftrightarrow expl_f)$

PROCEDURE ***FilterObservation***(o)
$\{o$ an observation over $P\}$
1: *cb* := *cb* $\wedge$ ***Time0***(*NNF(o)*)

PROCEDURE ***Time0***(ψ)
$\{\psi$ a formula$\}$

Return an equivalent circuit over time 0

1: **for** $f \in P$ in ψ **do**
2: $\quad$ replace $\neg f$ with the node pointed by *expl*$_{\neg f}$
3: $\quad$ replace f with the node pointed by *expl*$_f$
4: **return** ψ

PROCEDURE ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
$\{D$ a domain description, a_i actions$\}$

Extract "Next Value" and "Possible" Formulas

1: **for** $f \in P, a \in \langle a_i\rangle$ **do**
2: $\quad$ *NextVal(a,f)* := *NNF(Cause(a,f)* $\vee$ ($f \wedge \neg$*Cause(a,*$\neg$*f))*)
3: $\quad$ *Poss(a,f)* := *NNF(* $\neg$(*Cause(a,f)* $\wedge$ *Cause(a,*$\neg$*f))*)

Figure 2.6: *NNF-C-Filter* Algorithm

propositional variables that appear in an unrolling of the system over t steps; used in Bounded Model Checking). We need to guess n variable assignments, and then apply a linear algorithm to check it. The following result refines earlier complexity results.

Theorem 2.5.7 (Projection) *Let D be a domain with deterministic actions. The problem of answering whether all the states in $Filter[\pi](\varphi)$ satisfy Q, for belief state formula φ, sequence of actions π and query Q, is coNP-complete in the number of state variables, n. We assume π, φ, ActDesc and Q are polynomial in n.*

2.5.4 Representation Complexity

Our results have implications for the theory of representation-space complexity. A simple combinatorial argument shows that there are belief states of n fluents that cannot be described using logical circuit of size $o(2^n)$, i.e., strictly smaller than some linear function of 2^n. Nevertheless, our results for *C-Filter* show that those belief states that are reachable within a polynomial number of actions from an initial belief state are of size linear in t and the input size

(initial belief state size, and longest action description).

Also, (Amir and Russell, 2003) showed that for every general-purpose representation of belief states there is a (possibly nondeterministic) domain, an initial belief state, and a sequence of actions after which our belief state representation is exponential in the initial belief state size. Our results show that this does not hold for deterministic systems.

2.5.5 The Non-Deterministic Case

C-Filter handles deterministic domains only. However, many real-life environments are inherently non-deterministic. We now present two ways to handle this:

Converting into Deterministic Domains

Consider flipping a coin; given enough relevant parameters (weight, height, velocity, angle) we could predict the coin's outcome. Similarly, we can treat each non-deterministic action as a deterministic one which depends on a set of parameters unknown to us. More formally, the action of coin-flipping will have this effect model:

$$\textit{flip}_t \rightarrow [\textit{ExactlyOneCase} \wedge (\textit{case}_1 \rightarrow \textit{heads}_{t+1}) \wedge (\textit{case}_2 \rightarrow \neg \textit{heads}_{t+1})]$$

Where *ExactlyOneCase* is the logical formula specifying that exactly one of $\textit{case}_1, \textit{case}_2$ holds. Of course, we could use a binary encoding, so for n cases we would have $\log n$ *case* variables.

The main problem with this method is that the number of propositions grows linearly with time; each non-deterministic action adds new *case* variables to the formula.

Different Sorts of Circuits

We can maintain circuits for some special cases of non-deterministic domains. For example, note that even non-deterministic actions might have a deterministic inverse. For example, if a robotic arm placed a block randomly on the table, we always know the previous state: the block was held by the arm. That is, we look for transition relations in which for every pair s', a there is at most one s such that $R(\langle s, a, s' \rangle)$.

Using similar methods, we devise *PrevVal* formulas (state i as a function of state *i+1*). We build the circuit "upside-down", starting from G and moving back towards I. The leaves represent the final state (no other leaves are added).

Another case we can handle is if (in addition to the deterministic actions) we have actions of the kind "a **causes** $p \vee q$ **if** r". We further assume that r never appears in a non-deterministic effect. In this case, we can maintain a circuit belief-state in time polynomial in $|P|$ and t.

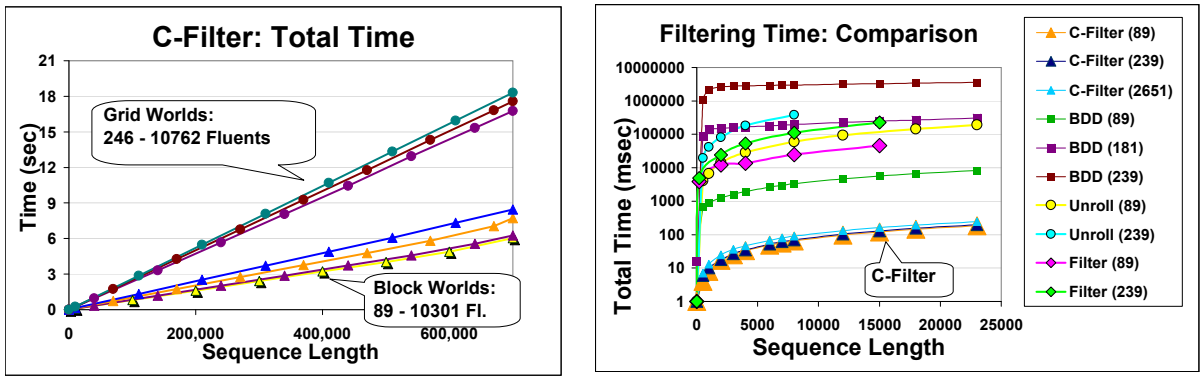


Figure 2.7: Left: Filtering time (sec) for C-Filter, applied to Block-World and Grid domains of different sizes. The time is linear, and does not depend on the domain size (slight differences in the graph are due to hash-table implementation). Right: Comparison of Filtering time (msec) for several methods (numbers represent domain size). Note that this is *log-scale*.

General idea: maintain explanation formulas, but instead of $f \leftrightarrow expl_f$ we maintain $l \rightarrow expl_l$ for every literal l, and $l_1 \wedge l_2 \rightarrow expl_{l_1,l_2}$ for every pair of literals.

2.6 Experimental Evaluation

Our Filtering algorithm was implemented in C++. Our implementation could also handle parametrized domain descriptions, such as STRIPS. We tested it on AI-Planning domains (Figure 2.8 lists several) of various sizes and observation models. We generated long action-observation sequences with a random sequence generator (implemented in Lisp), and ran inference on the results using our own *C-DPLL* and *NoClause* ((Thiffault et al., 2004b)). circuit SAT solver.

Blocks: 108/124	Ferry: 163/17	Grid: 251/53
Gripper: 110/6	Hanoi: 259/16	Logistics: 176/16
Movie: 47/13	Tsp: 98/15	

Figure 2.8: Overview of *C-Filter* experiments: AI-Planning domains (2000+ fluents, 10000 steps). Results presented as Filtering time/Model finding time (both in msec).

Figures 2.7, 2.8, 2.9 present some of the results. Figure 2.7 (left) shows that *C-Filter* is linear in the sequence length; note that time depends on the domain but *not* on the domain size. In both Block-World and Grid-World, filtering time almost does not change, even when the number of fluents grows 100 times larger (slight difference in the graph is due to hash-table implementation, instead of an array; the circuit size does not depend on this implementation and was the same).

The right part of Figure 2.7 shows a comparison to other filtering methods. We compared our algorithm to (1) *Filter* (Amir and Russell, 2003) (in Lisp), (2) Filtering by unrolling the system over t steps (using $|P|t$ propositions), (3)

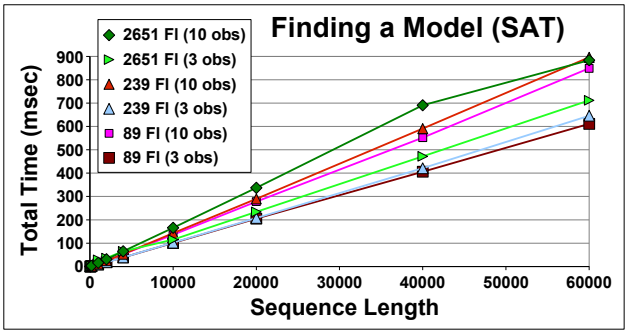


Figure 2.9: Total time for finding a model (msec), for Block-Worlds of different size and number of observations per step.

BDD-based Filtering, based on the BuDDy package (Lind-Nielsen, 1999). *C-Filter* outperformed them all, sometimes by orders of magnitude; note that the graph is *log-scale*.

Comparison Analysis: BDD sizes depend highly on variable ordering. Even for some very simple circuits, the representation can have either linear or exponential size depending on the order (finding an optimal ordering is known to be NP-complete). The long processing time at the beginning is due to heuristic methods that try to achieve a good ordering; after a while, a good ordering was reached, making processing faster. Even with those heuristics, we could not process large (> 300) domains. *Filter* and *Unroll* were also slower, and could not process long sequences or large domains (also, *Filter* can handle only a limited class of domains). *Unroll* suffers from the frame problem, i.e. needs to explicitly state the fluents that do not change ($f_t \leftrightarrow f_{t+1}$), and therefore depends on the domain size. *C-Filter* , however, managed to handle large domains (hundreds of thousands of fluents), taking a few milliseconds per step.

Figure 2.9 shows the time to find a model for the resulting circuits using a modified version of *NoClause*. Significantly, reasoning time grows only linearly with t. This allows practical logical filtering over temporal sequences of unbounded length. Note that the more observations an agent gets, the more constrained his belief state is. Therefore, it takes longer to find a satisfying model (also, the formula is larger).

2.7 Conclusions

A straightforward approach to filtering is to create all the prime implicates (or all consequences) at time $t+1$ from the belief state representation of time t. Previous work (e.g. (Liberatore, 1997b)) showed that deciding if a clause belongs to the new belief state is coNP-complete, even for deterministic domains. This discouraged further research on the problem.

Nevertheless, in this work we presented an exact and tractable filtering algorithm for all deterministic domains. Our result is surprising because it shows that creating a representation of *all* of the consequences at time $t+1$ is easier (poly-time) than creating the new belief state piecemeal.

Several approaches were developed in the past to represent belief states in logic (e.g., BDDs, (Amir and Russell, 2003)), but none of them guaranteed compactness. The key to our advance was our *logical circuits* representation. We also showed how to maintain NNF-Circuits.

The results obtained here have implications in many important AI-related fields. We expect our algorithms to apply to planning, monitoring and controlling, and perhaps stochastic filtering. We plan to explore these directions in the future.

Chapter 3

Learning in Partially Observable Worlds

3.1 Introduction

In the previous chapter we discussed agents that track the state of the world in a *known* environment. That is, the agents knew the possible outcomes of their actions, although they could not always observe them. However, the problem becomes a lot more challenging if we relax this assumption.

We now switch our attention to agents that operate in unfamiliar domains. Such agents can act intelligently if they learn the world's dynamics. Understanding the world's dynamics is particularly important in domains whose complete state is hidden and only partial observations are available. Example domains are active Web crawlers (that perform actions on pages), robots that explore buildings, and agents in rich virtual worlds.

Learning domain dynamics is difficult in general partially observable domains. An agent must learn how its actions affect the world as the world state changes and it is unsure about the exact state before or after the action.

In this chapter we present a logical approach to scaling up action learning in deterministic partially observable domains. We give special attention to the relational case; focusing on deterministic domains and the relational approach yields a strong result. The algorithm that we present learns relational schema representations that are rich and surpass much of PDDL (Ghallab et al., 1998). Many of the benefits of the relational approach hold here, including faster convergence of learning, faster computation, and generalization from objects to classes.

We show that the update step takes polynomial time, is exact (it includes all consistent models and only them), and increases the formula size by at most a constant additive (without increasing the number of state variables). We do so by updating a circuit (directed acyclic graph) representation of the formula. We conclude that the overall exact learning problem is tractable, when there are no stochastic interferences; it takes time $O(t \cdot p^{k+1})$, for t time steps, p predicates, and k the maximal precondition length. Thus, this is the first tractable relational learning algorithm for partially observable relational domains.

3.1.1 Related Work

This chapter is based on (Shahaf et al., 2006) and (Shahaf and Amir, 2006).

A number of previous approaches to learning action models automatically have been studied in addition to aforementioned work (Amir, 2005). Approaches such as (Wang, 1995; Gil, 1994; Pasula et al., 2004) are successful for fully observable domains, but do not handle partial observability. In partially observable domains, the state of the world is not fully known, so assigning effects and preconditions to actions becomes more complicated.

Our approach is closest to (Amir, 2005). There, a formula-update approach learns the effects (but not preconditions) of STRIPS actions in propositional, deterministic partially observable domains. In contrast, our algorithm learns models (preconditions and effects) that include conditional effects in a very expressive relational language. Consequently, our representation is significantly smaller, and the algorithm scales to much larger domains. Finally, our algorithm can generalize across instances, resulting in significantly stronger and faster learning results.

One previous approach in addition to (Amir, 2005) that handles partial observability is (Qiang Yang and Jiang, 2005). In this approach, example plan traces are encoded as a weighted maximum satisfiability problem, from which a candidate STRIPS action model is extracted. A data-mining style algorithm is used in order to examine only a subset of the data given to the learner, so the approach is approximate by nature.

Hidden Markov Models can be used to estimate a *stochastic* transition model from observations. However, the more complex nature of the problem prevents scaling up to large domains. These approaches represent the state transition matrix explicitly, and can only handle relatively small state spaces. Likewise, structure learning approaches for Dynamic Bayesian Networks are limited to small domains (e.g., 10 features (Ghahramani and Jordan, 1997; Boyen et al., 1999)) or apply multiple levels of approximation. Importantly, DBN based approaches have unbounded error in deterministic settings. In contrast, we take advantage of the determinism in our domain, and can handle significantly larger domains containing over 1000 features (i.e., approximately 2^{1000} states).

Another related approach is structure-learning in Dynamic Bayes Nets (Friedman et al., 1998). This approach addresses a more complex problem (stochastic domain), and applies hill-climbing EM. It is a propositional approach, and consequently it is limited to small domains. Also, it could have unbounded errors in discrete deterministic domains.

In recent years, the *Relational* Paradigm enabled important advances (Friedman et al., 1999; Dzeroski and Luc De Raedt, 2001; Pasula et al., 2004; Getoor, 2000). This approach takes advantage of the underlying structure of the data, in order to be able to generalize and scale up well. We incorporate those ideas into Logical Learning and present a *relational* logical approach.

This chapter is structured as follows: we introduce the learning problem in Section 3.2, explain the intuition behind our representation and give a meta-algorithm in Section 3.3. We present several languages (and their corresponding algorithms) in Section 3.4. We give special attention to the relational model. We proceed to analysis (Section 3.5) and experiments (Section 3.6), and then conclude.

3.2 A Transition Learning Problem

We now illustrate the combined problem of learning the transition model and tracking the world with an example. Consider a robot operating a service elevator (Figure 3.1). The robot can load and unload boxes, and make the elevator go up and down.

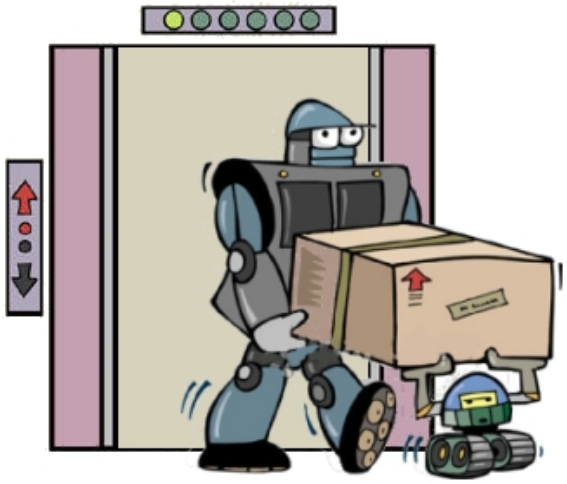


Figure 3.1: Learning example: a robot in a service elevator

The robot's knowledge of the world is incomplete, both because it can only observe the immediate surroundings of the elevator and because it lacks knowledge about the world's dynamics. That is, it is unsure about the current state of the world and about the way actions change it.

The robot is given some initial knowledge, and then it performs a sequence of actions and receives partial observations. Its goal is to determine the effects of his actions (to the extent he can, theoretically), while also tracking the world.

We now repeat the definition of a transition system (Definition 2.2.1).

