NOTE TO USERS

Calculating the probability of an LTL formula over a labeled Markov chain

Jacob Eliosoff
School of Computer Science
McGill University, Montreal

A thesis submitted to McGill University
in partial fulfilment of the requirements of
the degree of Master of Science
July 2003

Your file Votre référence
ISBN: 0-612-98626-8
Our file Notre référence
ISBN: 0-612-98626-8

Abstract

This thesis presents a new algorithm to compute the probability that a state in a labeled Markov chain model satisfies an LTL specification. A solution to this problem was given by Courcoubetis & Yannakakis in 1995, but unlike their solution the algorithm presented here requires no transformations of the input model. This advantage may be significant because of the large models which occur in most practical verification problems. The new algorithm is proved correct and shown to be not worse than doubly exponential in the size of the formula and cubic in the size of the model. However limited experimental results suggest that these bounds are pessimistic and that with further optimization the algorithm might approach the efficiency of current model checkers, which compute only true or false rather than an exact probability. I also include a working Java implementation, MCMC, and a proof that for any *plausible* path P followed by a Markov chain and any LTL formula ϕ, some finite prefix of P determines whether P satisfies ϕ.

Résumé

Cette thèse présente un nouvel algorithme pour calculer la probabilité qu'un état dans une chaîne de Markov étiquetté (modèle) satisfait une formule LTL (propriété). Un tel algorithme a été donné par Courcoubetis et Yannakakis en 1995, mais à la différence de leur solution, l'algorithme présenté ici n'exige aucune transformation du modèle d'entrée. Cet avantage peut être significatif en raison des grands modèles qui se produisent dans la plupart des problèmes pratiques de vérification. Le nouvel algorithme est prouvé valide et il a été démontré qu'il n'est pas plus mauvais que doublement exponentiel dans la taille de la formule et cubique dans la taille du modèle. Cependant, des résultats expérimentaux partiels suggèrent que ces limites sont pessimistes et qu'avec plus d'optimisation l'algorithme pourrait approcher l'efficacité des model-checkers courants, qui calculent seulement vrai ou faux plutôt qu'une probabilité exacte. J'inclus également une implémentation fonctionnant en Java, MCMC, et une preuve que pour n'importe quel chemin *plausible* P suivi par une chaîne de Markov et n'importe quelle formule LTL ϕ, un préfixe fini de P détermine si P satisfait ϕ.

Preface

The basic structure of this thesis is: introductory example, background, formal statement of problem, description of solution, proofs, implementation, conclusions, appendices.

I've tried to write it in such a way that readers with no relevant background can at least understand the problem and some of the ideas behind my solution. (Hi Mom!) For these readers I suggest Chapters 1 and 3, and the advice that these things are often better read twice fast than once carefully.

Acknowledgements

I am not good at finishing things, and this took a lot of people's help. I'm very grateful!

In particular:

First and foremost, Prakash Panangaden, my supervisor, debating opponent and friend. Prakash is a model of what I think a teacher should be. Without him, I would never have entered grad school, attempted this thesis, or finished it.

My labmates, Ernesto and Will, for many memorable conversations and accidents, and the best bet I ever lost.

Norm "Horse" Ferns, without whose painful but necessary correction this work might have ended much sooner.

My various minders, including Prakash and my parents, for being curious rather than impatient.

I would also like to especially thank the friends and family who put up with me in the antisocial later stages.

This work is dedicated to my grandmother, who thought she'd never see it, and to Raja Vallée-Rai.

Contents

Chapter 1

Introduction: A Simple Example

This thesis describes a new algorithm for the verification of probabilistic systems. The problem considered is as follows: given a state in a labeled Markov chain, and an LTL formula, compute the probability that the state satisfies the formula.

In this chapter I first give some context, then go through an example of what we want to do.

1.1 Interest of the algorithm

A solution to this problem was presented by Courcoubetis & Yannakakis in [CY95], but unlike the solution presented here, their algorithm requires repeated nontrivial transformations of the input Markov chain. Since model size is the limiting factor in most practical verification problems, a more popular technique in practice is to encode the probability in the specification (e.g., "formula ϕ is satisfied with probability ≥ 0.9"). The specification can then be checked efficiently using non-probabilistic model checking techniques, but the result is only true/false rather than an exact probability ([ASBBS95], [BCHKR97]). This work seeks to combine these advantages, using a BDD-based algorithm resembling model checking to compute exact probabilities without transforming the model. This algorithm cannot beat the theoretical complexity bounds proved and matched in [CY95], but it is hoped that it may be found more efficient in practice.

A detailed discussion of related work appears in Chapter 7.

1.2 First input: a state in a labeled Markov chain

Figure 1.1 shows a labeled Markov chain, $\mathcal{M}_1$.

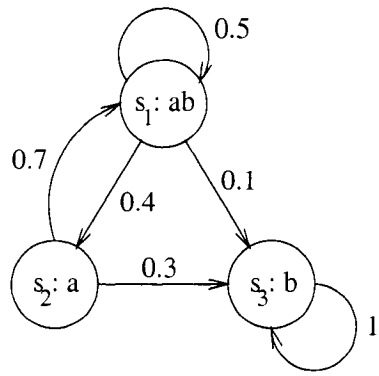

Figure 1.1: First input: a labeled Markov chain. $\mathcal{M}_1$.

$\mathcal{M}_1$ has three *states* (s_1, s_2 and s_3), six *edges* (arrows), and two *atoms* (a and b). a is true in s_1 and s_2, false in s_3. b is true in s_1 and s_3 but false in s_2.

We say that $\mathcal{M}_1$ is always "in" one of its three states. At each step, it may move to a different state, or stay where it is. The weight of an edge from one state s to another s' shows the probability of going from s to s' at the next step, assuming $\mathcal{M}_1$ is in s right now. The outgoing edge weights from any state always add up to 1. So, if $\mathcal{M}_1$ is in s_1, after the next step it has a 40% chance of being in s_2, a 10% chance of being in s_3, and a 50% chance of staying in s_1.

1.3 Second input: an LTL formula

LTL is a temporal logic: an LTL formula makes an assertion about the future (and/or present). Figure 1.2 shows some LTL formulas, with English translations.