Definition 3.2.1 *A **transition system** is a tuple* $\langle P,S,A,R \rangle$

- P *is a finite set of fluents.*
- $S \subseteq Pow(P)$ *is the set of world states; a state* $s \in S$ *is the subset of* P *containing exactly the fluents true in* s*.*
- A *is a finite set of actions.*
- $R \subseteq S \times A \times S$ *is the transition relation.*

$\langle s, a, s' \rangle \in R$ means that state s' is the result of performing action a in state s. When we refer to deterministic transition relations we sometimes use the function notation, $R(s, a) = s'$.

In a two-floor elevator world (Figure 3.2), P includes fluents such as *(elevator-floor1)*, *(box-at floor1)*, *(box-at floor2)*, *(box-at elevator)*.

The actions A depend on the domain description. Taking the elevator up can be represented as *GoUp()*, *GoUp(to)*, or (in the case of STRIPS) *GoUp(from,to)*.

As noted before, the robot cannot observe the state of the world completely and it does not know how his actions change it. In the previous chapter the robot handled state uncertainty by maintaining a *belief-state* – a set of world states he considered possible. Similarly, the robot that operates in an unknown environment can maintain a *transition belief state*: a set of possible $\langle$*world state*, *transition relation*$\rangle$ pairs that might govern the world.

Definition 3.2.2 *A **transition belief state** Let $\mathcal{R}$ be the set of all possible transition relations on S, A. Every $\rho \subseteq S \times \mathcal{R}$ is a transition belief state.*

We sometimes refer to it as "belief state" (when the meaning is clear from the context), to strengthen the connection to the previous chapter.

Informally, a transition belief state ρ is the set of pairs $\langle s, R \rangle$ that the agent considers possible. The agent updates his belief state as he performs actions and receives observations. We now define semantics for Simultaneous Learning and Filtering, or $\mathcal{SLAF}$.

Definition 3.2.3 (Simultaneous Learning and Filtering) *$\rho \subseteq S \times \mathcal{R}$ a transition belief state, a_i are actions. We assume that observations o_i are logical sentences over P.*

1. $\mathcal{SLAF}[\epsilon](\rho) = \rho$ *(ϵ: an empty sequence)*
2. $\mathcal{SLAF}[a](\rho) = \{\langle s', R \rangle \mid \langle s, a, s' \rangle \in R, \langle s, R \rangle \in \rho\}$
3. $\mathcal{SLAF}[o](\rho) = \{\langle s, R \rangle \in \rho \mid o \text{ is true in } s\}$
4. $\mathcal{SLAF}[\langle a_j, o_j \rangle_{i \leq j \leq t}](\rho) =$

   $\mathcal{SLAF}[\langle a_j, o_j \rangle_{i < j \leq t}]\ (\mathcal{SLAF}[o_i](\mathcal{SLAF}[a_i](\rho)))$

We call step 2 progression with a and step 3 filtering with o. In short, we maintain a set of pairs that we consider possible; the intuition behind this definition is that every pair $\langle s', R \rangle$ becomes a new pair $\langle \tilde{s}, R \rangle$ as the result of an action. If an observation discards a state $\tilde{s}$, then all pairs involving $\tilde{s}$ are removed from the set. We conclude that R is not possible when all pairs including it have been removed.

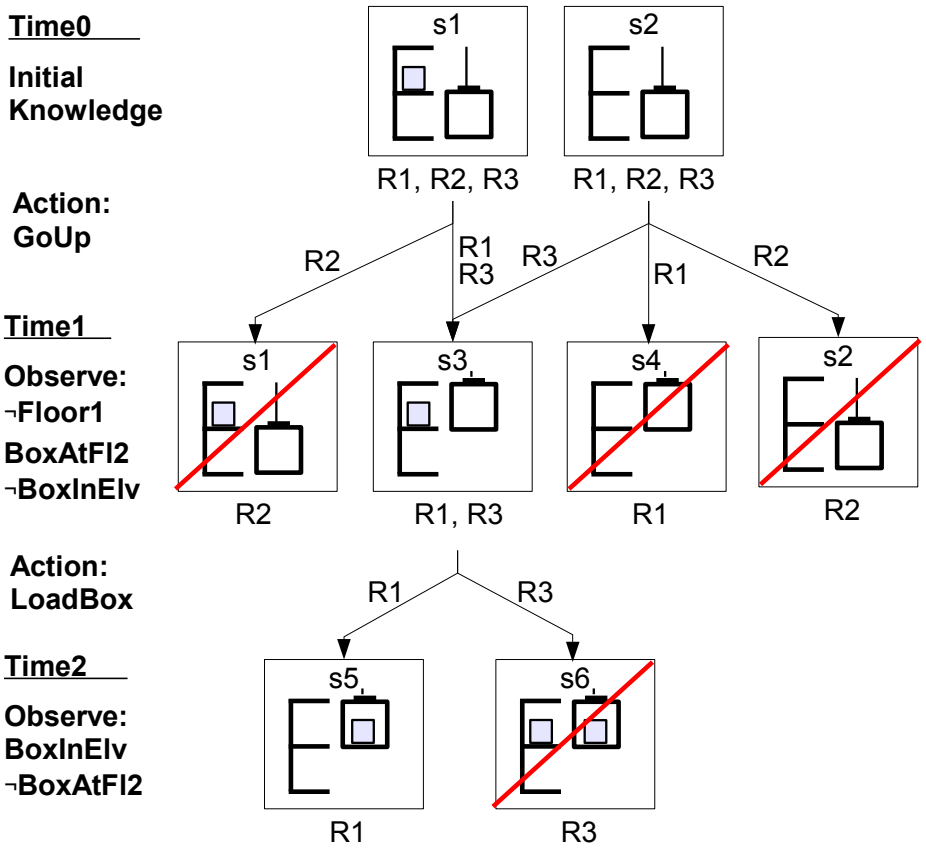


Figure 3.2: A transition belief-state update example: on the left – time line, including actions and observations. On the right – the corresponding transition belief-state. Under each state s appear the transition relations R such that $\langle s, R\rangle$ in the belief-state.

3.2.1 Extended SLAF Example

We now show a complete learning problem scenario. Consider the two-floor service elevator world in Figure 3.2; the elevator robot is trying to learn the dynamics of the world.

For the clarity of this example, we consider a very simple initial belief-state. Assume that the elevator is on the first floor and the robot can observe its surrounding: it knows the elevator is in the 1^{st} floor, is currently empty, and there is no box near it. There may or may not be a box in the 2^{nd} floor (actually there is, but the robot cannot tell).

The two world-states the robot considers possible are depicted in the topmost part of the Figure 3.2 ($s1, s2$). In addition to the uncertainty about the initial world state, the robot considers three transition relations possible:

- $R1$ – the correct transition relation: *GoUp* (*GoDown*) takes the elevator to the second (first) floor, *LoadBox* and *UnloadBox* try to move the box between the elevator and the current floor.
- $R2$ – same as $R1$, but *GoUp* does not take the elevator to the second floor.

- $R3$ – same as $R1$, but *GoUp* and *LoadBox* also cause a box to appear on the second floor.

In other words, the robot has six possible $\langle s, R\rangle$ pairs in its initial belief state. This is shown in the top part of Figure 3.2: the two states $(s1, s2)$ are depicted, and the possible transition relations that correspond to each state appear underneath it.

The robot than performs a short action-sequence (*GoUp*, *LoadBox*), receiving observations after each action. The timeline and the observations appear on the left of Figure 3.2. We now show how the robot's belief state progresses with this sequence.

After performing the first action, *GoUp*, each of the $\langle s, R\rangle$ pairs progresses into $\langle s', R\rangle$, such that s' is the result of performing *GoUp* in s if R is the true transition relation. For example, $\langle s1, R1\rangle$ changes into $\langle s3, R1\rangle$: $R1$ predicts the next state will be the same as $s1$ but with the elevator in the second floor (that is, $s3$).

The complete transition is shown in Figure 3.2:

- If $R1$ is true, we end up in state $s3$ or $s4$: the elevator is in the second floor, but we still do not know if there is a box there.
- If $R2$ is true, the initial state does not change: $R2$ states that *GoUp* keeps the elevator in the first floor.
- If $R3$ is true, then both initial states progress into the same state ($s3$): the elevator is in the second floor and there is a box near it. This is because $R3$ assumes going up created a box in the second floor.

At this stage, the robot's belief state contains five elements. The robot now receives an observation: the elevator is in the second floor, and there is a box outside. This immediately allows it to delete from its belief state every $\langle s, R\rangle$ pair such that s is not consistent with the observation: in this case, it deletes $\langle s1, R2\rangle$, $\langle s2, R2\rangle$ and $\langle s4, R1\rangle$.

Note that after this elimination, there are no longer pairs that include $R2$. That is, $R2$ is no longer a possible transition relation. The robot currently knows his exact state, $s3$, but it still needs to choose between $R1$ and $R3$. After picking up the box, the picture becomes clearer: $R1$ predicts no box on the second floor, while $R3$ disagrees. Observing no box leaves the robot with only one pair in its belief-state. Thus, learning is complete.

3.3 Learning Transition Models with Logical Circuits

Similar to the Filtering problem, enumerating transition belief states is clearly intractable. In this section we present an algorithm for updating transition belief states. This algorithm builds on the results of (Amir, 2005); it uses a representation of logical circuits which guarantees compactness and exact computation, thus extending those previous results.

This chapter presents several logical representations suited for transition learning problems, described in detail in Section 3.4. However, in this section we explain our method independently from the language we choose. In other words, this section describes our meta-algorithm.

3.3.1 Representation

We use logical languages to represent transition belief states. Those languages are composed of two different types of propositions:

1. Fluents, P (representing the state of the world). They can be purely propositional, or (as in later sections) they can also be ground instances of relational fluents.

2. *Action propositions*, L, which are propositions that represent the possible transition relations.

The exact nature of the action propositions depends on the transition relations that we want to be able to represent. The idea is that any transition relation $R \in \mathcal{R}$ can be fully described by some assignment to those propositions. For example, those propositions could correspond to if-then rules:

"*UnloadBox* **causes** *(box-at floor1)* **if** *(elevator-floor1)* $\wedge$*(box-at elevator)*"

Or if-then schemas:

"*GoToFloor(X)* **causes** *(elevator-at X)* **if** *TRUE*"

Alternatively, they can respond to a part of an if-then rule, e.g.

"*(elevator-floor1)* appears in the effect of *GoToFloor(floor1)*"".

We discuss these in detail in the following sections. After choosing a vocabulary $L \cup P$, we can use logical formulas to encode belief states.

A belief state formula φ over $L \cup P$ is equivalent to the belief-state

$$\{\langle s, R\rangle \mid s \wedge R \wedge \varphi \text{ satisfiable}\}$$

s and R are expressed as logical formulas over some $\Sigma \supseteq L \cup P$. In the rest of the thesis logical formulas and set-theoretic notions of belief states will be used interchangeably.

Example

Refer back to the elevator example of 3.2. In this example

$P = \{$*(elevator-floor1)*, *(box-at floor1)*, *(box-at floor2)*, *(box-at elevator)*$\}$.

To represent $R1$, $R2$ and $R3$ we use the vocabulary

$L = \{$"*GoUp* **causes** $\neg$*(elevator-floor1)*", "*GoUp* **causes** *(box-at floor2)*", "*LoadBox* **causes** *(box-at floor2)*"$\}$.

The idea is that those proposition describe the effects of actions that are $R1, R2$ and $R3$ disagree on (*GoUp* causes the elevator to be in the second floor, and so on). $R1, R2$ and $R3$ differ in the assignments to those propositions: $R1$ assigns true only to the first one, $R2$ assigns false to all, and $R3$ assigns true to all. This will be used in the next section to construct the belief-state representation.

3.3.2 Circuit Representation

The previous section explained the intuition of the logical languages that we use. However, instead of flat formulas, we use a *logical circuit* representation for transition belief-states. A very similar representation is used in the previous chapter for Filtering belief-states (Section 2.3); here we only reiterate the main concepts.

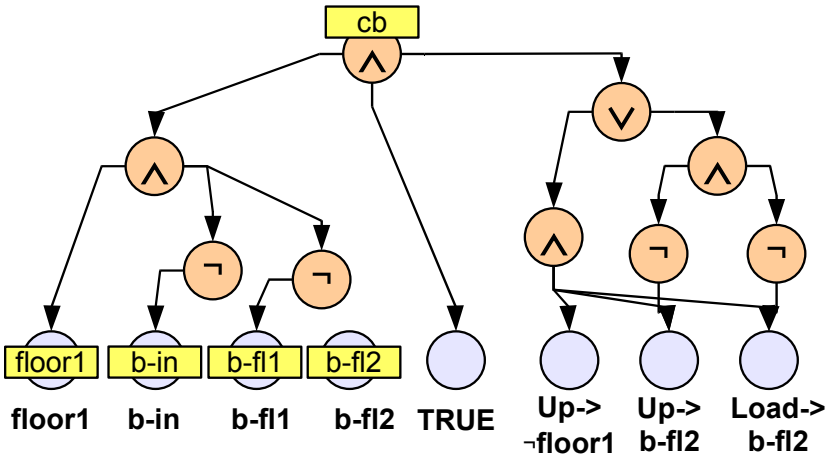


Figure 3.3: Circuit example: initial transition belief-state for the elevator robot. The leaves on the left are fluents of time 0, and on the right – transition propositions

Circuits are directed acyclic graphs: leaves correspond to variables (P of time 0 and L), and internal nodes are assigned logical connectives (such as $\wedge, \vee, \neg$). An internal node is interpreted as the formula rooted in this node – the application of the connective to the child nodes. In addition to the circuit, we maintain pointers to several special nodes:

1. An *explanation formula* for every fluent $f \in P$, $expl_f$, representing its value as a function of the leaves. Note that, unlike the previous chapter, the value is a function of the initial state *and* the transition relation.
2. A formula *cb*, or *constraint base*, representing the constraints on the values of the leaves.

Belief-states as Circuits

A circuit corresponds to belief-state ρ in the following way: each assignment to the leaves that satisfies the constraints, *cb*, corresponds to a state $\langle s, R \rangle \in \rho$. R is determined by the assignment to the leaves of L, and s is determined by the

explanation formulas: $f \in s$ iff *expl*$_f$ is *True* for this assignment. Assignments that do not satisfy *cb* do not correspond to any state in ρ.

Refer the initial belief-state circuit in Figure 3.3. This is a representation of the elevator robot's initial belief state. Nodes are circles, and pointers are depicted as rectangles.

Recall the previous section for the interpretation of the nodes on the right: the rightmost node, for example, corresponds to "*LoadBox* **causes** *(box-at floor2)*".

cb asserts that the robot is in the first floor, and there is no box in the elevator or outside it. *cb* also states that $R1$, $R2$ or $R3$ are true. We also keep a node to represent *TRUE*, although any tautology would do. This is a simple example, as the explanation pointers are all assigned to leaves. As we explain later, they can represent complex formulas.

The intuition behind this representation is that the circuit encodes the way the belief-state changes through a sequence of actions. The leaves represent the state of the world before the sequence, and the pointers represent the state after the sequence (as a function of the initial state). For this reason, the leaves in Figure 3.3 are annotated with *fluent-name*.

3.3.3 Meta-Algorithm Overview

Meta-C-SLAF is presented in Figure 3.4 and demonstrated in Figure 3.5. It receives an action-observation sequence, $\langle a_i, o_i \rangle_{0<i\leq t}$, an initial belief state formula, φ, over $P \cup L$, a domain description D. It outputs the filtered belief state as a logical circuit.

D includes the semantics of L, that is the description of actions in A as formulas over L, P. An action has a precondition and a list of conditional effects. *Every* deterministic domain can be represented like this (including, for example, STRIPS and PDDL). The precondition has to hold before executing the action, and the conditional effects occur iff their condition holds.

The algorithm starts by processing the domain description; it extracts *next-state* and *possible* formulas. *Poss(a)* is the precondition of action a, and *NextVal(a,f)* is the value of fluent f after action a as a function of the world before it (intuitively, the formula "a caused f or f already held and a did not change it"). Note that we only calculate this for fluents that might have been affected by a; the circuit structure obviates the need for specifying that other fluents do not change (Frame axioms).

The algorithm also extracts that *base* formula, which encodes restrictions on transition models (for example, an action cannot have contradicting effects).

In our example, action *GoUp* is always possible to execute, and it affects the fluents *(elevator-floor1)*, *(box-at floor2)*. It causes *(elevator-floor1)* to be true iff it was true before the action, and "*GoUp* **causes** $\neg$*(elevator-floor1)*"

[1] *Cause(a,f)* represents the conditions for a to cause f, extracted from the domain description (See (*))

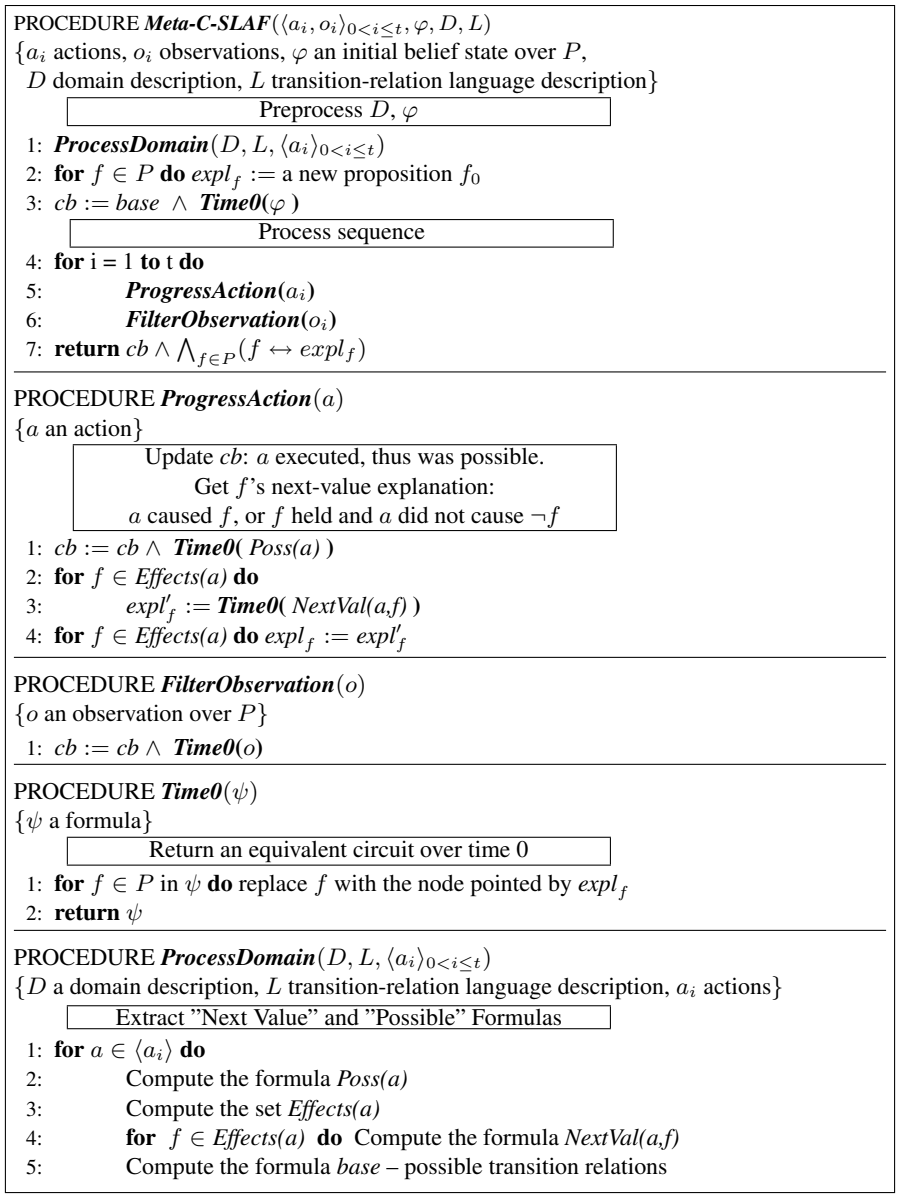


Figure 3.4: *Meta-C-SLAF* Algorithm

is false. It causes *(box-at floor2)* to be true iff it was true before or "*GoUp* **causes** *(box-at floor2)*" is true. In other words,

NextVal(GoUp,(elevator-floor1)) =

(elevator-floor1) $\wedge\neg$"*GoUp* **causes** $\neg$*(elevator-floor1)*"

NextVal(GoUp,(box-at floor2)) =

(box-at floor2) $\vee$ "*GoUp* **causes** *(box-at floor2)*"

Poss(GoUp) = TRUE

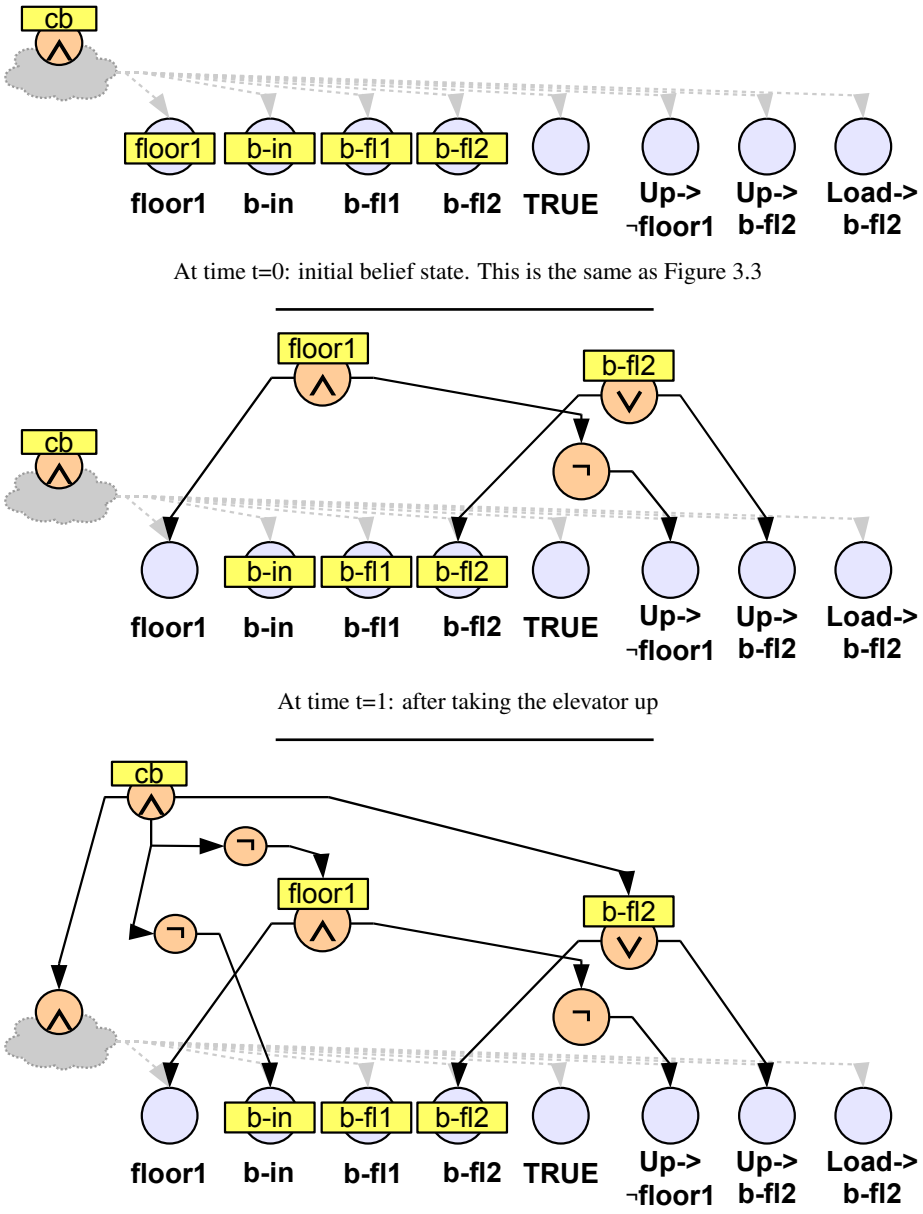


Figure 3.5: At time t=1, after observing the elevator's surroundings

The update step is illustrated in Figure 3.5 for the elevator example of Figure 3.2. Figure 3.5a shows the initial belief-state. This is the same circuit shown in Figure 3.3, but for clarity we do not show the *cb* formula explicitly this time (this formula is represented by the cloud).

Again, the circuit contains a node for each proposition in L and fluent $f \in P$, and the pointer *e(f)* is set to it. The pointer is shown as a rectangle. 3.5b shows how we progress with the first action, *GoUp*. The action is always applicable (the *Poss* formula is *TRUE*), so *cb* does not change. The only fluents that can be affected by this action are

(elevator-floor1), *(box-at floor2)*. We progress each one of them, by building the NextVal formulas (see Equation 4.1). After building the formulas that explain the current state of those fluents, we assign the corresponding pointers to the root of those formulas.

3.5c goes one step further, and filters with the observation: $\neg$*(elevator-floor1)*$\wedge$*(box-at floor2)*$\wedge\neg$*(box-at elevator)*. To do this, we construct that formula, using the nodes pointed by the pointers whenever we need a fluent. The resulting circuit is then added to *cb*.

Note that we update the only possibly-affected fluents (no Frame axioms). Also, when we refer to fluents of the previous time step we use the internal node constructed in the 3.5b; this allows recursive sharing of subformulas, resulting in compactness.

3.4 Transition Relation Languages

To make this meta-algorithm into a real algorithm, we need to specify some more details. Most importantly, we should specify the language we use to describe transition relations and its semantics. In this section we discuss some alternative languages, their expressiveness and complexity.

3.4.1 STRIPS

We start with the STRIPS language. This language was introduced by Fikes and Nilsson (Fikes et al., 1981), and has since shaped most the work in Planning.

Expressivity

In STRIPS, each action has a precondition and an effect, both of them conjunctions of literals. The precondition needs to be true to apply the operator, and the effects are the way the world changes from its previous state.

For example, in our elevator domain the action *(up floor1 floor2)* might look like:

```
(:action up
  :parameters (floor1 floor2)
  :precondition (and (lift-at floor1) (above floor1 floor2))
  :effect (and (lift-at floor2) (not (lift-at floor1))))
```

Note that if we use the parameterized notation (*action(object1, object2, ...)*), the action affects only objects that were specified as parameters. For example, the action of taking the elevator up has to include not only the destination floor but also the current floor; otherwise, it would be impossible to state that taking the elevator up causes the elevator to stop being in the floor it was in.

Logical Representation

In order to represent STRIPS actions, we use a simple propositional vocabulary. Define action propositions

$$L_f = \bigcup_{a \in A} \{\text{“}a \textbf{ causes } f\text{”}, \text{“}a \textbf{ causes } \neg f\text{”}, \text{“}a \textbf{ keeps } f\text{”}, \text{“}f \textbf{ precondition of } a\text{”}, \text{“}\neg f \textbf{ precondition of } a\text{”} \}$$

for every $f \in P$. Let the vocabulary for the formulas representing transition belief states be defined as $L = P \cup \bigcup_{f \in P} L_f$. Intuitively: “$a$ **causes** f”, “a **causes** $\neg f$” is true if and only if action a in the transition relation causes f ($\neg f$) to hold after a is executed. “a **keeps** f” is true if and only if action a does not affect fluent f. “f **precondition of** a” (“$\neg f$ **precondition of** a”) is true if and only if f ($\neg f$) is in the precondition of a.

For example, in our elevator domain we would like to learn that

“*(up floor1 floor2)* **causes** *(lift-at floor2)*”,

“*(up floor1 floor2)* **causes** ¬*(lift-at floor1)*”,

“*(lift-at floor1)* **precondition of** *(up floor1 floor2)*”,

“*(above floor1 floor2)* **precondition of** *(up floor1 floor2)*”

“*(up floor1 floor2)* **keeps** *(box-at floor2)*”.

Transition Rule Semantics and the *STRIPS-C-SLAF* Algorithm

In order to learn transition rules, we should first give our new propositions semantics. We consider several cases: the first, and most common, assumes that the sequence contains only successful actions. In other words, it assumes that if an action with precondition G and effect F has executed, then G held before it and F after it. More formally,

Definition 3.4.1 (Transition Rules Semantics: Successful Actions) *Given $s \in S, a \in A$ and R, a transition relation represented as a set of transition rules, we define s', the result of performing action a in state s:*

1. *For every literal f, if “a **causes** f” $\in R$ then $s' \models f$.*
2. *If “a **keeps** f” then $s \models f \Leftrightarrow s' \models f$.*

*In addition, we assume the actions were successful: For every literal g, if “g **precondition of** a” $\in R$,then $s \models g$.*

Given the semantics, we can now turn the meta-algorithm from Section 3.3.3 into a real algorithm, *STRIPS-C-SLAF* . To do this, we only need to show how to calculate the *next-state*, *possible* and *base* formulas.

NextVal(a,f) := [“a **causes** f” $\vee$ (“a **keeps** f” $\wedge \neg \wedge f$)]

[the value of f after a]

Poss(a) := $\bigwedge_{f \in P}$(¬“f **precondition of** a” $\vee f$) $\wedge$ (¬“¬f **precondition of** a” $\vee \neg f$)

[the precondition of action a]

Also, let the formula *base* encode the axioms that inconsistent models are not possible. That is, models in which "f **precondition of** a" and "$\neg f$ **precondition of** a" both hold, or models where it is not the case that exactly one of { "a **causes** f", "a **causes** $\neg f$", "a **keeps** f" } holds are disallowed.

We plug these formulas into *Meta-C-SLAF* . This specifies our first learning algorithm, *STRIPS-C-SLAF* .

3.4.2 Ground

Expressivity

STRIPS is a very useful language, but it is not very expressive. It does not allow conditional effects, or preconditions that are not conjunctions. For example, flipping a switch cannot be modeled in STRIPS as one action; instead, we should use *switchUp*, *switchDown*. In this section we present another language, which can represent any deterministic transition relation.

For example, flipping the switch might look like:

```
(:action flip
  :parameters (switch1)
  :effect (and (when (up switch1) (not (up switch1)))
               (when (not (up switch1)) (up switch1)))
```

Logical Representation

We define a vocabulary of *action propositions* which can be used to represent transition relations as propositional formulas. Let $\mathcal{F}$ be the set of literals of P. Let $L = \{$"a **causes** F **if** G"$\}$, where $a \in A$, $F \in \mathcal{F}$ a literal, G a conjunction of literals (a term). We call "a **causes** F **if** G" a transition rule, G its *precondition* and F its *effect*. "a **causes** F **if** G" means "if G holds, executing a causes F to hold".

Claim 3.4.2 *Any deterministic transition relation can be described by a finite set of such propositions.*

Any deterministic transition relation can be described by a finite set of propositions "a **causes** F' **if** G'" where F' and G' are any formulas (for example, by letting G' be a complete world specification, and F' the effect of executing a in this world). Note that "a **causes** F' **if** $G1 \vee G2$" can be replaced by "a **causes** F' **if** $G1$","a **causes** F' **if** $G2$". That is, if G' is some formula, we can take its DNF form and split to rules with term preconditions. F' can be converted to DNF too: it must be a term, since the domain is deterministic. Note that "a **causes** $F1 \wedge F2$ **if** G" is equivalent to "a **causes** $F1$ **if** G","a **causes** $F2$ **if** G". Again we can split the rule into several rules with a literal as their effect, resulting in the form descried above.

For example, using this language we would like to learn that "*(up floor1 floor2)* **causes** *(lift-at floor2)* **if** *(lift-at floor1)* $\wedge$ $\neg$*(lift-at floor2)*" , "*(up floor1 floor2)* **causes** $\neg$*(lift-at floor1)* **if** *(lift-at floor1)* $\wedge$ $\neg$*(lift-at floor2)*".

Imagine the elevator has an on-off switch, and the action *(switch)* flips it up and down. This is a conditional action, which cannot be represented in STRIPS. It is easily represented as

"*(switch)* **causes** *(on elevator)* **if** $\neg$*(on elevator)*"

"*(switch)* **causes** $\neg$*(on elevator)* **if** *(on elevator)*".

Transition Rule Semantics and the *C-SLAF* Algorithm

We can again assume successful actions. The definition is very similar to the successful STRIPS semantics (Definition 3.4.1).

Definition 3.4.3 (Transition Rules Semantics: Successful Actions) *Given $s \in S, a \in A$ and R, a transition relation represented as a set of transition rules, we define s', the result of performing action a in state s: if "a **causes** F **if** G" $\in$ R, then $s \models G$ and $s' \models F$. The rest of the literals do not change.*

However, we can assume different semantics. For the sake of the example, we now present a different semantics, and show how it affects the algorithm. We relax the assumption of successful actions. Instead, we assume that all actions were executed, but if the precondition did not hold nothing changed (we can also make action failure lead to a *sink state* of some sort).