These formulas contain all LTL's basic constructs: $\mathcal{T}/\mathcal{F}$ (true/false), atoms, $\neg$ (not), $\wedge$ (and), X (next), and $\mathcal{U}$ (until). Short formulas are easy to understand, but more elaborate combinations become hard to express in English.

a	a is true
b	b is true
$\neg a$	a is false
$a \wedge b$	a and b are both true
Xa	a will be true after one step
$X\mathcal{F}$	False will be true after one step (a contradiction)
$X(a \wedge b)$	a and b will both be true after one step
$(Xa) \wedge b$	b is true, and a will be true after one step
XXa	a will be true after two steps
$\mathcal{T}\mathcal{U}\neg b$	True until b is false (b will eventually be false)
$\neg(\mathcal{T}\mathcal{U}\neg a)$	Not (true until a is false) (i.e., a will always be true)
$X(b\mathcal{U}a)$	Starting from the next step, b will continually be true until a is true
$b\mathcal{U}X\neg a$	b will continually be true until it is true that a will be false after one step
$\mathcal{T}\mathcal{U}\neg(\mathcal{T}\mathcal{U}a)$	Eventually a will never again be true (i.e., a occurs finitely often)

Figure 1.2: Second input: some LTL formulas.

1.4 Output: probability of the formula

Now, given these inputs, we want to calculate the probability that the state satisfies the formula. What does this mean? If you start $\mathcal{M}_1$ in state s_1 and let it run from there, it will follow some sequence of states like $(s_1, s_2, s_1, s_3, s_3, \ldots)$, called a *path*. For any state s and formula ϕ, we want the probability that a path starting from s satisfies ϕ. See the example output in Figure 1.3.

Some of these outputs are easy to calculate. The simple LTL formula a is certain to be true in s_1 and s_2 and false in s_3. To compute the probability that s_1 satisfies the formula Xa, we observe that starting from s_1, $\mathcal{M}_1$ has a 0.9 probability of going to either s_1 or s_2 (where a is true) and a 0.1 probability of moving to s_3 (where a is false).

However, to compute the probability of an input like formula $(\mathcal{T}\mathcal{U}\neg b)$ ("b is eventually false") in s_1 is not so easy. Informally, we can get the answer by reasoning as follows. If $\mathcal{M}_1$ starts in s_1, it may stay in s_1 for a while, but if we wait long enough it's bound to go to either s_2 or s_3. If it goes to s_2, the $\neg b$ clause of the until is satisfied, and the formula is true. If it goes to s_3, it will loop around there forever, and $\neg b$ will

	s_1	s_2	s_3
a	1	1	0
b	1	0	1
$\neg a$	0	0	1
$a \wedge b$	1	0	0
Xa	0.9	0.7	0
$X\mathcal{F}$	0	0	0
$X(a \wedge b)$	0.5	0.7	0
$(Xa) \wedge b$	0.9	0	0
XXa	0.73	0.63	0
$\mathcal{T}\mathcal{U}\neg b$	0.8	1	0
$\neg(\mathcal{T}\mathcal{U}\neg a)$	0	0	0
$X(b\mathcal{U}a)$	0.9	0.7	0
$b\mathcal{U}X\neg a$	0.44	0.3	1
$\mathcal{T}\mathcal{U}\neg(\mathcal{T}\mathcal{U}a)$	1	1	1

Figure 1.3: Output: the probability of each input formula being satisfied, from each state of $\mathcal{M}_1$.

never be satisfied. So the probability that a path from s_1 satisfies $(\mathcal{T}\mathcal{U}\neg b)$ depends entirely on which of s_2 and s_3 is visited first. The edge weights tell us that visiting s_2 first is four times likelier, which gives us our answer probability of 0.8.

Ad hoc reasoning worked this time, but it won't in general. No one could be expected to perform such reasoning on a longer input formula full of nexts and untils. To guarantee a solution for all possible input Markov chains and formulas, we need a mechanical procedure which always computes the right answer (eventually) — an *algorithm*.

This thesis describes such an algorithm.

Chapter 2

Background

This chapter contains background directly relevant to our algorithm. I first discuss what model checking algorithms are used for. Then I give more detailed explanations of the two inputs to the algorithm, labeled Markov chains and LTL formulas. Finally, we look at why LTL is more useful for our purposes than a branching-time logic like CTL.

More basic background material is in Appendix A, covering the original CTL model checking algorithm of Emerson & Clarke, and BDDs (binary decision diagrams).

2.1 What is model checking?

Model checking is a popular technique for the formal verification of concurrent systems.

A *concurrent system* is a system in which the order of events is unpredictable. Systems commonly checked by model checkers include circuit designs and communications protocols. In circuits it is the order of signals being transmitted through gates which is unpredictable; in communications protocols, it is the order in which messages are sent and received.

The unpredictability inherent in concurrent systems makes them difficult to design correctly. Even if a design is correct, its correctness is often hard to establish with any confidence. A traditional approach to this problem is *testing*: simulate the designed system's behavior on a bunch of test cases and, if no flaw is revealed, conclude that the design is probably correct. This is a reasonable strategy for systems of a manageable

size, but many modern systems are large and complex enough that the amount of testing required to establish confidence of correctness is prohibitive.

An alternative approach to testing is *formal verification*, which aims to rigorously prove the correctness or incorrectness of a concurrent system's design. There are various techniques which fall under this general category. Because formal verification is of most interest for the large systems which cannot feasibly be tested, the most popular techniques are those which have been found to work on these large systems. Model checking is such a technique.

There are three components to the model checking strategy. First, one needs a formal *model of the system*, typically a finite-state transition system of some sort like the finite automata described below. The model must be designed so that its states and transitions accurately correspond to the concurrent behavior of the system being model checked.

Second, one must specify a *specification to check*, representing the property one wants to see if the system satisfies. For example, a protocol for managing print jobs sent to a printer might be checked for *safety*: the property that no two jobs are ever sent through the printer at the same time. Specifications are typically written in a temporal logic like CTL or LTL, which can express properties like safety.

Finally, once one has a formal model and specification, one needs a *model-checking algorithm* to actually check if the model satisfies the specification. The original model checking algorithm, described below, operated on a model represented as a finite automaton, and a specification expressed in CTL. Other model checking algorithms expect the system or specification to be expressed in different formalisms.

This thesis presents an algorithm which takes a model represented as a Markov chain and a specification expressed in LTL, and returns not true or false, but the probability that the model satisifies the specification.

2.2 Markov chains

Markov chains, introduced in section 1.2, are a probabilistic extension of finite automata. (For more on finite automata, consult a standard reference such as [Sip97].)) Whereas a finite automaton only says which states a given state can and cannot move to at the next step, a Markov chain specifies the precise probability of moving to each other state.