Definition 3.4.4 (Transition Rules Semantics: Conditional Effect) *Given $s \in S, a \in A$ and R, a transition relation represented as a set of transition rules, we define s', the result of performing action a in state s: if "a **causes** F **if** G" $\in$ R and $s \models G$, then $s' \models F$. The rest of the literals do not change. If there such s', we say that action a is* possible *in s.*

With this semantics, we can now construct an algorithm for the ground case. The algorithm *C-SLAF* is created from our meta-algorithm (Figure 3.4) by plugging in the following formulas:

Poss$(a, f) =$ (1)

$\neg[(\bigvee_G$ ("a **causes** f **if** G" $\wedge$ G)) $\wedge$ $(\bigvee_{G'}$ ("a **causes** $\neg f$ **if** G'" $\wedge$ G'))]

NextVal$(a, f) =$ (2)

$[(\bigvee_G$ ("a **causes** f **if** G" $\wedge$ G))$\vee$ $(f \wedge (\bigwedge_{G'} \neg$ ("a **causes** $\neg f$ **if** G'" $\wedge$ G')))]

base $:=$

$[\bigwedge_{a,F,G} \neg$("a **causes** F **if** G" $\wedge$ "a **causes** $\neg F$ **if** G")]$\wedge$ $[\bigwedge_{a,F,G \to G'}$("$a$ **causes** F **if** G'" $\rightarrow$ "a **causes** F **if** G")]

3.4.3 Relational

The ground algorithm allows complex domains to be learned, but its complexity is sometime high (as we show later), and it cannot generalize. In this section we present another family of algorithms, designed to take advantage of relational domains.

Expressivity

For humans, the action of opening a door and opening a book is the same meta-action; in both cases, the object will be opened. We try to capture this intuition in the languages that we present next.

For example, we can represent the opening action as:

```
(:action open
  :parameters (?x)
  :precondition (not (opened ?x))
  :effect (opened ?x))
```

Note that `?x` is a variable: therefore, this is an action *schema* rather than a ground action (compare to the previous sections; in the Ground language, we would replace this schema by many ground actions, such as *(open book)*, *(open door)*).

To take advantage of the relational framework, we need to slightly change our transition definition:

Definition 3.4.5 *A **relational transition system** is a tuple* $\langle$*Obj*, *Pred*, *Act*, *P,S,A,R*$\rangle$

- *Obj*, *Pred, and Act are finite sets of objects in the world, predicate symbols, and action names, respectively. Predicates and actions also have an arity.*
- P *is a finite set of fluents of the form* $p(c_1, ..., c_m)$*, where* $p \in$ *Pred,* $c_1, .., c_m \in$ *Obj.*
- $S \subseteq$ *Pow(P) is the set of world states; a state* $s \in S$ *is the subset of* P *containing exactly the fluents true in* s*.*
- $A \subseteq \{a(\bar{c}) \mid a \in$ *Act*, $\bar{c} = (c_1, .., c_n), c_i \in$ *Obj*, $\}$*, ground instances of Act.*
- $R \subseteq S \times A \times S$ *is the transition relation.*

$\langle s, a(\bar{c}), s' \rangle \in R$ means that state s' is the result of performing action $a(\bar{c})$ in state s.

3.4.4 Logical Languages for the Relational Case

Our logical languages represent transition belief states, using ground relational fluents from P (representing the state of the world), and *action-schemas*, which are propositions that represent the possible transition relations. Informally,

schemas correspond to if-then rules; together, they are very similar to actions' specification in PDDL (Ghallab et al., 1998). For example, a schema in our language is

"*SwUp(x)* **causes** *Up(x)* **if** *TRUE*"

[switching up an object causes it to be up, if *TRUE*].

This schema represents a set of *instances*– ground transition rules, e.g.

"*SwUp(lSw)* **causes** *Up(lSw)* **if** *TRUE*" and

"*SwUp(rSw)* **causes** *Up(rSw)* **if** *TRUE*".

Definition 3.4.6 (Schemas) *A schema is a* proposition *of the form "$a(x_1, ..., x_n)$* ***causes*** *F* ***if*** *G" (read: $a(\bar{x})$ causes F if G). $a \in$ Act is an n-ary action name, $\bar{x}$ are n different symbols, F (the effect) is a literal and G (the precondition) is a sentence, both over P_{Pat} which we now define. W.l.g., G is a conjunction of literals; otherwise, we can take its DNF form and split it to several schemas of this form.*

Let Pat be a set of symbols that includes $\{x_1, x_2, ...\}$. P_{Pat} is the set of patterned fluents over Pat:

$P_{Pat} = \{p(y_1, ..., y_m) \mid p \in Pred, y_1, .., y_m \in Pat\}$.

In other words, a schema is a transition rule containing variables. Its instances can be calculated by assigning objects to these variables; the result is a ground transition rule "$a(\bar{c})$ **causes** F **if** G" for $a \in A$,F, G over P. That is, every patterned fluent (*Up(x)*) becomes a fluent (*Up(lSw)*) after the assignment. In order to compute the instances, we may need to know some relations between objects, e.g. which switch controls which bulb. The set of possible relations is denoted by *RelatedObjs* (see *SL-H* below).

Transition Rule Semantics and the *C-SLAFS* Algorithm

We again assume that actions are always executable, and do not change the state unless one of their preconditions fired.

Definition 3.4.7 (Transition Rule Semantics: Conditional Effect) *Given a state s and a ground action $a(\bar{c})$, the resulting state s' satisfies every literal $F \in \mathcal{F}$ which is the effect of an activated rule (a rule whose precondition held in s). The rest of the fluents do not change– in particular, if no precondition held, the state stays the same. If two rules with contradicting effects are activated, we say that the action is not possible.*

Our algorithm for the relational case is presented in Figure 3.6. Just like in previous cases, the algorithm needs to calculate the *NextVal* and *Poss* formulas.

[1] If the language does not involve related objects, assume *RelatedObjs*=$\{$*TRUE*$\}$.

[2] Implementation depends on the schema language used.

PROCEDURE PossibleAct(f,a($\bar{c}$))
input: $f \in P$, $a(\bar{c})$ an action

1: $\psi = TRUE$
2: **for** *relobjs* $\in$ *RelatedObjs* [1] **do**
3: Compute all schema-instance pairs with effect f, $\{(sch+,inst+)\}$, and those with effect $\neg f$, $\{(sch\text{-},inst\text{-})\}$, regarding action $a(\bar{c})$ and *relobjs*. [2]
4: $\psi = \psi \wedge relobjs \rightarrow$
$\neg[(\bigvee sch+ \wedge prec(inst+)) \wedge (\bigvee sch- \wedge prec(inst-))]$
{the action is possible if *relobjs* is true}
5: **return** ψ

PROCEDURE NextVal(f,a($\bar{c}$))
input: $f \in P$, $a(\bar{c})$ an action

1: $\psi = TRUE$
2: **for** *relobjs* $\in$ *RelatedObjs* **do**
3: Compute all schema-instance pairs with effect f, $\{(sch+,inst+)\}$, and those with effect $\neg f$, $\{(sch\text{-},inst\text{-})\}$, regarding action $a(\bar{c})$ and *relobjs*.
4: $\psi = \psi \wedge relobjs \rightarrow$
$[\bigvee sch+ \wedge prec(inst+)] \vee [f \wedge \neg(\bigvee sch- \wedge prec(inst-))]$
{f's value after the action if *relobjs* is true}
5: **return** ψ

Figure 3.6: *C-SLAFS* Algorithm

The algorithm is given a ground action (such as *(open book1)*), and looks for all schema-instance pair that might change the next state. For example, the schema "*(open x)* **causes** *(opened x)* **if** *TRUE*" can influence the value if *(opened book1)*. It uses this to construct a formula for the next state of each fluent (the action caused it to hold, or it held and the action did not change it), and for the action to be possible (no contradicting effects).

The formula *base* restricts possible assignments:

$base :=$ $base_{relobjs} \wedge$

$\bigwedge_{a,F,G} \neg(\textbf{causes}(a, F, G) \wedge \textbf{causes}(a, \neg F, G))$

$\bigwedge_{a,F,G \rightarrow G'} [\textbf{causes}(a, F, G') \rightarrow \textbf{causes}(a, F, G)\ ]$

and $base_{relobjs}$ is a formula that the related objects must satisfy. It depends on the schema language used.

Schema Languages

Like in ground domains, transition relations in relational domains can be of different degrees of expressivity. We now present several languages to represent schemas, starting from our most basic language.

SL_0: The Basic Language

In this language, $Pat = \{x_1, x_2, ...\}$. For any schema "$a(x_1, ..., x_n)$ **causes** F **if** G", F and G include only symbols from $x_1, ..., x_n$. An instance is an assignment of objects to $x_1, ..., x_n$. No related objects are needed (we set $RelatedObjs = \{TRUE\}$).

Examples include "*SwUp(x_1)* **causes** *Up(x_1)* **if** *TRUE*",

"*PutOn(x_1, x_2)* **causes** *On(x_1, x_2)* **if** *Has(x_1)* $\wedge$ *Clear(x_2)*" (you can put a block that you hold on another, clear one). Any STRIPS domain can be represented in SL_0. Note that SL_0 can only describe domains in which every action $a(\bar{c})$ can affect only the elements in $\bar{c}$; the following extensions are more expressive.

SL-V: Adding Quantified Variables

Some domains permit quantified variables in effects and preconditions; there, an action can affect objects other than its parameters. For example,

"*sprayColor(x_1)* **causes** *Color(x_2, x_1)* **if** *RobotAt(x_3)* $\wedge$ *At(x_2, x_3)*" (spraying color x_1 causes everything at the robot's location to be painted).

Pat is still $\{x_1, x_2, ...\}$, but F, G can include *any* symbol from *Pat*. $\{x_1, ..., x_n\}$ are the action's parameters, $\bar{c}$ (thus, they are specified by the action $a(\bar{c})$). $\{x_{n+1}, ...\}$ represent free variables. Similarly to PDDL, free variables that appear in the effect part of the schema are considered to be universally quantified, and those that appear only in the precondition part are considered existentially quantified. In the previous example, x_2 is universally quantified and x_3 is existentially quantified. No related objects are needed.

In a more expressive variant of *SL-V*, the variables range only over objects that *cannot* be described any other way– that is, they do not range over the action's parameters, $\bar{c}$ (and in richer languages, not over their functions). This allows us to express defaults and exceptions, as in "blowing a fuse turns every light bulb off, except for a special (emergency) bulb, which is turned on", or "moving the rook to (c_1, c_2) causes it to attack every square (x, c_2) except for (c_1, c_2)". If *Pred* includes equality, we can use a simpler variant.

SL-H: Adding Hidden Object Functions

Pat $= \{x_1, x_2, ...\} \cup \{h_1, h_2, ...\}$. We write h_1 as a shorthand for $h_1(\bar{x})$. This extension can handle hidden objects– objects that are affected by an action, although they do not appear in the action's parameters. For example, the rules

"*SwUp(lSw)* **causes** *On(lBulb)* **if** *TRUE*"

"*SwUp(rSw)* **causes** *On(rBulb)* **if** *TRUE*"

are instances of the schema

"*SwUp(x_1)* **causes** *On*($h_1(x_1)$) **if** *TRUE*"

(flipping up switch c_1 causes its light bulb, $h_1(c_1)$, to turn on. Note that h_1 is a function of the action's parameters, which does not change over time). *SL-H* includes related object propositions, which specify these functions: $\{h_j(\bar{d}) = d' \mid d_i \in$ *Obj*$, d' \in$ *Obj*$\cup \perp\}$. $\perp$ means 'undefined'. Every *relobjs* $\in$ *RelatedObjs* completely specifies those functions.

Other Possible Extensions:

Extended Hidden Objects: in *SL-H*, the hidden objects depended only on the action's parameters. We add to the language new functions, that can depend on the quantified variables as well. We add their specifications to *RelatedObjs*. (example schema: *OpenAll* causes all the doors for which we have a key to open)

Invented Predicates: sometimes the representation of the world does not enable us to learn the transition model. For example, consider a Block-world with predicates *On(x,y),Has(x)*; this suffices for describing any world state, but we cannot learn the precondition of *Take(x)*: it involves universal quantification, $\forall y.\neg$*On(y,x)*. If we add a predicate *Clear(x)*, it is easy to express *all* of the transition rules (including those that affect *Clear*) in our language. This idea is similar to the ones used in Constructive Induction and Predicate Invention (Muggleton and Buntine, 1988).

We can also combine the languages mentioned above. For example, *SL-VH* allows both variables and hidden objects.

3.4.5 Extended Example

After presenting our base languages, we take a closer look at the way we update our circuit.

Imagine a room with two switches, one on the left wall and one on the right. In some other room there are lightbulbs which these switches control; however, the agent does not know which switch controls which bulb. Assume the agent decides to flip the left switch.

In order to update the circuit, we need to construct a next-state formula for every possibly-affected fluent. Take the state of left lightbulb, for example. The idea is to find all ground instances that can affect the lightbulb, and find their matching schemas.

Figure 3.7 shows how to update the current state of the left lightbulb, $expl_{On(lBulb)}$, after executing *SwUp(lSw)*. The circuit in Figure 3.7 *is* the formula $expl'_{On(lBulb)}$ (after update). The node labeled "expl" is the root of the circuit before the update. The bottom nodes (the leaves) are the propositions: This is a simplified example, so we only show two *relobjs* nodes– (*p1,p2*), and two schemas– *tr1,tr2*. *tr1* claims that switching up an object x causes its hidden object, $h(x)$ to become on. *tr2* claims that it turns off everything that is currently on. *p1,p2* relate the left switch with the left and right light bulbs, respectively.

The $\rightarrow$ nodes (second layer) correspond to different cases of *relobjs*. The $\vee$ node is the explanation of *On(lBulb)* in case *p1* holds. Its left branch describes the case that the action caused the fluent to hold– *tr1* is true, and its preconditions hold; the right branch deals with the case that *On(lBulb)* held before the action, and the action did not change it (that is, either *tr2* is false, or its precondition does not hold). The formula in Figure 3.7 can be simplified, but this is an optimization that is not needed for our algorithm.

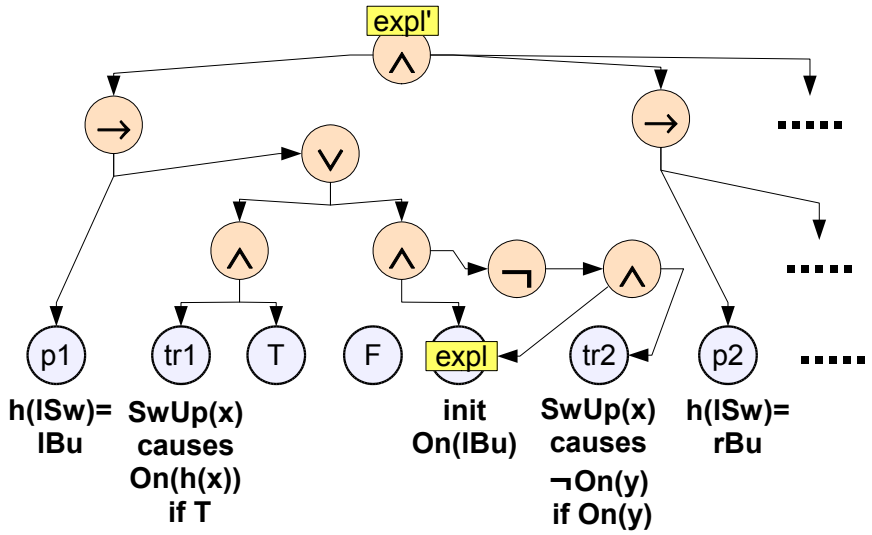


Figure 3.7: A (simplified) update of *On(lBulb)* after the first action, *SwUp(lSw)*

In general, given *relobjs*, a ground action $a(\bar{c})$ and a fluent f, we want to update $expl_f$ and *cb*. To do this, we first identify the instances that can affect fluent f, and the schemas they correspond to.

Denote by (*sch+*,*inst+*) a schema-instance pair that can cause f. *inst+* is a transition rule with effect f, which is an instance of schema *sch+*. In other words– if the schema is true in our domain, and the precondition of the instance (*prec(inst)*) holds, f will hold after the action. Similarly, (*sch-*,*inst-*) is a pair that can cause $\neg f$. We need *relobjs* and $a(\bar{c})$ to match schemas and instances.

Fluent f is true after the action if either (1) a schema *sch+* is true and the precondition of its instance holds, or (2) f holds, and for every schema *sch-* that is true, no precondition holds. *cb* asserts that the action was possible; it cannot be the case that there are two schema-instance pairs, such that their effects are f and $\neg f$, and both preconditions hold.

We assume that the sequence consists of possible actions; if the agent has a way to know whether the action was possible, we do not need this assumption.

3.5 Analysis

3.5.1 Correctness and complexity

Theorem 3.5.1 *The algorithms we presented are correct. For any formula φ and a sequence of actions and observations,*

$$\{\langle s, R\rangle \textit{ that satisfy STRIPS-C-SLAF}(\langle a_i, o_i\rangle_{0<i<\leq t}, \varphi)\} = \mathcal{SLAF}[\langle a_i, o_i\rangle_{0<i<\leq t}](\{\langle s, R\rangle \textit{ that satisfy } \varphi\})$$

$$\{\langle s, R\rangle \textit{ that satisfy C-SLAF}(\langle a_i, o_i\rangle_{0<i<\leq t}, \varphi)\} = \mathcal{SLAF}[\langle a_i, o_i\rangle_{0<i<\leq t}](\{\langle s, R\rangle \textit{ that satisfy } \varphi\})$$

$\{\langle s, R\rangle$ *that satisfy C-SLAFS* $(\langle a_i(\bar{c}_i), o_i\rangle_{0<i<\leq t}, \varphi)\} = \mathcal{SLAFS}[\langle a_i(\bar{c}_i), o_i\rangle_{0<i<\leq t}](\{\langle s, R\rangle$ *that satisfy* $\varphi\})$.

[such that the domain description can be expressed in the corresponding transition relation language]

PROOF OVERVIEW The proof is very similar to the Filtering proof in Section 2.5.1. We define an effect model for action $a(\bar{c})$ at time t, $T_{\mathit{eff}}(a(\bar{c}), t)$, which is a logical formula consisting of Situation-Calculus-like axioms (Reiter, 2001). It describes the ways in which performing the action at time t affects the world. We then show that $\mathcal{SLAFS}[a(\bar{c})](\varphi)$ is equivalent so consequence finding in a restricted language of $\varphi \wedge T_{\mathit{eff}}(a(\bar{c}), t)$. Consequence finding can be done by resolving all fluents that are not in the language; we show that *C-SLAFS* calculates exactly those consequences.

COMPLEXITY The complexity of these algorithms depends on the expressivity of the transition relations we consider. The general time-space complexity for a sequence of length t is the size of the initial knowledge + total size of observations + (t times the size of the next-state and action-possible formulas). That is because we start with a circuit that represents the initial knowledge, at any time step we add an observation, an assertion that the action was possible, and next-state formulas to update our fluents.

STRIPS-C-SLAF

Theorem 3.5.2 *The space and time complexities of the algorithm are $O(|\varphi_0| + |Obs| + t|P|)$, where φ_0 is the initial belief state, $|Obs|$ is the total length of the observations throughout the sequence (can be omitted if observations are always conjunctions of literals).*

C-SLAF

Theorem 3.5.3 *The space and time complexities of the algorithm are $O(|\varphi_0| + |Obs| + tk(2|P|)^{k+1})$, where φ_0 is the initial belief state, $|Obs|$ is the total length of the observations throughout the sequence (can be omitted if observations are always conjunctions of literals), t is the length of the sequence, and k is a parameter of the domain- the minimum number such that preconditions are k-DNF.*

If there are no preconditions (always executable actions), we can maintain a flat formula (instead of a circuit) with complexity $O(|\varphi_0| + |Obs| + t|P|)$.

C-SLAFS Let φ_0 be the initial belief state, $|Obs|$ the total length of the observations (if observations are always conjunctions of literals, we can omit it), t is the length of the sequence. The maximal precondition length, k, is at most $\min\{k' \mid$ preconditions are k'-DNF$\}$. Let *pairs* be the maximal number of schema-instance pairs for an action $a(\bar{c})$. Let r_a, r_p be the maximal arities of actions and predicates, respectively.

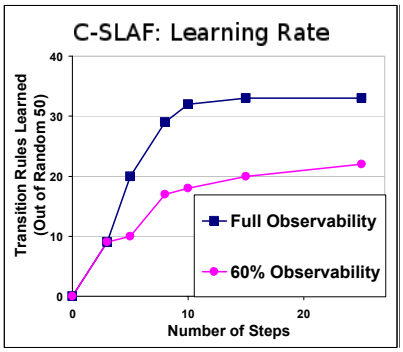


Figure 3.8: Experimental results showing learning rates for the *C-SLAF* algorithm under different degrees of observability (the percentage of randomly selected fluents that are observed at each step).

Theorem 3.5.4 *With circuit implementation, C-SLAFS 's time and space (formula size) complexities are* $O(|\varphi_0| + |Obs| + t \cdot k \cdot pairs)$ *We can maintain an NNF-circuit (no negation nodes) with the same complexity.*

If we allow preprocessing (allocating space for the leafs): In SL_0, $pairs = (2|Pred| \cdot {r_a}^{r_p})^{k+1}$. In *SL-H* with f functions, $pairs = |RelatedObjs|(2|Pred| \cdot (r_a + f)^{r_p})^{k+1}$. In *SL-V* without existential quantifiers, $|P|^{r_p} \cdot (2|Pred| \cdot (r_a + r_p)^{r_p})^{k+1}$, and with them- $|P|^{(k+1)r_p} \cdot (2|Pred| \cdot (r_a + (k+1)r_p)^{r_p})^{k+1}$. If we add invented predicates, we increase $|Pred|$ accordingly.

Since r_a, r_p and k are usually small, this is tractable.

Interestingly, SL_0 (and some cases of *SL-H*) allow runtime that *does not* depend on the domain size (requiring a slightly different implementation). Importantly, SL_0 includes STRIPS.

If there are no preconditions (always executable actions), we can maintain a flat formula with the same complexity.

Note: The inference on the resulting circuit is difficult (SAT with $|P|$ variables). The related problem of temporal projection is coNP-hard when the initial state is not fully known.

3.6 Experimental Evaluation

We implemented two versions of our $\mathcal{SLAF}$ meta-algorithm: *C-SLAF* , for the ground language, and *C-SLAFS* , for the relational case.

C-SLAF

We tested our ground algorithm, *C-SLAF* (Section 3.4.2), on several domains. These domains include Blocks world, Briefcase world, a variation of the Safe world, Bomb-in-toilet world, and the Driverlog domain from the 2002 International Planning Competition [3]. In the variation of the Safe world domain tested, the agent must try a number of

[3] http://planning.cis.strath.ac.uk/competition/

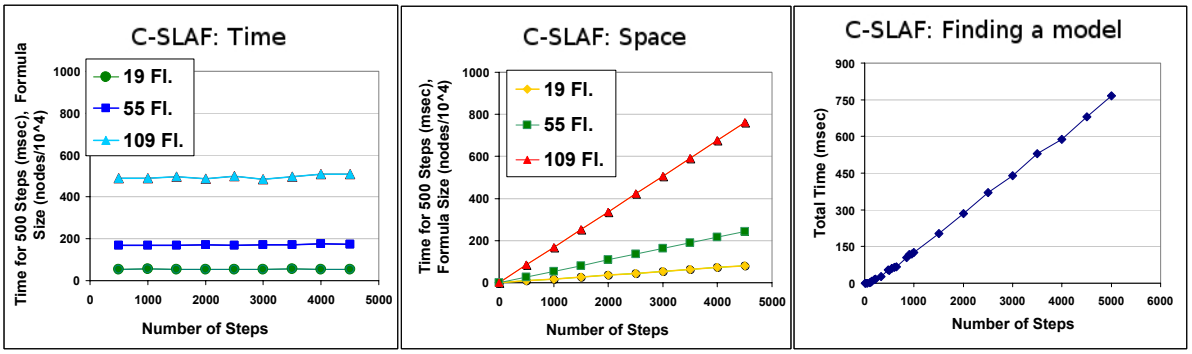


Figure 3.9: Experimental results for the *C-SLAF* algorithm showing time required to process actions, space required to process actions, and time required to extract an action model.

different possible combinations in order to open a safe. In addition, there is an action `CHANGE` which permutes the right combination. Note that many of these domains (Briefcase world, Safe world, Bomb-in- toilet world) feature conditional actions, which cannot be handled by previous learning algorithms.

Figure 3.9 shows the time (always in milliseconds) and space results for instances of Block World of various sizes. The time plots show the time required to process every 500 actions. Note, that *C-SLAF* requires less than 2 milliseconds per action. The space plots show the total space needed in bytes. The rightmost graph shows the time taken by our inference procedure to find an action model of the Blocksworld domain (19 fluents) from the formula produced by *C-SLAF* . Finding an action model is accomplished by simply finding a model of the logical formula produced by the algorithm using our inference procedure; inference times are generally longer for more complex queries.

```
(CHANGE) causes (RIGHT COM2)
    if (RIGHT COM1)
(CHANGE) causes (RIGHT COM3)
    if (RIGHT COM2)
(CHANGE) causes (NOT (RIGHT COM2))
    if (RIGHT COM2)
(CLOSE) causes (NOT (SAFE-OPEN))
    if (TRUE)
(TRY COM1) causes (SAFE-OPEN)
    if (RIGHT COM1)
```

Figure 3.10: A Model of the Safe World (after 20 steps)

Figure 3.10 shows part of an action model learned after running the *C-SLAF* algorithm on the Safe world domain for 20 steps. The model shown was the first model extracted by our inference procedure on the logical formula returned by the algorithm. In this particular model, the algorithm was able to partially learn how the correct combination is permuted after the `CHANGE` action executes.

The learning rate graph in Figure 3.8 show learning rates for *C-SLAF* . The graph shows the number of transition rules (corresponding to action propositions) successfully learned out of 50 randomly selected rules from the domain. Initially, the algorithm has *no* knowledge of the state of the world or the action model. The first graph shows results for the Safe world domain containing 20 fluents and the latter graph shows results for the Driverlog domain containing 15 fluents. Note that in the domains tested, it is not possible for any learning algorithm to completely learn how every action affects every fluent. This is because in these domains certain fluents never change, and thus it is impossible to learn how these fluents are affected by any action. Nevertheless, our algorithms demonstrate relatively fast learning rate under different degrees of partial observability. Unsurprisingly, the results show that learning rate increased as the degree of observability increased. It is also worth noting that inference time generally decreased as the degree of observability increased.

Our algorithm finds exactly all action models that are consistent with the actions and observations, while the algorithm of (Qiang Yang and Jiang, 2005) only guesses a (possibly incorrect) action model heuristically. The goal of this chapter is to perform exact learning where there is no assumed probabilistic prior on the set of action models. In this case, there is no principled way to favor one possible action model over another. However, introducing bias by preference between models was studied by the non-monotonic reasoning community, and can be applied here as well.

C-SLAFS

We implemented and tested our relational algorithm, *C-SLAFS* (Section 3.4.3), for SL_0 and for a variant of *SL-V*; we also implemented a version for the case of always-executable actions, which returns a flat formula. In addition, we implemented a DPLL SAT-search algorithm for circuits (*C-DPLL* , described in the previous chapter). It finds satisfying assignments for the algorithm's output. We tested the $\mathcal{SLAFS}$ algorithms on randomly generated partially observable action sequences, including STRIPS domains (Block-Worlds, Chess, Driverlog), and ADL, PDDL domains (Briefcase, Bomb-In-Toilet, Safe, Grid) of various size, ranging from tens to thousands of propositional fluents.

Figures 4, 3.13 present some of our results. We measured the time, space, *knowledge rate* (percentage of schemas that were learned, out of all the schemas currently in the knowledge base), and *learning rate* (percentage of schemas that were learned, out of all of the schemas that could have been learned from the given sequence). A schema is *learned*, if all models assign the same truth value to it.

As expected, the algorithm takes linear time and space in the sequence length, and does not depend on the domain's size (Figure 3.11). Importantly, simple SAT queries return relatively fast, especially regarding schemas which were contradicted by the sequence. Naturally, more complicated queries take longer time.

Decreasing the number of observations also resulted in long inference time: in the Bomb-In-Toilet domain, we generated several action-observation sequences, with different degrees of observability (from full observability to 10%

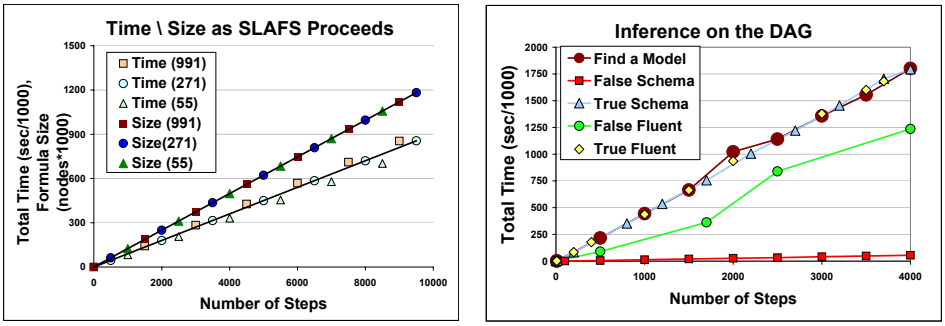


Figure 3.11: Left: Time and space for several Block-Worlds (numbers represent $|P|$). As can be seen, the time and space do not depend on the size of the domain. Slight time differences are due to a hash table. Right: Inference time on *C-SLAFS* ' output, for several simple queries.

of the fluents). Then, we then chose 40 transition rules at random, and checked how many of them were learned for each sequence. Not surprisingly, both learning and inference were faster when the number of observations was higher. If there are no observations we can still eliminate some models, since we know that the actions were possible.

Another important point is that, most of the schemas that can be learned are learned very quickly, even for larger domains. In most domains, more than 98% of schemas were learned after 200 steps (Figure 3.13). This is mainly because the number of action schemas *does not* depend on the size of the domain, e.g. all Block-Worlds have exactly the same number of schemas. Compare this with the decreasing knowledge rate in the propositional approach of (Amir, 2005). The latter does not generalize across instances, and the number of encountered (propositional) transition rules grows faster than those that are learned.

```
(dunk ?bomb ?toilet) causes (NOT (armed
?bomb)) if (NOT (clogged ?toilet))
(dunk ?bomb ?toilet) causes (clogged
?toilet) if (TRUE)
(dunk ?bomb ?toilet) causes (toilet
?toilet) if (TRUE)
(flush ?toilet) causes (not (clogged
?toilet)) if (TRUE)
-----------------
(dunk ?bomb ?toilet) causes (NOT (armed
?bomb)) if (AND (bomb ?bomb) (toilet
?toilet) (NOT (clogged ?toilet)))
(dunk ?bomb ?toilet) causes (clogged
?toilet) if (TRUE)
(flush ?toilet) causes (not (clogged
?toilet)) if (toilet ?toilet)
```

Figure 3.12: Possible Models of the Bomb-Toilet World (Top: after 5 steps. Bottom: after 20 steps)

In another comparison, we ran sequences from (Wu et al., 2005), and each one took our algorithm a fraction of a second to process. We also cross-validated our output and the output of (Wu et al., 2005) with a known model. We

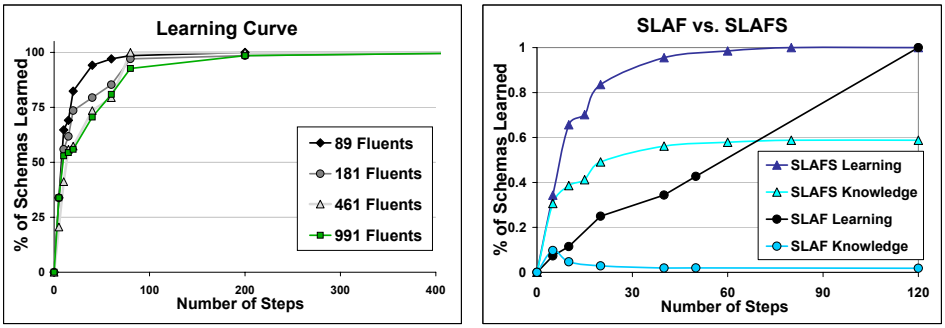


Figure 3.13: Block Worlds. Left: $\mathcal{SLAFS}$ learning rate. Right: $\mathcal{SLAF}$ (Amir, 2005) and $\mathcal{SLAFS}$ knowledge and learning rates.

found that several outputs of (Wu et al., 2005) were inconsistent with the action-observation sequence, while the true model was consistent with our final transition belief state.

Note that their algorithm returns one (approximate) model, whereas our algorithm return a formula that represents all consistent models. Figure 3.12 shows two models of Bomb-In-Toilet world. Those models were found by running our DPLL algorithm on the resulting circuit after 5 and 20 steps, respectively, and returning the first satisfying assignment. The second model (20 steps) is more refined than the first one, and is quite close to the real model.
Trying a schema language that is too weak for the model (for example, trying SL_0 for the Briefcase World) resulted in no models, eventually.

3.7 Conclusions

We presented an approach for learning action schemas in partially observable domains. The contributions of our work are a formalization of the problem, the schema languages, and the tractable algorithm. Our results compare favorably with previous work, and we expect to apply and specialize them to agents in text-based adventure games, active Web crawling agents, and extensions of semantic Web services.