See Figure 1.1 from page 10 for a sample Markov chain, $\mathcal{M}_1$.

2.2.1 History-independence

Like finite automata, Markov chains have the property that their behavior is *history-independent*: the future behavior of the system depends only on the node it is currently in, not on the route it took to get there. For example, if $\mathcal{M}_1$ is in state s_1, it has a 40% probability of moving to state s_2 at the next step, irrespective of how it got to s_1.

This memoryless property characterizes Markov chains, so that systems with this trait are called *Markovian*.

2.2.2 Internal vs external vs probabilistic choice

There is now an interesting decision we must make in specifying how our Markov chains will behave: who will choose the input symbols?

To make sense of this decision requires first a discussion of choice. A transition system may be thought of as a machine whose behavior arises as a result of ongoing choices. These choices can be divided into three categories based on the agent making the choice: internal, external, and probabilistic.

Some choices are *internal*: made within the system. For example, a deterministic finite automaton (DFA) which reads symbol a in state s_1 is forced to some other state s_i. The decision about which state to move to is determined by the design of the system, and requires no outside consultation. Insofar as a system's choices are internally made (i.e., determined by its definition), it is called a *generative* or *closed* system.

Other choices are left to an *external* agent, which may be a user or the environment. (To the system there is little distinction.) A DFA is often thought of as a machine which knows exactly what to do on a given input symbol, but has no means of itself choosing the symbol; this is left to the user, who feeds it a stream of input symbols. A system which leaves choices to the external user/environment is called *reactive*.

Finally, some choices are *probabilistic*. Who makes probabilistic choices is a subtle issue deserving of much thought. For our purposes it suffices to think of probabilistic

choices as being made neither by the system itself nor by the user, but by some probabilistic oracle which obeys the statistical properties of the distribution in question. For example, in saying that when $\mathcal{M}_1$ is in state s_1 it has a 40% probability of moving to state s_2 at the next step, we mean something like the following: "If we ran $\mathcal{M}_1$ for long enough, and kept a tally of where it moved to each time it was in s_1, we would find that the proportion eventually approached 40%." (In fact we mean more than this. For example, suppose in keeping our tally we found that $\mathcal{M}_1$ repeatedly moved to s_1 five times, then to s_2 four times, then to s_3 once. This behavior would meet one but certainly not all of our expectations for a properly functioning probabilistic oracle. But never mind this for now.)

2.2.3 Generative vs reactive Markov chains

Having set out these three categories of choice, we must now decide which will be responsible for choosing the input symbols of our Markov chains.

We could leave the choice of symbols to the user, as in finite automata. Thus, a transition in the Markov chain occurs as follows: the user chooses an input symbol, and based on this symbol and the current state, a probabilistic choice is made as to which state to move to. It follows that in such a model the outgoing probabilities from each state must sum to 1 *for each input symbol*. See the example in Figure 2.1. We will call this the *reactive model* of Markov chains.

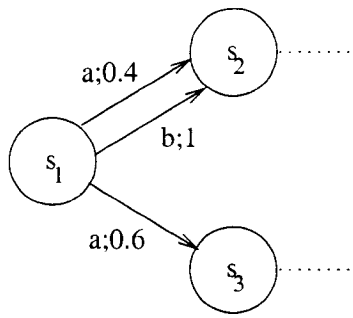

Figure 2.1: The reactive Markov chain model: the user picks a symbol, and the system probabilistically chooses the next state. Note that outgoing probabilities sum to 1 over each symbol.

Alternatively, we could have the symbols chosen probabilistically within the sys-

tem, like the state transitions. In this scenario, a transition happens like this: the system itself consults the probabilistic oracle as to which symbol to read and which state to move to. Thus the outgoing probabilities from each state sum to 1 *over all input symbols*, as in Figure 2.2. This *generative model* is a fully closed system: once started, its behavior is determined without external input.

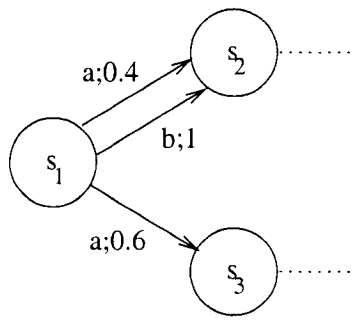

Figure 2.2: The generative Markov chain model: the system probabilistically chooses both the symbol read and the next state. Outgoing probabilities sum to 1 over all symbols.

Each of these models is interesting. The generative model, lacking external input, is simpler and therefore a sensible place to start in our exploration of probabilistic model checking. As it turns out, in the probabilistic setting, even a closed system like this is nontrivial to model check.

2.2.4 Symbols as atoms

In a finite automaton, the input consists of a sequence of input symbols. Our Markov chains will use a more general notion of symbols, in which a symbol is thought of as a boolean variable rather than an input token. To indicate this shift in interpretation, we will call our symbols *atoms*. So rather than "What symbol is read next?", we will ask, "Which atoms are true at the next step?" In the case where only one atom is true at once, this reduces to the interpretation of symbols as inputs. More generally, any number of atoms may be true.

So, our Markov chains are strictly generative, with no external input, and allow any assortment of atoms to be true at each step. The example Markov chain $\mathcal{M}_1$ above is of this sort.

2.2.5 Matrix representation

Figure 2.3 shows how we can represent $\mathcal{M}_1$ textually (without a diagram) as a matrix.

	s_1	s_2	s_3
s_1	0.5	0.4	0.1
s_2	0.7	0	0.3
s_3	0	0	1
a	1	1	0
b	1	0	1

Figure 2.3: Matrix representation of $\mathcal{M}_1$, with the edge weights and the atom probabilities.

The matrix captures the information formally identifying the Markov chain. First there is the set of states, $S = s_1 \ldots s_n$. Then we need a transition matrix $\delta : S \times S \rightarrow [0, 1]$, where $\delta(i, j)$ is the probability of moving from s_i to s_j. The outgoing edges from any state must sum to 1: $\forall i, \sum_{j=1}^{n} \delta(i, j) = 1$.

We will work with a state-labeled Markov chain model, specifying a set of symbols (aka atoms) $\Sigma = a_1 \ldots a_m$, and defining $\gamma : S \times \Sigma \rightarrow \{\mathcal{T}, \mathcal{F}\}$ where $\gamma(s_i, a) = \mathcal{T}$ (i.e., 1) iff atom a is true in state s_i. We could further generalize to allow atom probabilities between 0 and 1 ($\gamma : S \times \Sigma \rightarrow [0, 1]\}$), and this requires only a small modification of the main `measure()` algorithm described by this thesis. However the generalization is of limited use, so for clarity we will restrict ourselves to the $\mathcal{T}/\mathcal{F}$ case.

Finally, we may want to specify the probability of being in each state initially, as a vector of initial probabilities $\alpha = \alpha_1 \ldots \alpha_n$ summing to 1: $\sum_{i=1}^{n} \alpha_i = 1$. In practice we will normally be taking the initial state as an input to our model checker, and so will not bother to specify the initial distribution. This is not important.

2.2.6 Probabilistic language

Our discussion of Markov chains has made no mention of acceptance. In finite automata, a string was either accepted or rejected. In a probabilistic model like Markov chains, a boolean outcome like this will not do. Instead, a Markov chain $\mathcal{M}$ implicitly associates a probability with each input string t: the probability that $\mathcal{M}$ follows a

path along which the atoms which make up t are true. We write this probability as $\pi_{\mathcal{M}}(t)$.

For example, suppose $\mathcal{M}_1$ above starts in s_1 (with probability 1). What probability is associated in $\mathcal{M}_1$ with the very short string a? In other words, what is $\pi_{\mathcal{M}_1}(a)$? To answer, we note that the first state visited by $\mathcal{M}_1$ is guaranteed to be s_1, and a is true in s_1 with probability 1. So the answer is 1. Similarly, $\pi_{\mathcal{M}_1}(b) = 1$.

Now, what is $\pi_{\mathcal{M}_1}(ab)$? The probability that the first state visited satisfies a is 1, so we need only calculate the probability that the second state satisfies b. By $\mathcal{M}_1$'s matrix, this second state is 50% likely to be s_1 (in which b is true), 40% likely to be s_2 (false), and 10% likely to be s_3 (true). Thus the total probability of moving to a state where b is true is 0.6, and $\pi_{\mathcal{M}_1}(ab) = 0.6$. You can similarly verify that $\pi_{\mathcal{M}_1}(aa) = 0.9$ and $\pi_{\mathcal{M}_1}(aba) = 0.45$.

We call $\pi_{\mathcal{M}_1}$ the *probabilistic language* of M_1.

2.3 LTL

We now look at some of the logics used to write model checking specifications, beginning with LTL.

A *temporal logic* is a language for expressing temporal propositions: assertions about what is true and what will be true. Such assertions are made in English by sentences such as "a is true", "b will be false in one time unit", "a and b will always be true (from now onwards)", and so on. However phrasing more complicated assertions in English quickly becomes awkward. Temporal logics have been developed to let us formulate temporal assertions precisely and compactly.

2.3.1 Continuous vs discrete time

A temporal logic may view time as continuous or discrete. *Continuous*-time logics let us make assertions about what is true at any point in the future: 1 time unit (step) from now, $\frac{1}{2}$ step, π steps, and so on. *Discrete*-time logics restrict us to integer steps: now, one step from now, two steps, etc. Continuous-time logics are more expressive, but much harder to work with mathematically, and we will not use them.

2.3.2 LTL syntax & semantics

LTL (Linear-time Temporal Logic)[1], described in section 1.3, is a discrete-time logic for making assertions about a sequence of points in time $(t_0, t_1, \ldots)$, called a *path*. (The path's elements needn't necessarily represent points in time, but they usually do.) Refer back to Figure 1.2 on page 11 for some examples of LTL formulas.

$$\begin{aligned} \phi \stackrel{def}{=} \quad & \mathcal{T} \mid \mathcal{F} \\ \mid \quad & a \\ \mid \quad & \neg\psi \\ \mid \quad & \psi \wedge \omega \\ \mid \quad & X\psi \\ \mid \quad & \psi \mathcal{U} \omega \end{aligned}$$

Figure 2.4: LTL syntax: the six types of LTL formulas.

LTL syntax, summarized in Figure 2.4, is simple once one understands the different constructs. There are six types of formulas: booleans, atomic propositions, nots, ands, nexts, and untils. A given path $P = (t_0, t_1, \ldots)$ is said to either *satisfy* a given formula ϕ ($P \models \phi$) or *fail to satisfy* ϕ ($P \not\models \phi$).

The *boolean* formulas $\mathcal{T}$ (true) and $\mathcal{F}$ (false) have the same interpretations at any time. An *atomic proposition*, or *atom*, is a variable which is either true or false at each point in time. For instance, a might be true at t_0, t_1 and t_2, but then false at t_3, then true again and so on (see Figure 2.5). A path satisfies an atom iff the atom is true in its first element; so $P \models a$.

Time:	0	1	2	3	4	...
a	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$	...
b	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	...

Figure 2.5: The truth value of an atom may vary over time.

If some formula ψ is satisfied by a path P, then $\neg\psi$ ("*not* ψ") is not satisfied by P, and vice versa. Similarly, $\psi \wedge \omega$ ("ψ *and* ω") is satisfied iff both ψ and ω are satisfied.

[1] Sometimes also called PTL (Propositional Temporal Logic), or LPTL (Linear-time Propositional Temporal Logic).

These are the traditional operators from propositional logic. However, interpreting them can become subtle when ψ or ω contains temporal operators, as we will see.

$X\psi$, "*next* ψ", holds now iff ψ holds at the next step; that is, $P = (p_1, p_2, \ldots) \models X\psi$ iff $P_2 = (p_2, p_3, \ldots) \models \psi$. For example, using the atom a described above, P satisfies a, Xa, XXa, $XXX\neg a$, and $XXXXa$. (We will normally write $X\neg a$ rather than $\neg Xa$.) Of course, for any P, $P \models X\mathcal{T}$ and $P \not\models X\mathcal{F}$.

$\psi\mathcal{U}\omega$, "ψ *until* ω", is the most powerful and subtle LTL operator. However its meaning is essentially given by its English pronunciation: $\psi\mathcal{U}\omega$ is true iff (1) ω is eventually true, and (2) ψ is true in every step up until (not including) the first step where ω is true. That is, $P = (p_1, p_2, \ldots) \models \psi\mathcal{U}\omega$ iff $\exists j$ such that (1) $P_j = (p_j, p_{j+1}, \ldots) \models \omega$, and (2) $\forall 1 \leq i < j$, $P_i \models \psi$. (It follows that $(\psi\mathcal{U}\omega) \equiv (\omega \vee (\psi \wedge X(\psi\mathcal{U}\omega)))$. If ω is true immediately, $\psi\mathcal{U}\omega$ is true irrespective of ψ. See Figure 2.6 for some examples.