Significantly, our approach is a natural bridge between machine learning and logical knowledge representation. It shows how learning can be seen as logical reasoning in commonsense domains of interest to the KR community. It further shows how restrictions on one's knowledge representation language give rise to efficient learning algorithms into that language. Its use of logical inference techniques (especially resolution theorem proving served in proving correctness of our theorems) and knowledge representation techniques makes it applicable to populating commonsense knowledge bases automatically.

Chapter 4

Applications

Filtering and Learning can be useful for many applications in partially observable domains. In this section we focus on two of them, Conformant Planning and Bounded Model Checking; we demonstrate how to solve them using our circuit representation, and evaluate our solution. We believe our methods can be useful in many other domains.

4.1 Conformant Planning

Assume that you have a robot in a corridor (see Figure 4.1). The robot can move left and right. If it is in front of a door, it can attempt opening, closing or locking it. The robot cannot open a locked door. Also, trying to lock an open door will damage it.

The robot's goal is to lock all doors in the corridor. It might have some partial knowledge about its initial state, but it has no sensors. In Conformant Planning (CP), the robot tries to find a sequence of actions that guarantees it reaches its goal despite this uncertainty.

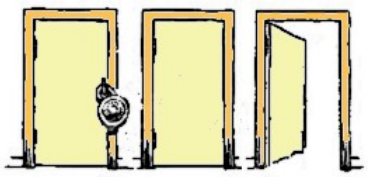


Figure 4.1: A corridor: doors can be opened, closed, or locked.

One possible solution is to first go left until it is sure it reached the end of the corridor (in Figure 4.1, that would mean going left twice). Then, it should apply ⟨*Close*,*Lock*,*Move-Right*⟩ until all doors are locked.

Formally, a planning problem is a tuple $\langle I,G,D\rangle$ in which D is a domain description, $I \subseteq S$ is a non-empty initial set of world states and $G \subseteq S$ is a set of goal world states. A solution for $\langle I, G, D\rangle$ is a plan π (a sequence of actions, no observations) such that performing π from a state in I necessarily results in a state in G. In other words, $\mathit{Filter}[\pi](I) \subseteq G$.

In Conformant Planning we assume that we start in a world state in I, but we do not know which world in I it is. There, planning includes no sensing actions, no automatically sensed conditions, and no branching on conditions.

PROCEDURE ***Forward-Search*** (I, G, D)
I, G initial belief-state and goal formulas, D domain description

1: ***ProcessDomain*** (D)
2: $c_I :=$ ***CreateCircuit*** $(True)$
3: c_I.ActionSequence= $\emptyset$
4: Queue.insert(c_I)
5: **while** NOT Queue.empty() **do**
6: $c :=$Queue.pop()
7: **if** ***Goal*** (c, I, G) **then**
8: **return** c.ActionSequence
9: **else**
10: **for** $a \in A$ **do**
11: $c' :=$***ProgressAction*** (c, a)
12: c'.ActionSequence= $\langle c$.ActionSequence, $a\rangle$
13: Queue.insert(c')

PROCEDURE ***Currently***(c, ψ)
$\{c$ a belief-state circuit, ψ a formula$\}$
Replace fluents with current values
1: **for** $f \in P$ in ψ **do** replace f with the node pointed by $c.expl_f$
2: **return** ψ

PROCEDURE ***Initially***(c, ψ)
$\{c$ a belief-state circuit, ψ a formula$\}$
Replace fluents with leaves
1: **for** $f \in P$ in ψ **do** replace f with the leaf $c.init_f$
2: **return** ψ

PROCEDURE ***CreateCircuit*** (φ)
φ formula

1: c an empty circuit
2: **for** $f \in P$ **do**
3: c.AddNewLeaf(init$_f$)
4: $c.expl_f :=$ init$_f$
5: $c.cb :=$ ***Currently***(c, φ)
6: **return** c

PROCEDURE ***Goal*** (c, I, G)
c a belief-state circuit, I, G initial-state and goal formulas

1: **return** UNSAT(***Initially*** $(c, I) \wedge\neg($$c.cb \wedge$ ***Currently*** $(c, G)))$

Figure 4.2: Forward-Search Algorithm

We can use *C-Filter* and *C-DPLL* to forward search for the goal, starting from the initial belief state. At every step we can check if the goal conditions are satisfied (entailed) by the current belief state. If they do, we stop and return the plan that satisfies this projection. The attractive nature of this approach is that we can apply standard planning techniques without fear of reaching belief states that are too large to represent.

Algorithm Overview

The template of forward-search is shown in Figure 4.2. The algorithm receives an initial belief-state formula I, goal formula G. and a domain description D. D includes the description of actions in A; an action has a precondition and a list of conditional effects. *Every* deterministic domain can be represented like this (including, for example,

STRIPS and PDDL); see Section 3.4.2. The precondition has to hold before executing the action, and the conditional effects occur iff their condition holds. In our example,

Move-Left Precondition: *True*

"**Causes** *(at pos1)* $\wedge\neg$*(at pos2)* **if** *(at pos2)*"

"**Causes** *(at pos2)* $\wedge\neg$*(at pos3)* **if** *(at pos3)*"

Close Precondition: *True*

"**Causes** $\neg$*(open door1)* **if** *(at pos1)*"

"**Causes** $\neg$*(open door2)* **if** *(at pos2)*"

"**Causes** $\neg$*(open door3)* **if** *(at pos3)*"

Lock Precondition: $[$*(at pos1)* $\wedge$ $\neg$*(open door1)*$]$ $\vee$ $[$*(at pos2)* $\wedge$ $\neg$*(open door2)*$]$ $\vee$ $[$*(at pos3)* $\wedge$ $\neg$*(open door3)*$]$

"**Causes** *(locked door1)* **if** *(at pos1)*"

"**Causes** *(locked door2)* **if** *(at pos2)*"

...

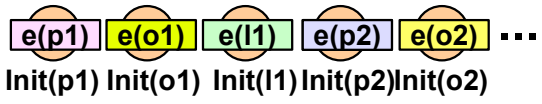
The algorithm starts by processing the domain description; using *C-Filter* subroutines, it extracts *next-state* and *possible* formulas. *Poss(a)* is the precondition of action a, and *NextVal(a,f)* is the value of fluent f after action a as a function of the world before it (intuitively, the formula "a caused f or f already held and a did not change it"). Note that we only calculate this for fluents that might have been affected by a; the circuit structure obviates the need for specifying that other fluents do not change (Frame axioms). For example,

$$\begin{aligned} &\textit{NextVal(Close,(open door1))} = \textit{(open door1)} \wedge \neg\textit{(at pos1)} \\ &\textit{NextVal(Open,(open door1))} = \\ &\quad \textit{[(at pos1)} \wedge \neg\textit{(locked door1)]} \vee \textit{(open door1)} \end{aligned} \tag{4.1}$$

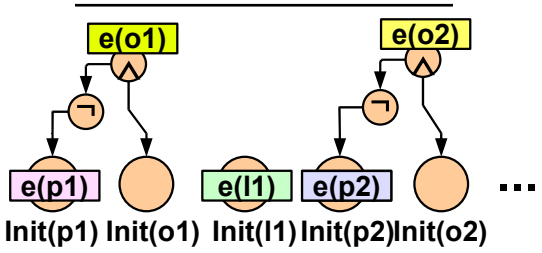
The algorithm maintains a set of alive belief-states circuits in a priority queue, for which a priority function must be specified (this function is the most significant difference between various search algorithms). The circuits correspond to action sequences: this is similar to (Hoffmann and Brafman, 2004), that represent a belief state by the initial belief state representation and the action sequence that leads to it.

In every iteration a candidate is extracted from the queue and checked. If it is a solution sequence (starting from I entails that the sequence is applicable and results in G), the algorithm returns the associated sequence. Otherwise, the algorithm applies every action $a \in A$, builds the updated circuits and stores them in the queue.

The update step is the one used in *C-Filter* . We do not use observations, since they are not included in the Conformant model.



(a) A belief-state update: action sequence ⟨⟩



(b) A belief-state update: action sequence ⟨Close⟩

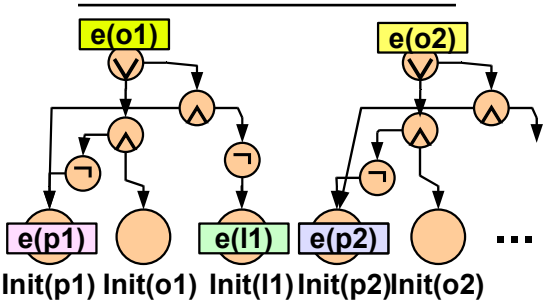


Figure 4.3: (c) A belief-state update: action sequence ⟨Close, Open⟩

The update is illustrated in Figure 4.3. 4.3a shows a belief-state that corresponds to the empty sequence, the result of ***CreateCircuit*** (True): the circuit contains a node for each fluent f, and the pointer *e(f)* is set to it. *cb* is set to *True* (not shown in the figure).

4.3b shows how we progress with the action *Close*. The action is always applicable (no preconditions), so *cb* does not change. The only fluents that can be affected by this action are *(open door1)*, *(open door2)*, *(open door3)*. We progress each one of them, by building the NextVal formulas (see Equation 4.1). After building the formula that explains the current state of door1, we assign the pointer to the root of this formula (similarly for the other doors).

4.3c goes one step further, and progresses 4.3b with another action, *Close*. Note that we update the only possibly-affected fluents (no Frame axioms). Also, when we refer to fluents of the previous time step we use the internal node constructed in the 4.3b; this allows recursive sharing of subformulas, resulting in compactness.

We showed how to create belief-states that correspond to action sequences. We still need to determine solution sequences: in Procedure ***Goal*** (Figure 4.2) we look for a counterexample, an initial state in I for which the sequence is either not applicable or does not result in G. We construct this formula (leaves represent initial state, *cb* is applicability, and the explanation pointers represent final state), and look for a satisfying assignment.

4.1.1 Analysis

Theorem 4.1.1 (Correctness) *The algorithm is sound. If the queue implements systematic search, it is also complete.*

Theorem 4.1.2 (Complexity) *Let ActDesc be the length of the longest action-description in domain D (the length of the precondition and conditional effects). Building a query circuit that corresponds to an action-sequence of length k, initial belief-state I and goal G takes time $O(|I| + |G| + k \cdot ActDesc)$. Its output is a circuit of the same size, with at most $|P|$ variables.*

Note that size *does not* depend on the domain size, $|P|$. *ActDesc* is usually small– especially if the actions in the domain affect a small number of fluents, and have simple preconditions. A flat formula, on the other hand, will be of size $O(CNF(|I|) + CNF(|G|) + k \cdot (|P| + CNF(ActDesc)))$, and will involve $k|P|$ variables. BDDs cannot guarantee compactness at all, and sometimes blow up exponentially. The proofs follow from the correctness of the *C-Filter* algorithm and from (LaValle, 2006).

4.1.2 Extensions

We have described our representation and basic algorithm. In this section we extend them in several ways to handle non-determinism, backward search and parallel actions.

Non-Determinism

Many real-life environments are inherently non-deterministic. However, the circuit-representation update step relies heavily on determinism (specifying the state of each fluent as a function of the previous step). Please recall Section 2.5.5 for ways to make the Filtering algorithm handle non-determinism; each of these ways can be used for Conformant Planning as well.

Backward Search

A backward version of the forward search algorithm can be made. The algorithm is very similar, but actions are added at the beginning of the sequence. To achieve this, the update-step builds *NextVal* and *Poss* formulas as before, but adds them to the bottom of the circuit instead (creating new leaves).

Action Fluents

Instead of building circuits that correspond to one action sequence, we can add propositions representing actions and construct a circuit that represents all sequences of length k. The algorithm is similar, but the *NextVal* and *Poss* formulas

should now take those variables into account. We can take advantage of many encodings, including allowing actions to execute in parallel.

Reasoning with those circuits is different: we now need to answer an $\exists$*sequence*$\forall$*initial-state*-query, instead of $\forall$*initial-state* like previously. We first look for a possible plan (satisfies to "***Initially*** $(c, I) \wedge c.cb \wedge$ ***Currently*** (c, G)"). We take the (perhaps partial) assignment to the action fluents, and check (the same way we did before) every action-sequence assignment that agrees with it.

4.1.3 Optimizations

In this section we discuss the efficiency of our search method, and present some domain-independent optimizations we implemented.

Reuse

Two main sources of inefficiency in our algorithm are the independence of satisfiability checks (resulting in learning things over and over) and building circuits from scratch every time. To solve this we first implemented an "undo" command, so that some search procedures (e.g. DFS) can operate on a single data structure. In addition, whenever a SAT check determined the value of a fluent it replaces it with the value, preventing future checks. As a special case of this, if an action sequence is not executable the *cb* formula becomes *False*, and no sequences that start with this prefix are checked.

Search Heuristics

One of the most important parts in our searching algorithm the queue sorting function. Much research was spent on using heuristics to guide the CP search (e.g. (Bonet and Geffner, 2000; Bertoli et al., 2001a)). Many of the results can be applied with small changes to our circuits.

We describe three heuristics which we found most effective: one estimated the belief-state cardinality by sampling. We prefer small, high-informed belief-states when performing forward search and big belief-states when proceeding backward. The second heuristics also estimates the level of knowledge associated with the belief-state by checking how many propositions are known in it. This requires $|P| + 1$ SAT checks, but seems to guide the search well.

The final heuristic split the goal into subgoals. The distance estimation was the number of subgoals that were not entailed by the current belief state, divided by the number that can be reached in a single step (estimating the minimal number of steps to the goal). We added a random ordering on subgoals to break ties. Later we refined this heuristic to take into account preconditions of subgoals that are entailed. This heuristics worked well for domains in which one knows when a subgoal was achieved (i.e. Bomb in Toilet), but breaks in others (i.e. Ring domain).

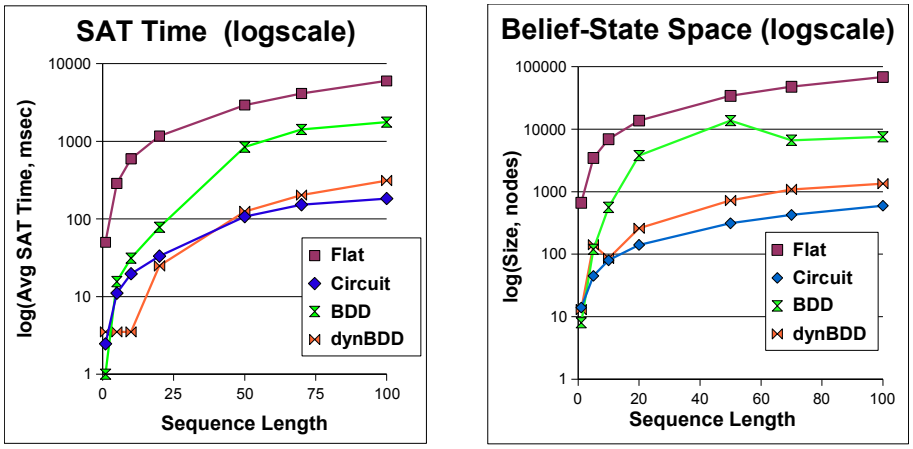


Figure 4.4: Comparison of flat formulas, BDDs, dynamic-reorder BDDs and circuits for various sequences in Block-World variant with $\sim$ 100 fluents. Time for SAT is on the left (msec), Space (nodes) on the right; both are logscale.

SAT Heuristics

Like CNF-SAT, the efficiency of the circuit-SAT solver depends heavily on the variable-instantiation order. We experimented with several heuristics, among those choosing the variable that is closest/farthest from root, ZChaff's VSIDS heuristic (which counts the number of conflicts that a variable causes) , and the most-effective variable (in the sense of flow algorithms).

Bombs	*Circuit*	*CMBP*	*CGP*	*GPT*
Time, sec (deterministic)				
2	0	0	0	0.074
5	0.09	0.02	0.13	0.094
8	0.53	0.15	13.69	0.288
10	1.26	0.71	157.6	1.309
16	11.57	99.2	TO	351.4
Time, sec (non-det)				
2	0	0	–	0.076
5	0.109	0.01	–	0.098
8	0.578	0.17	–	0.38
10	1.484	0.72	–	1.828
16	159.06	98.27	–	486.25

Table 4.1: Comparison: Conformant Planners in Bomb-in-Toilet domains (sec)

We noticed a tradeoff: "closest" and "farthest" instantiated the most variables (i.e. did not choose good ones), but since they are both logarithmic in the size of the circuit their overall time was better than the others. This situation changed as action sequences became longer; ZChaff's heuristic outperformed the others on our longest sequences.

4.1.4 Preliminary Experimental Results

We implemented our method and tested it empirically. Our main goal was to evaluate the space-time tradeoff of our representation compared to other methods. In order to eliminate as many sources of bias as possible, our algorithm was designed to work with four representations: BDDs, dynamic-reordering BDDs, flat formulas and circuits. Furthermore, to avoid different SAT-solvers we encoded CNF formulas as shallow circuits and applied the same circuit-solver to all.