Note that for $\psi\mathcal{U}\omega$ to be satisfied, ω really must eventually be satisfied: $(\mathcal{T}\mathcal{U}\mathcal{F}) = \mathcal{F}$, not $\mathcal{T}$.

2.3.3 Operator binding precedence

Generally the binding precedence of LTL operators (from tightest to loosest) goes: $\neg$ and X, $\wedge$, $\mathcal{U}$. In other words, $Xa \wedge b\mathcal{U}\neg Xc \wedge s$ should be parsed as: $((Xa) \wedge b)\mathcal{U}((\neg(Xc)) \wedge s)$. We will often use parentheses to clarify the binding order, and always when the left-/right-associativity of $\wedge$ or $\mathcal{U}$ is in question.

2.3.4 Other operators

There are many other useful operators we might want to use in writing temporal formulas: for example, $\psi \vee \omega$ ("ψ or ω"), $\psi \rightarrow \omega$ ("ψ implies ω"), $F\psi$ ("eventually ψ"), $G\psi$ ("always ψ"), and others. We don't bother to include these in LTL syntax because they can all be encoded using the operators described above. In particular: $(\psi \vee \omega) = \neg(\neg\psi \wedge \neg\omega)$, $(\psi \rightarrow \omega) = \neg(\psi \wedge \neg\omega)$, $(F\psi) = (\mathcal{T}\mathcal{U}\psi)$, and $(G\psi) = \neg(\mathcal{T}\mathcal{U}\neg\psi)$. So, we can freely use such operators without fear of accidentally writing something not expressible in LTL.

An especially important derived operator is *bounded until* from [HJ94], of the form $\phi\mathcal{U}^{\leq k}\psi$ for some $k \geq 0$. As its name suggests, a bounded until is like an until except that it is only satisfied if its right-hand operand is satisfied within k steps (inclusive).

Formula	Satisfied by path P?	Explanation
Xb	No	b is false at time 1.
$b\mathcal{U}a$	Yes	a is true immediately.
$a\mathcal{U}b$	No	b is first true at time 4, but a is only true up to time 2.
$a\mathcal{U}Xb$	Yes	a is true up to time 2, and then at time 3 Xb is true.
$\mathcal{T}\mathcal{U}X(a \wedge b)$	Yes	$a \wedge b$ is true at time 4, and $\mathcal{T}$ is true until then (since $\mathcal{T}$ is always true).
$(Xa)\mathcal{U}XXb$	Yes	Xa is true up to time 1, and then at time 2 XXb is true.
$((XXb)\mathcal{U}Xa)\mathcal{U}b$	Yes	Careful now. b is first true at time 4, so we need to check that $\psi = ((XXb)\mathcal{U}Xa)$ is true at times 0–3. ψ is true immediately at times 0, 1, and 3, because Xa is true at all these times. And ψ is also true at time 2, because XXb is true until time 3 when Xa is true. So ψ is true at times 0–3, until b is true at time 4, and therefore the entire until formula is satisfied.

Figure 2.6: Explanations of some LTL until formulas, using the atoms from Figure 2.5.

Bounded untils are often useful in practical model-checking problems where one wants to include time constraints in the spec, e.g., to verify that a property is satisfied within a fixed number of steps.

Because of its finite horizon, a bounded until can be encoded without the pure until operator:

Definition 2.1

$$\begin{aligned} \tau\mathcal{U}^{\leq 0}\upsilon &\stackrel{def}{=} \upsilon \\ \tau\mathcal{U}^{\leq (k\geq 0)}\upsilon &\stackrel{def}{=} \upsilon \vee (\tau \wedge X(\tau\mathcal{U}^{\leq k-1}\upsilon)) \end{aligned}$$

The resulting formula may be large (though not worse than linear in k), but it contains no pure untils. For example, $a\mathcal{U}^{\leq 2}b$ is equivalent to $b \vee (a \wedge X(b \vee (a \wedge Xb)))$.

So, as with the other derived operators, for the purposes of our proofs we will normally ignore bounded until and work only with the primitive operators. An important exception is complexity proofs, where we can often achieve better bounds for input formulas containing no pure untils (see chapter 4). These results reflect the fact that although bounded untils are less elegant than pure untils, and lead to more complicated formulas, they can be more efficient to model check, especially when nested.

2.4 Linear-time or branching-time?

Traditional model checking works on nondeterministic systems such as an nondeterministic finite automata (NFAs). However, what interests us here is model checking probabilistic systems. The labeled Markov chains described in section 2.2 give us a natural probabilistic model, but the choice of specification language is less obvious. In particular, we must choose between linear-time and branching-time temporal logics (compared in section A.1.1). Here we will argue that, for probabilistic model checking, linear-time logics like LTL are more appropriate.

At first, branching-time logics like CTL (section A.1.2 of Appendix A) might seem appropriate: like NFAs, Markov chains have many possible futures and therefore generate computation trees like those described in section A.1.1. The probabilistic information given by the Markov chain's transition probability matrix lets us additionally associate a probability with each branch in the tree. More precisely, we can label each node in a Markov chain's computation tree with the probability of reaching that node.[2]

Figure 2.7 shows the computation tree generated by $\mathcal{M}_1$ from Figure 1.1. The probability $R(n_i)$ of reaching a node n_i is easy to compute: it's the probability of reaching n_i's parent node n_p, $R(n_p)$, multiplied by $\delta(n_p, n_i)$, the probability of moving from n_p to n_i in the Markov chain. Note that the probabilities of reaching any node n's children sum to the probability of reaching n itself, since once n is reached, exactly

[2]The probability of reaching a node in the computation tree is not to be confused with the probability of reaching the corresponding state in the Markov chain. The same state may appear multiple times in the tree. In fact a state will generally appear infinitely often in the tree.