The representation is (almost) orthogonal to the search method and the heuristics; therefore, we believe that those experiments can reveal the potential of our method. Figure 4.4 shows some of the results in a Block-World variant with ~ 100 fluents: the left part shows the average time for SAT queries for action-sequences of different length, and the right part shows the corresponding representation size (number of nodes). Note that both graphs are logscale.

As can be seen, our representation is the most compact, and is not subject to oscillations like the BDD. In some bigger domains and longer sequences the BDD algorithm could not even represent the initial belief-state. Our algorithm is also faster than the flat-formula SAT; dynamic-reordering BDD is comparable (faster at the beginning, but does not scale as well). This makes sense, since BDDs do not need SAT inference, but tend to grow with long sequences.

Figure 4.4 also shows some comparisons of our naive algorithm to times reported by other algorithms on Bomb-In-Toilet deterministic and non-deterministic domains. Despite being naive, it is comparable with them some of the time and even outperforms them. We believe that incorporating techniques from other CP solvers into our solver (such as pruning, or fast-forward search (Hoffmann and Brafman, 2004)) could result in a strong CP algorithm.

4.1.5 Conclusions

We have presented algorithms for solving Conformant Planning via a novel logical circuit representation. This representation is better than other representations is some ways and worse in others. Thus, we strike a different point in the tradeoff. Unlike enumeration or BDDs, this representation guarantees compactness; unlike flat logical formulas, it uses only $|P|$ variables and keeps more structural information.

We showed how to take advantage of the benefits of our representation and how to minimize the effect of its drawbacks. We analyzed our method experimentally and analytically, and proposed several extensions and domain-independent optimizations.

Finally, we showed that our representation outperforms others. Most importantly, our algorithm could handle large domains and long sequences that other (BDD-based) algorithms could not process. We expect that more sophisticated search and analysis techniques for planning would yield significant improvements when combined with our method.

4.2 Formal Verification

Testing alone cannot prove that a system does not have a certain defect, or that it has a certain property (unless all possible states are tested). In contrast, Formal Verification is the act of proving or disproving such properties, using formal methods of mathematics.

In Formal Verification we are given a formal model of a system and check whether the system satisfies some property, usually specified in temporal logic (e.g., a bad state is not reachable). In Bounded Model Checking (BMC), an initial belief state is given, and we check whether the system satisfies the property after at most k time steps.

The connection of Filtering to Bounded Model Checking is straightforward. If we know how to maintain a belief-state, we can unfold our system for k steps, and then check for the desired property (for example, check that the register is non-zero). If the property does not hold, there is some k for which we can find a counterexample. In fact, the most interesting part in applying Filtering to BMC is handling those temporal-logic queries. We now turn our attention to translating queries into our circuit structure.

4.2.1 Queries

Assume we have constructed the circuit that corresponds to the belief-state after k steps. The next step of the BMC algorithm is finding a counterexample.

To do this, we first show how to translate a Linear Temporal Logic (LTL) formula into a circuit. LTL has many applications; in particular, it is a popular choice for specifying temporal properties in verification domains. We start (Section 4.2.1) with a description of LTL and our translation procedure, ***LTLtoCircuit***. After the translation, finding a counterexample reduces to circuit-satisfiability problem (addressed in Section 2.3.3).

LTL

LTL is a modal temporal logic with modalities referring to time. It can encode formulas about the future of paths such as that a condition is always true, will eventually be true, will be true until another fact becomes true, etc. Formally, an LTL formula is:

Definition 4.2.1 (LTL Formula)

1. *φ a propositional variable is an LTL formula.*
2. *If φ, ψ are LTL formulas then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $N\varphi$ (Next. $X\varphi$ is also used), $G\varphi$ (Globally), $F\varphi$ (Finally), $\psi U \varphi$ (Until), $\psi R \varphi$ (Release).*

For example, $G(x_0 \vee x_1 \vee x_2 \vee x_3)$ specifies that the register never contains only zeros.

Several translations of LTL to SAT have been proposed in the past (Cimatti et al., 2002c; Frisch et al., 2002). Our translation is based on the one presented in (Latvala et al., 2004), since it is the most compact one and can be easily incorporated into our circuit structure. We now describe it in detail.

We first note that properties specified in LTL can require two different types of counterexamples: properties such as "always φ" require finite counterexamples, while others (like "eventually φ") require infinite ones. However, BMC considers only finite paths. In order to handle both types, we note that a finite path can represent an infinite path if it contains a loop. Therefore, for every path of length k, we consider $k+1$ possibilities: the k simple loops (states at time i, k are equal), and the no-loop case.

The basic idea of (Clarke et al., 2001) is to construct a formula for each of these cases, such that a satisfying assignment corresponds to a counterexample. The complete translation simply joins those formulas in a disjunction. (Clarke et al., 2001) have shown that it is enough to use a finite prefix of a path, and case split based on the loops the path contains. If a formula is true in the bounded semantics, it is true is the normal semantics.

Like (Latvala et al., 2004), our translation uses the fact that for lasso-shaped paths, the semantics of LTL and CTL coincide. An LTL formula can therefore be evaluated by a CTL model checker, if we add E quantifiers in front of each temporal operator. Thus, the fixed-point characterization of CTL model checking serves as a starting point for the translation.

We take the circuit generated in earlier steps of the algorithm, the one representing our belief-state after k time steps. Note that unlike the propositional formulas generated in similar approaches, the only variables that the circuit involves are those of time 0. Variables of time $0 < i \leq k$ are internal nodes in the graph. In the following, whenever we refer to p_i, we mean the internal node that represents p at time i.

We generate $k+1$ extensions to the circuit, representing the different loop cases. Each extension is of the form $|[\textit{counterexample}_i]|_0 \wedge \bigwedge_{f \in P}(f_i \leftrightarrow f_k)$, where $|[\textit{counterexample}_i]|_0$ is the translation of the negation of the LTL property for the case of an i-loop. Of course, if f_i and f_k are the same node (that is, if f was not affected between steps i and k), there is no need to add the "$\leftrightarrow$" constraint. The no-loop case formula is only $|[\textit{counterexample}_i]|_0$; we do not require that a loop actually not exist.

The translation of the LTL formula is shown in Figure 4.5. It is defined recursively, and according to the loop case. Again, p_i refers to an internal node; at no stage we need to add variables. The until operator $E(\varphi_1 U \varphi_2)$ and the release operator $E(\varphi_1 R \varphi_2)$ are evaluated by computing the least and greatest fixed-point, respectively. The fixed-points are evaluated by first computing an approximation $\langle\langle\rangle\rangle$ (defined in Figure 4.5). The results of the approximation are used to compute the exact result (see (Latvala et al., 2004) for more details). That translation allows a lot of sharing between different formulas, thus is very suited for our circuit framework.

PROCEDURE ***C-Filter-BMC***$(\langle a_i, o_i\rangle_{0<i\leq k}, \psi_0, \varphi, D)$
{ $\langle a_i, o_i\rangle$: actions-observations, ψ_0: initial belief state over P, φ: LTL formula to verify,
D: domain description}

Preprocess D, φ

1: ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
2: **for** $f \in P$ **do** $expl_f$:= a new proposition f_0
3: cb := ***Time0***(ψ_0)

Process sequence

4: **for** i = 1 **to** t **do**
5: ***ProgressAction***(a_i)
6: ***FilterObservation***(o_i)
7: **return** $cb \wedge$ ***LTLtoCircuit***$(\neg\varphi, k)$

PROCEDURE ***LTLtoCircuit***(φ, k)
{φ an LTL formula, k depth}

Return the translation of φ to a circuit

:=	**i≤k**	**i=k+1 j-1 loop case**	**i=k+1 no-loop case**
$\|[p]\|_i$ (p proposition)	p_i (the internal node representing p at time i)	p_j	*FALSE*
$\|[\neg p]\|_i$	$\neg p_i$	$\neg p_j$	*FALSE*
$\|[N\psi]\|_i$	$\|[\psi]\|_{i+1}$	$\|[\psi]\|_{j+1}$	*FALSE*
$\|[\psi U\varphi]\|_i$	$\|[\varphi]\|_i \vee (\|[\psi]\|_i \wedge \|[\psi U\varphi]\|_{i+1})$	$\langle\langle\psi U\varphi\rangle\rangle_j$	*FALSE*
$\|[\psi R\varphi]\|_i$	$\|[\varphi]\|_i \wedge (\|[\psi]\|_i \vee \|[\psi R\varphi]\|_{i+1})$	$\langle\langle\psi R\varphi\rangle\rangle_j$	*FALSE*
$\|[G\psi]\|_i$	$\|[\psi]\|_i \wedge \|[G\psi]\|_{i+1}$	$\langle\langle G\psi\rangle\rangle_j$	*FALSE*
$\|[F\psi]\|_i$	$\|[\psi]\|_i \vee \|[F\psi]\|_{i+1}$	$\langle\langle F\psi\rangle\rangle_j$	*FALSE*
$\langle\langle\psi U\varphi\rangle\rangle_i$	$\|[\varphi]\|_i \vee (\|[\psi]\|_i \wedge \langle\langle\psi U\varphi\rangle\rangle_{i+1})$	*FALSE*	*FALSE*
$\langle\langle\psi R\varphi\rangle\rangle_i$	$\|[\varphi]\|_i \vee (\|[\psi]\|_i \wedge \langle\langle\psi R\varphi\rangle\rangle_{i+1})$	*TRUE*	*TRUE*
$\langle\langle G\psi\rangle\rangle_i$	$\|[\psi]\|_i \wedge \langle\langle G\psi\rangle\rangle_{i+1}$	*TRUE*	*TRUE*
$\langle\langle F\psi\rangle\rangle_i$	$\|[\psi]\|_i \vee \langle\langle F\psi\rangle\rangle_{i+1}$	*FALSE*	*FALSE*

1: $\psi := |[\varphi]|_0$ { translate φ according to no-loop case. use *History[f]* pointers }
2: **for** i=1...k **do**
3: $\psi := \psi \vee ([\bigwedge_{f\in P} f_{i-1} \leftrightarrow f_k] \wedge |[\varphi]|_0$ { translate φ according to *i-1* loop case}
4: **return** ψ

Figure 4.5: *C-Filter-BMC* Algorithm

The until operator and the release operator are evaluated by computing fixed-points. First, we compute an approximation $\langle\langle\cdot\rangle\rangle_i$ for each state and subformula. The results of the approximation are used to compute the final result $|[\cdot]|_i$.

Note that implementing this translation in (Latvala et al., 2004) required adding new variables to the formula. However, our circuit structure obviates the need for those, as those fluents are represented as internal nodes.

Our algorithm is now complete. We have shown how to unfold the state of the system for k timesteps, and how to construct a query. We now argue for the correctness and compactness of our algorithm; in the next section we discuss the final part of the method, reasoning with the algorithm's output.

Theorem 4.2.2 (Correctness) *Let $\langle P, S, A, R\rangle$ be a transition system, ψ_0 the initial belief-state formula, and φ an LTL formula that we wish to verify. Let k be the model-checking depth, and $\langle a_i, o_i\rangle_{0\leq i<k}$ be a sequence of actions*

sec	*CFilter*	*NuSMV*	*Tip*
eijk.S444.S			
30	38	34	34
180	46	38	39
360	47	39	40
540	48	39	40
720	48	41	41
900	**49**	41	41
eijk.S208.S			
30	23	23	28
180	28	33	36
360	30	39	41
540	32	42	44
720	33	45	46
900	34	47	**48**
eijk.S349.S			
30	16	11	finish
180	26	19	(property
360	29	24	holds)
540	33	28	
720	35	30	
900	**37**	31	

CFilter	*NuSMV*	*Tip*
nusmv.syncarb52.B		
860	73	103
2092	134	142
2927	170	158
3603	195	169
4106	214	184
4581	230	187
eijk.S1238.S		
11	5	finish
66	14	(property
142	21	holds)
204	26	
273	30	
346	34	
texas.ifetch1E		
count. example found	count. example found	count. example found

CFilter	*Tip*	*NuSMV*
nusmv.syncarb102.B		
598	40	101
1452	74	155
2040	94	172
2493	108	181
2866	119	187
3196	128	190
eijk.S1423.S		
8	9	6
10	9	10
10	9	10
11	9	11
11	10	11
11	10	**11**
nusmv counter		
count. example found	count. example found	count. example found

Figure 4.6: Comparison of maximal depth reached vs. time (sec), best results marked in bold. In the last two domains, counterexamples were found in < 30 seconds.

and observations.

The resulting circuit is satisfiable iff there is a path of length at most k satisfying $\neg\varphi$.

Theorem 4.2.3 (Complexity) *1. If the property does not require checking loops (e.g. $G\varphi$), the resulting circuit size is $O(|\psi_0| + |Obs| + k \cdot (|\varphi| + |Act_c|))$, where $|Obs|$ is the total size of observations in the sequence (often 0), and $|Act_c|$ is the longest circuit-representation of an action that appears in the sequence.*

2. Otherwise, the size is $O(|\psi_0| + |Obs| + k \cdot (k|\varphi| + |Act_c| + |P|))$

In both cases, the circuit has $|P|$ variables.

We can also construct a circuit of size $O(|\psi_0| + |Obs| + k \cdot (|\varphi| + |Act_c| + |P|))$ and $|P| + k$ variables for the second case (the extra variables determine the loop).

A typical action is represented as a precondition, and a set of if-then rules that represent conditional effects. For example, the action *negate(x)* (that is, $x :=!x$) has an empty precondition, *TRUE*, and two conditional effects: *negate(x)* causes x if $\neg x$, *negate(x)* causes $\neg x$ if x.

The proofs follow from the correctness and compactness results of (Shahaf and Amir, 2007; Latvala et al., 2004).

4.2.2 Experiments

In order to evaluate our method, we used a benchmark of SMV files available online and compared our algorithm to two state-of-the-art bounded model checkers, Tip (Een and Sorensson, 2003) and nuSMV (Cimatti et al., 2002b).

First, we implemented a utility that translates smv files to our logical format. The resulting files were later used

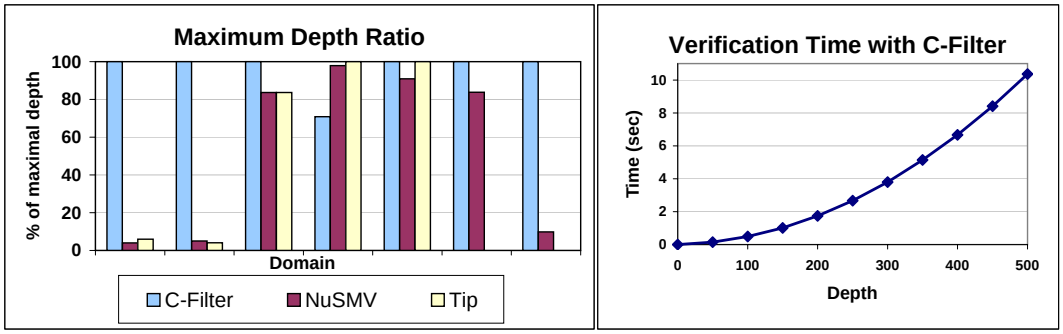


Figure 4.7: Left: for each domain, the percentage of the maximal depth each solver achieved (in the last two, Tip stopped with a proof). Right: Time for Verification (sec) on *nusmv.syncarb52.B* vs. Search depth.

to extract *next-state* and *possible* formulas (Procedure ***ProcessDomain*** in Figure 2.3). Many of those systems are non-deterministic, in the sense that transitions depend on unspecified user input. This was handled as described in Section 2.5.5. The translation took linear time in the size of the input file.

After the translation phase, we ran the comparison experiments. They were run in an iterative-deepening fashion, increasing k every iteration and trying to find a counterexample of length k. Some of the results are shown in Figures 4.6-4.7.

Figure 4.6 compares the maximal depth the solvers reached in a fixed amount of time for several domains. The domains were of varied complexity, so the maximal depths differ a lot between them. The normalized results are also shown in Figure 4.7 (left) for each domain, the maximal depth reached is set to be 100%, and the figure shows how well the model checkers did in comparison. Note that *C-Filter-BMC* outperformed the others on many of those domains. Figure 4.7 (right) shows a sample runtime graph for *C-Filter-BMC*.

The results are interesting: no algorithms seems to dominate the others on *all* domains. However, on most of the domains our algorithm is comparable with the rest of the model checkers, or outperforms them. On several domains (such as *eijk.S444.S*), our algorithm outperforms NuSMV and Tip by orders of magnitude. This is especially impressive since clausal SAT solvers have been studied and optimized for years, while circuit SAT solvers are relatively new. We strongly believe that circuit SAT solvers will improve over time, resulting in better results for our algorithm.