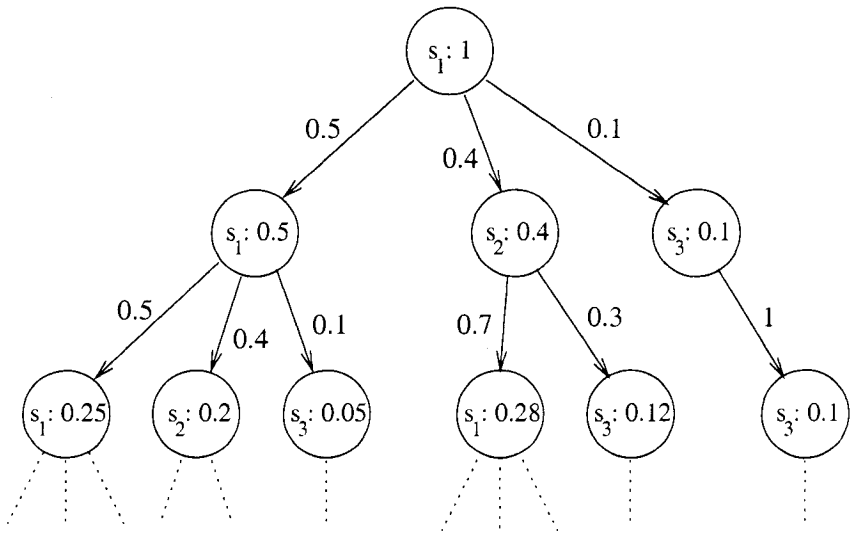

Figure 2.7: The first few levels of the computation tree generated by Markov chain $\mathcal{M}_1$ from Figure 1.1. Each node n is labeled with $R(n)$, the probability of reaching n given that one starts at the top of the tree in s_1.

one of its child nodes will be reached. Also note that the probabilities of the nodes at any level in the tree sum to 1, since if you let the Markov chain run for ℓ steps, you will reach exactly one of the nodes at level ℓ.

In traditional model checking, CTL is attractive as a specification language because of its path quantifiers E and A, which let us write specifications such as "at least one path from state s satisfies a within two steps" or "every path from s eventually satisfies b". But note that these assertions are non-probabilistic in nature: to check the truth of such an assertion in a state s of a Markov chain $\mathcal{M}$, we needn't know the exact edge probabilities in $\mathcal{M}$, only which edges have probability > 0 and which don't. In other words, any CTL specification beginning with E or A can be solved by the old non-probabilistic model checking algorithms.

Furthermore, the results obtained by model checking CTL specifications in probabilistic systems can be misleading. For example, consider the spec $A(\mathcal{TU}\neg a)$ ("along every path, a is eventually false") evaluated in state s_1 of Markov chain $\mathcal{M}_1$ from page 10. A non-probabilistic model checker will conclude that this assertion is false in s_1, since there exist infinite paths like $(s_1, s_1, s_1, \ldots)$ along which a is always true. But it is easy to see that the probability of $\mathcal{M}_1$ following such a path is 0. In fact,

$\mathcal{T}\mathcal{U}\neg a$ is *almost surely* satisfied (that is, with probability 1), because $\mathcal{M}_1$ is bound to end up in state s_3 eventually, where a is false.

In examples like this, it is generally less useful to know that *some* path in $\mathcal{M}$ contradicts the base formula ($\mathcal{T}\mathcal{U}\neg a$), than to know that *no plausible* path in $\mathcal{M}$ does. (This useful idea of a plausible path must be defined; we do so in section 5.1.)

In short, the types of all-or-nothing specifications expressed with CTL's E and A quantifiers can be checked in probabilistic systems like Markov chains without new techniques, and in any case can be misleading because they fail to identify probability 0 events as impossible. But apart from these quantifiers, CTL is just a subset of LTL. (The same holds for CTL*, the more powerful branching-time logic of which CTL is a subset.) So for the type of probabilistic model checking we want to do, LTL is a more useful specification language than a branching-time logic like CTL.

2.5 PCTL/pCTL*

Work such as [HJ94] and [ASSB96] has proposed logics (PCTL and pCTL*, respectively) which extend LTL with state formulas of the form $Pr_{>k}(\phi)$. Such a formula is true in a state s iff the probability that a path from s satisfies path formula ϕ is greater than k. For example, in Markov chain $\mathcal{M}_1$ from Figure 1.1, the pCTL* formula $Pr_{>0.8}(Xa)$ is true in s_1 but false in s_2 and s_3, since the probabilities of Xa in these states are 0.9, 0.7, and 0 (Figure 1.3).

The attraction of these logics is that fast model checking algorithms can be used to check them, making them practical for the sorts of large models arising in practical verification problems. On the other hand, because such algorithms produce only true or false, some expressiveness is sacrificed for this performance gain. The algorithm described by this thesis produces a probability rather than just true or false, and is therefore for most purposes more informative than the PCTL/pCTL* algorithms.[3] How much speed is sacrificed in exchange for this stronger output, on practical ver-

[3]Logics like PCTL can distinguish non-bisimilar systems, whereas LTL cannot. For example, the PCTL model-checking algorithm can check formulas such as $Pr_{>0}(XPr_{>0.5}(\mathcal{T}\mathcal{U}a))$: "With probability greater than 0, after one step, it will be more likely than not that a will eventually be true." However, most formulas checked in practice have the form of a pure LTL path formula inside a $Pr_{>k}$ operator. On such input formulas, the algorithm presented here is strictly more informative than the PCTL/pCTL* algorithms.

ification problems, is probably the most important open question of this work (see Chapter 8).

Chapter 3

The Algorithm

This chapter presents the solution algorithm. I begin with a formal statement of the problem and an outline of the solution, then lay out the algorithm in detail. The chapter ends by tracing through the algorithm as it solves some sample problems from Chapter 1.

3.1 Formal statement of the problem

Given:

- $\mathcal{M}$, a labeled Markov chain with states $S = s_1 \ldots s_n$, transition matrix $\delta : S \times S \rightarrow [0, 1]$, atoms $\Sigma = a_1 \ldots a_m$, and atom truth values $\gamma : S \times \Sigma \rightarrow \{\mathcal{T}, \mathcal{F}\}$
- s, a state in $\mathcal{M}$
- ϕ, an LTL formula

We want an algorithm to compute $\texttt{prob}(s \models \phi)$, the probability that an infinite path $P = (p_1 = s, p_2, p_3, \ldots)$ starting from s satisfies ϕ. More precisely, there is a measure μ_s on the set of paths starting from s, and we want to compute $\mu_s(\{P \mid P \models \phi\})$.

(In pCTL* terms, we are trying to find the k such that $\texttt{prob}_{=k}(\phi)$ is true. Note that though ϕ is a path formula, $\texttt{prob}_{=k}(\phi)$ is a state formula.)