In addition to those experiments, we compared *C-Filter-BMC* to a BDD-based model checker. BDDs are not guaranteed to be compact, and indeed – many times the belief-state was too large for this solver. In some cases, even the BDDs for the initial state could not be constructed, while *C-Filter-BMC* was able to look for counterexamples in large depth.

Discussion It seems that our algorithm does especially well on domains that are mostly deterministic, and involve many don't-care variables (variables whose value is not important for the SAT check). This is the case, for example, when the next-state of fluents depends mostly on the last action, and less on their previous value. CNF

solvers could spend a lot of time branching on variables that have become don't-care due to previous decisions, while it is easier for circuit solvers to avoid those areas. *C-Filter-BMC* also did well on files with many definitions; those are formulas that are shared between many next-state formulas. Our circuit takes full advantage of this sharing.

4.2.3 Conclusions and Future Work

We presented an approach for BMC which limits the number of propositional variables that must be assigned by the SAT solver. It uses a Boolean circuit representation to encode the set of possible states up to the last execution step. This is similar in spirit to the Symbolic Model Checking approach, only with a more space-efficient representation. The new approach applies SAT solvers for our logical-circuit representation of possible states. The result is a BMC-like algorithm that is faster empirically on almost all models, sometimes improving over current techniques by a significant margin. The empirical results are particularly striking because there is much space for optimizing and speeding up SAT solvers for logical circuits.

One important future direction that we plan to pursue is software BMC, e.g., (Clarke et al., 2004). Several features of *C-Filter-BMC* make it promising for software BMC. Transitions for hardware systems usually include only one action, and thus each transition involves all the state variables. In contrast, software verification typically involves many actions, each affecting only few variables (e.g., an action x++ affects only the variable x). *C-Filter-BMC* does not modify the logical circuit where fluents need not change. Thus, we expect software BMC to manipulate formulas whose number gates grows slowly with the number of steps. Finally, many scenarios in software verification are deterministic, thus requiring us to add little or no variables at all to the representation. The minimal growth in variables and gates during processing of *C-Filter-BMC* suggests that it will scale particularly well in software verification applications.

Chapter 5

Conclusions

A straightforward approach to filtering is creating all prime implicates at time $t+1$ from the belief state representation of time t. Previous work (e.g. (Liberatore, 1997b)) showed that deciding if a clause belongs to the new belief state is coNP-complete, even for deterministic domains. This discouraged further research on the problem.

Nevertheless, in this work we presented an exact and tractable filtering algorithm for all deterministic domains. Our result is surprising because it shows that creating a representation of *all* of the consequences at time $t+1$ is easier (poly-time) than creating the new belief state piecemeal.

We extended our method and presented an algorithm for learning action schemas in partially observable domains. The contributions of our work are a formalization of the problem, the schema languages, and the tractable algorithm.

The key to our advance was our *logical circuits* representation. We also showed how to maintain NNF-Circuits. We have implemented those algorithms and tested them in learning, Conformant Planning and Bounded Model Checking domains. Our results compare favorably with previous work, sometimes by orders of magnitude.

Significantly, our approach is a natural bridge between machine learning and logical knowledge representation. It shows how learning can be seen as logical reasoning in commonsense domains of interest to the KR community, and how restrictions on one's representation language give rise to efficient learning algorithms into that language.

Criticism and Future Directions This work presented results for partially observable deterministic domains. We obtained encouraging results for several classes of domains, mainly relational ones with simple action models. However, there is room for improvement.

Firstly, our work is not robust to noise (because of its logical nature). This limits its potential utility. Naïve ways to handle noise will affect the efficiency of the inference.

In addition, we only handled the case of a single agent. For example, assume an agent knows that some switch in another room is up, and then another agent flips it. When the first agent notices that the switch is down, it will reach a contradiction. Extending our methods to handle multi-agent systems could be an interesting direction.

Several other topics need to be addressed: How does observability affect learning? How does the choice of schema language affect it? Also, a more detailed analysis of convergence to the correct underlying model is needed.

References

Amir, E. (2002). Planning with nondeterministic actions and sensing. Technical report, CogRob'02.

Amir, E. (2005). Learning partially observable deterministic action models. In *IJCAI '05*. MK.

Amir, E. and Russell, S. (2003). Logical filtering. In *IJCAI '03*. MK.

Baral, C., Kreinovich, V., and Trejo, R. (2000). Computational complexity of planning and approximate planning in the presence of incompleteness. *AIJ*, 122(1-2):241–267.

Bertoli, P., Cimatti, A., and Roveri, M. (2001a). Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI '01*. MK.

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001b). Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI '01*, pages 473–478. MK.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Biere, A., Cimatti, A., Clarke, E., Fujita, M., and Zhu, Y. (1999). Symbolic model checking using SAT procedures instead of BDDs. In *DAC'99*.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search. In *Proc. AIPS'00*.

Boyen, X., Friedman, N., and Koller, D. (1999). Discovering the hidden structure of complex dynamic systems. In *Proc. UAI '99*, pages 91–100. MK.

Bryant, R. E. (1992). Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002a). NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 359–364. Springer Verlag.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002b). NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV 2002*.

Cimatti, A., Pistore, M., Roveri, M., and Sebastiani, R. (2002c). Improving the encoding of LTL model checking into SAT. In *VMCAI*, pages 196–207.

Cimatti, A. and Roveri, M. (2000). Conformant planning via symbolic model checking. *JAIR*.

Clarke, E., Biere, A., Raimi, R., and Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design*.

Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In *TACAS'04*.

Doucet, A., de Freitas, N., Murphy, K., and Russell, S. (2000). Rao-Blackwellised particle filtering for dynamic bayesian networks. In *Proc. UAI '00*, pages 176–183. MK.

Dzeroski, S. and Luc De Raedt, K. D. (2001). Relational reinforcement learning. *Machine Learning*, 43(1-2):7–52.

Een, N. and Sorensson, N. (2003). Temporal induction by incremental sat solving.

Eiter, T. and Gottlob, G. (1992). On the complexity of propositional knowledge base revision, updates, and counterfactuals. *AIJ*, 57(2-3):227–270.

Even-Dar, E., Kakade, S. M., and Mansour, Y. (2005). Reinforcement learning in POMDPs. In *IJCAI '05*.

Fagin, R., Ullman, J. D., and Vardi, M. Y. (1983). On the semantics of updates in databases. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 352–365, Atlanta, Georgia.

Ferraris, P. and Giunchiglia, E. (2000). Planning as satisfiability in nondeterministic domains. In *Proc. AAAI '00*, pages 748–753.

Fikes, R., Hart, P., and Nilsson, N. (1972). Learning and executing generalized robot plans. *AIJ*.

Fikes, R., Hart, P., and Nilsson, N. (1981). Learning and executing generalized robot plans. In Webber, B. and Nilsson, N., editors, *Readings in Artificial Intelligence*, pages 231–249. MK.

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning probabilistic relational models. In *IJCAI '99*, pages 1300–1307. MK.

Friedman, N., Murphy, K., and Russell, S. (1998). Learning the structure of dynamic probabilistic networks. In *Proc. UAI '98*. MK.

Frisch, A., Sheridan, D., , and Walsh, T. (2002). A fixpoint encoding for bounded model checking. In *FMCAD'02*.

Getoor, L. (2000). Learning probabilistic relational models. *Lecture Notes in Computer Science*, 1864:1300–1307.

Ghahramani, Z. and Jordan, M. I. (1997). Factorial Hidden Markov Models. *Machine Learning*, 29:245–275.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL – The Planning Domain Definition Language, version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale center for computational vision and control.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proc. ICML-94*, pages 10–13.

Giunchiglia, E. and Lifschitz, V. (1998). An action language based on causal explanation: preliminary report. In *Proc. AAAI '98*, pages 623–630.

Hoffmann, J. and Brafman, R. (2004). Conformant planning via heuristic forward search: A new approach.

Jaakkola, T., Singh, S. P., and Jordan, M. I. (1994). Reinforcement learning algorithm for partially observable Markov decision problems. In *Proc. NIPS'94*, volume 7.

Järvisalo, M., Junttila, T., and Niemelä, I. (2004). Unrestricted vs restricted cut in a tableau method for Boolean circuits. AI&M 15.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *AIJ*, 101:99–134.

Kearns, M., Mansour, Y., and Ng, A. Y. (2000). Approximate planning in large pomdps via reusable trajectories. In *Proc. NIPS'99*, pages 1001–1007.

Kumar, T. S. and Russell, S. (2006). On some tractable cases of logical filtering. In *Proc. ICAPS'06*.

Latvala, T., Biere, A., Heljanko, K., and Junttila, T. (2004). Simple bounded LTL model checking. Research report, Helsinki University of Technology.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press, Cambridge, U.K. Also available at http://planning.cs.uiuc.edu/.

Liberatore, P. (1997a). The complexity of belief update. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 68–73.

Liberatore, P. (1997b). The complexity of the language A. *ETAI*.

Lifschitz, V. (2000). Missionaries and cannibals in the causal calculator. In *Proc. KR '2000*, pages 85–96. MK.

Lin, F. and Reiter, R. (1995). How to progress a database II: The STRIPS connection. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence (IJCAI '95)*, pages 2001–2007, Montreal, Canada.

Lin, F. and Reiter, R. (1997). How to progress a database. *AIJ*.

Lind-Nielsen, J. (1999). Buddy - a binary decision diagram package. Technical report, Institute of Information Technology, Technical University of Denmark.

Littman, M. L. (1996). *Algorithms for sequential decision making*. PhD thesis, Brown U.

McCarthy, J. (1986). Applications of circumscription in formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116.

McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence http://www-formal.stanford.edu/jmc/mcchay69.html. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press.

Miller, R. and Shanahan, M. (1999). The event calculus in classical logic – alternative axiomatizations. *ETAI*, 4:nr 16. under review.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proc. ICML-88*.

Nebel, B. and Bäckström, C. (1992). On the computational complexity of temporal projection and plan validation. In *Proc. AAAI '92*, pages 748–753, Cambridge, MA, USA. MIT Press.

Ostrowski, R., Gregoire, E., Mazure, B., and Sais, L. (2002). Recovering and exploiting structural knowledge from cnf formulas. In *CP'02*.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2004). Learning probabilistic relational planning rules. In *Proc. ICAPS'04*.

Petrick, R. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *ICAPS-04*. AAAI Press.

Qiang Yang, K. W. and Jiang, Y. (2005). Learning action models from plan examples with incomplete knowledge. In *Proc. ICAPS'05*.

Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press.

Reiter, R. (2001). *Knowledge In Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press.

Sandewall, E. (1994). *Features and Fluents*. Oxford University Press.

Selman, B., Levesque, H. J., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California. American Association for Artificial Intelligence, AAAI Press.

Shahaf, D. and Amir, E. (2006). Learning partially observable action schemas. In *Proc. AAAI '06*. AAAI Press.

Shahaf, D. and Amir, E. (2007). Logical circuit filtering. In *IJCAI '07*. MK.

Shahaf, D., Chang, A., and Amir, E. (2006). Learning partially observable action models: Efficient algorithms. In *Proc. AAAI '06*. AAAI Press.

Son, T. C. and Baral, C. (2001). Formalizing sensing actions a transition function based approach. *AIJ*, 125(1–2):19–91.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: an introduction*. MIT Press.

Thielscher, M. (1998). Introduction to the fluent calculus. *ETAI*, 3:nr 14.

Thiffault, C., Bacchus, F., and Walsh, T. (2004a). Solving non-clausal formulas with DPLL search. In *CP'04*.

Thiffault, C., Bacchus, F., and Walsh, T. (2004b). Solving non-clausal formulas with dpll search. In *Principles and Practice of Constraint Programming*.

Val, A. D. (1992). Computing knowledge base updates. In *Proc. KR '92*, pages 740–750. MK.

Val, A. D. and Shoham, Y. (1994). A unified view of belief revision and update. *Journal of Logic and Computation*, 4(5):797–810.

Wang, X. (1995). Learning by observation and practice: an incremental approach for planning operator acquisition. In *Proc. ICML-95*, pages 549–557. MK.

Williams, B. C. and Nayak, P. P. (1996). A model-based approach to reactive self-configuring systems. In *Proc. AAAI '96*, pages 971–978.

Winslett, M. (1990). *Updating Logical Databases*. Cambridge U. Press.

Wu, K., Yang, Q., and Jiang, Y. (2005). Arms: Action-relation modelling system for learning action models. Proc. ICAPS'05.

Vita

Dafna Shahaf

Research Interests

Artificial Intelligence (AI), Learning theory, Game Theory and Algorithms.

Education

M.S. in Computer Science, University of Illinois at Urbana-Champaign, IL

2005–Current: anticipated graduation in May 2007

Research supervised by Prof. Eyal Amir

GPA: 4.0 / 4.0

B.Sc. Mathematics and Computer Science (Double Major), Tel Aviv University, Israel

2001–2005, Graduated with honors *(Summa Cum Laude)*

GPA: 96% (4.0 / 4.0)

Took several M.Sc. classes before coming to UIUC, GPA: 4.0 / 4.0

Honors & Awards

2006: Siebel Fellowship – given to five graduate students from ten leading universities (siebelscholars.com)

2006: Phi Kappa Phi Honor Society (phikappaphi.org)

2004: Tel-Aviv University Dean's List award of Distinction.

2004: Tel-Aviv University Computer Science school award of Distinction.

2004: The Shmuel Beck Scholarship – given to one student in the faculty of Exact Sciences

2004: Laureate of Wolf Foundation prize of Distinct Student.

2003: Tel-Aviv University Dean's List award of Distinction.

2002: Center of Technology, 8200 Unit – Distinct Soldier.

2002: Tel-Aviv University Dean's List award of Distinction.

2001: Tel-Aviv University Freshmen Fellowship – given to one freshman from several selected programs

2001: President of Israel - IDF Distinct Soldier Award.

Publications

Refereed Conference Papers

[1] D. Shahaf and E. Amir, *Logical Circuit Filtering*, in 20th International Joint Conference on Artificial Intelligence (IJCAI'07), 2007

[2] D. Shahaf and E. Amir, *Learning Partially Observable Action Schemas*, in 21st National Conference on Artificial Intelligence (AAAI'06), 2006

[3] D. Shahaf, A. Chang, and E. Amir, *Learning Partially Observable Action Models: Efficient Algorithms*, in 21st National Conference on Artificial Intelligence (AAAI'06), 2006

[4] D. Shahaf and E. Amir, *Towards a Theory of AI Completeness*, in CommonSense'07, 2007

Academic Service

Reviewer, 21st National Conference on Artificial Intelligence (AAAI'06), 22st National Conference on Artificial Intelligence (AAAI'07), CommonSense'07, Artificial Intelligence Journal.

Volunteer, 21st National Conference on Artificial Intelligence (AAAI'06).

Selected Coursework

Computer Science Theory Computational Models, Efficiency of Computations, Computational Complexity, Distributed Computing, On-line and Approximation algorithms.

Pure Mathematics Logic, Probability Theory, Graph Theory, Number Theory, Game Theory (Cooperative Games), Group Theory, Bioinformatics and Learning Seminar (Advanced Statistical Methods).

Applied Mathematics and Computer Science Reasoning and Knowledge Representation, Decision Making Under Uncertainty, Planning Algorithms, Modern Cryptography, Introduction to Artificial Intelligence, Neural Computation, Reinforcement Learning Workshop, Software Project (genetic algorithms).

Work Experience

Research Assistant (2005–Current), Department of Computer Science, University of Illinois at Urbana-Champaign (Under the supervision of Prof. Eyal Amir). We develop algorithms that keep track of the world and learn its dynamics, using insights from Mathematical Logic. We presented the first algorithm that is tractable and exact for *all* deterministic domains; we are currently applying our techniques to Planning and Verification problems. We also investigate ways to establish theoretical foundations for AI.

Researcher at Safend Ltd. (2004–2005) 32 Habarzel Street, Tel Aviv, Israel.

Analyzed software and protocols, found endpoint security holes, implemented proof-of-concept attacks and tested security solutions.

Grader, Tel-Aviv University and University of Haifa (2003–2004)

For classes "Introduction to Modern Cryptography" and "Introduction to AI", respectively.

Israel Defense Force (IDF), Intelligence Corps, 8200 Unit (2000–2003)

Served as an individual contributor in an award-winning communication software project.

Skills

Programming Languages: C++, Visual Basic, Lisp, Scheme, Ocaml, Perl, Prolog, Matlab, SQL, Pascal.

Web Skills: HTML, ASP, VBScript.

Languages: Hebrew (native), English (high fluency), French as a foreign language.

Interests: Scuba Diving, Swimming (a certified instructor), Science Fiction and Animation.