3.2 Outline

This problem is more difficult than typical non-probabilistic model checking problems, mainly because there is no obvious way to solve it compositionally. In a non-probabilistic setting you can usually determine the truth value of $\psi \wedge \omega$ in state s by, e.g., combining the recursively computed truth values for ψ and ω in s. But in a probabilistic setting, simply knowing that the probabilities of ψ and ω are 0.6 and 0.4 respectively is not enough to derive the probability of $\psi \wedge \omega$. Certainly just multiplying the probabilities to give 0.24 won't work, because this assumes they're independent. Suppose for example that ψ is Xa and ω is $X(a \wedge b)$. Then $\texttt{prob}(s \models \psi \wedge \omega) = \texttt{prob}(s \models (Xa) \wedge X(a \wedge b)) = \texttt{prob}(s \models X(a \wedge (a \wedge b))) = \texttt{prob}(s \models X(a \wedge b)) = \texttt{prob}(s \models \omega) = 0.4$. On the other hand, if $\psi = Xa$ and $\omega = X\neg a$ then no path satisfies both — so $\texttt{prob}(s \models \psi \wedge \omega)$ is 0!

Nevertheless, the problem can be solved recursively. The algorithm, $\texttt{measure}(\phi, s)$, follows this basic outline:

1. Compute $\phi' = \texttt{step}(\phi, s)$: a formula such that $P = (p_1 = s, p_2, p_3, \ldots)$ satisfies ϕ iff its suffix $P_2 = (p_2, p_3, \ldots)$ satisfies ϕ'. (For example, $\texttt{step}(Xa, s) = a$.)

2. For each possible successor state s', recursively compute $\texttt{measure}(\phi', s')$, and sum the results weighted by edge probability: $\sum_{s'} \delta(s, s') \cdot \texttt{measure}(\phi', s')$.

However, because $\texttt{step}(\phi, s)$ is not necessarily smaller than ϕ, we need three additional tricks to ensure the recursion terminates:

- Use BDDs (see appendix) to represent the input formula and all derived formulas. This ensures that only a finite number of different formulas will be created, resulting in a bound on the number of recursive calls.

- There will be cases where the recursion loops, i.e., $\texttt{step}(\texttt{step}(\ldots \texttt{step}(\phi, s))) = \phi$. Handle these by remembering which recursive calls have already been made, assigning a variable to each call, and returning the variable rather than continuing to recurse if the same call is repeated. The result will be a system of n equations in n variables, which can then be solved.

- The system of equations will be unsolvable if any of the equations reduce to a redundancy like $x = x$. We can show that this can only happen for input

formulas containing *unrealizable* untils (Definition 3.4), of the form $\psi \mathcal{U} \omega$ where $\texttt{prob}(s \models \omega) = 0$ in all reachable states s. Any such until is equivalent to $\mathcal{F}$, so we can avoid the $x = x$ problem by making `step()` "detrivialize" ϕ by replacing unrealizable until subformulas with $\mathcal{F}$.

With these three tricks, we can show that `measure()` terminates.

3.3 Representing LTL formulas as LTL-BDDs

The way we represent LTL formulas as BDDs ("LTL-BDDs") is quite simple. Essentially we represent booleans and atoms as usual by their corresponding BDDs, and each distinct next or until formula by a unique atom. The following $\texttt{bdd}(\phi)$ procedure constructs the LTL-BDD for a given LTL formula ϕ:

Definition 3.1

$$
\begin{aligned}
\texttt{bdd}(\mathcal{F}) &\stackrel{def}{=} \text{BDD } \mathcal{F} \\
\texttt{bdd}(\mathcal{T}) &\stackrel{def}{=} \text{BDD } \mathcal{T} \\
\texttt{bdd}(a) &\stackrel{def}{=} \text{BDD } (a \ ? \ \mathcal{T} : \mathcal{F}), \text{ representing atom "}a\text{"} \\
\texttt{bdd}(\neg\psi) &\stackrel{def}{=} \texttt{not}(\texttt{bdd}(\psi)) \\
\texttt{bdd}(\psi \wedge \omega) &\stackrel{def}{=} \texttt{and}(\texttt{bdd}(\psi), \texttt{bdd}(\omega)) \\
\texttt{bdd}(\psi \vee \omega) &\stackrel{def}{=} \texttt{or}(\texttt{bdd}(\psi), \texttt{bdd}(\omega)) \\
\texttt{bdd}(X\psi) &\stackrel{def}{=} \text{BDD } (X\psi \ ? \ \mathcal{T} : \mathcal{F}), \text{ representing atom "}X\psi\text{"} \\
\texttt{bdd}(\psi \mathcal{U} \omega) &\stackrel{def}{=} \text{BDD } (\psi \mathcal{U} \omega \ ? \ \mathcal{T} : \mathcal{F}), \text{ representing atom "}\psi \mathcal{U} \omega\text{"}
\end{aligned}
$$

See the examples in Figure 3.1.

Using BDDs requires that we impose an order on all our BDD atoms: those matching LTL atoms (e.g., "a"), those representing nexts ("Xa"), and those representing untils ("$a \mathcal{U} b$"). Any consistent ordering will do, so the choice of ordering can be left up to the implementation (and, e.g., based on observed performance effects). For example, one can order LTL atoms before nexts before untils, and lexicographically within each type: e.g., $a < b < Xa < Xb < a \mathcal{U} b < b \mathcal{U} a \ldots$.

Unfortunately this is not a canonical LTL representation. For example, the LTL-BDD for $X\neg a$ will be different from the LTL-BDD for $\neg Xa$, although both LTL formulas are equivalent. However, the representation does have the following useful

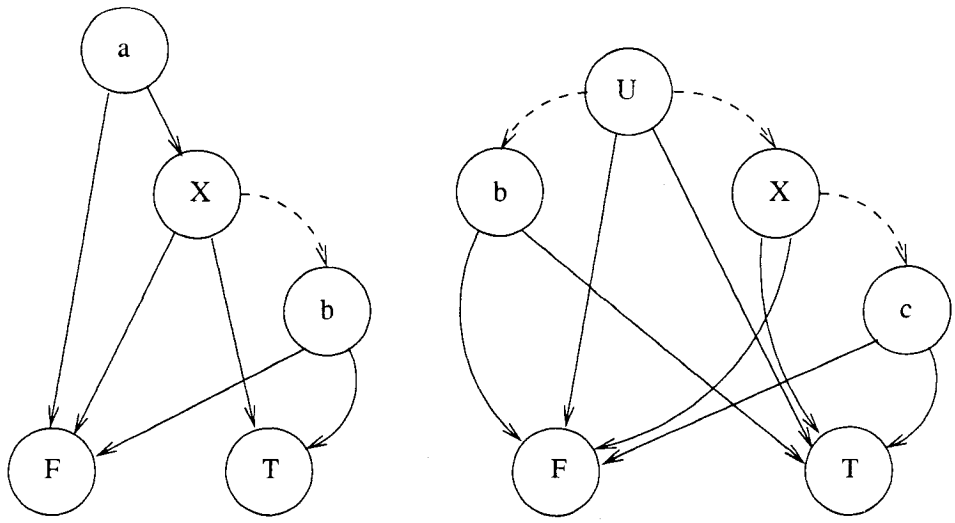

Figure 3.1: Example LTL-BDD representations of LTL formulas: $a \wedge Xb$. $b\mathcal{U}\neg Xc$. Apart from the basic boolean nodes for $\mathcal{F}$ and $\mathcal{T}$, each node contains a formula α and has two child nodes indicated by the solid edges: a left child it's equivalent to when α is false, and a right child it's equivalent to when α is true. The X-node with a dotted edge to b "contains", and represents, the formula Xb. The X-node with a dotted edge to c contains Xc, but represents $\neg Xc$ (note that its right child is $\mathcal{F}$, not $\mathcal{T}$). The $\mathcal{U}$-node contains and represents $b\mathcal{U}\neg Xc$.

property: starting from a finite set of LTL-BDDs, only a bounded number of new LTL-BDDs can be generated by applications of basic BDD operators like `not()` and `and()` to elements of the set, since none of these operations create new atoms (Proposition 5.15). This fact will help us prove that `measure()` terminates.[1]

[1]Adapting BDDs to give a truly canonical LTL representation, TBDDs ("temporal BDDs"), was originally one of the aims of this thesis. In TBDDs, $X\psi$ was represented by a special non-atom node with a link to the TBDD for ψ, and $\psi\mathcal{U}\omega$ as a self-looping structure: `or`(ω, `and`(ψ, `next`($\psi\mathcal{U}\omega$))). The TBDD for a formula ϕ was like a little Turing machine, inspecting the input path one state at a time and eventually recognizing if ϕ was satisfied or contradicted. However my colleague Norm Ferns pointed out that as I had designed them TBDDs could not express LTL's infinitary properties. For example, a formula like $\mathcal{T}\mathcal{U}\neg(\mathcal{T}\mathcal{U}a)$ ("a occurs finitely often") is not satisfied or contradicted by any finite subpath of the input, and therefore its TBDD was the same as the TBDD for $\mathcal{T}\mathcal{U}\neg(\mathcal{T}\mathcal{U}b)$ (and many others). This flaw proved fatal and, since a truly canonical representation wasn't necessary for the main algorithm, I eventually had to drop the idea. But it was cool and anyone interested in pursuing it is invited to get in touch.

In this work we will represent LTL formulas exclusively as LTL-BDDs, so we will often refer to LTL-BDDs informally as "formulas".

3.4 Procedures

The algorithm makes use of two main procedures, `measure()` and `step()`, and an auxiliary procedure, `solve()`. The following sections define and explain them.

3.4.1 `measure()`

Definition 3.2

```
measure(φ, s):
    if φ = 𝒯 or φ = ℱ:
        return 1 if 𝒯, 0 if ℱ
    else if a solution for (φ, s) has been cached:
        return fetch(φ, s)
    else:
        generate new var x_φs
        cache(φ, s, x_φs)
        φ' := step(φ, s)
        e := sum over all s' such that δ(s, s') > 0 of:
            δ(s, s')·measure(φ', s')
        r_φs := solve(x_φs = e)
        substitute(x_φs := r_φs)
        return r_φs
```

`measure()`'s fundamental property is Theorem 5.48: `measure`$(\phi, s) =$ `prob`$(s \models \phi)$. The pseudocode shown closely corresponds to the Java code in MCMC, the implementation described in Chapter 6. Several lines deserve explanation.

`measure()` begins by checking for $\mathcal{T}$ or $\mathcal{F}$, or a cached solution. (`cache`$(\phi, s, r_{\phi s})$ simply stores a solution $r_{\phi s}$ in a global cache where `fetch`(ϕ, s) can retrieve it.) If ϕ doesn't fall into one of these simple cases, a new variable $x_{\phi s}$ representing `prob`$(s \models \phi)$ is created and cached so that recursive calls to `measure`(ϕ, s) will return it (rather

than diverging). Then an expression (linear combination of numbers and variables) for the value of $x_{\phi s}$ is recursively computed and substituted for it. The LTL-BDD representation and the global cache make it easy to prove Proposition 5.18: `measure()` terminates.

The key idea of the algorithm is the recursion to compute the value of $x_{\phi s}$. The input Markov chain $\mathcal{M}$ specifies $\delta(s, s')$, the probability that the next state will be s' given that the current state is s. Now, suppose we know the next state s'. `step`(ϕ, s) will give us a formula ϕ' which is satisfied by any path $P_2 = (p_2 = s', p_3, p_4, \ldots)$ iff ϕ is satisfied by the corresponding $P = (p_1 = s, s', p_3, p_4, \ldots)$. And by induction, a recursive call to `measure`(ϕ', s') will compute `prob`($s' \models \phi'$) — which, again under our assumption that the next state is s', is equal to `prob`($s \models \phi$).

In other words, if we know the next state s', we can recursively compute `prob`($s \models \phi$). So, we can compute the real `prob`($s \models \phi$) as the sum of `measure`(`step`(ϕ, s), s') for each s', weighting by the edge probabilities $\delta(s, s')$. This is what `measure()` does.

Having computed the weighted sum e, `measure()` uses it to substitute for $x_{\phi s}$. Simply substituting e for $x_{\phi s}$ wouldn't necessarily eliminate $x_{\phi s}$, since e itself may contain an $x_{\phi s}$-term. So `measure()` calls `solve`($x_{\phi s} = e$) to get another expression $r_{\phi s}$, also equal to $x_{\phi s}$ but free of any $x_{\phi s}$-term (see section 3.4.3). `substitute`($x_{\phi s} := r_{\phi s}$) then eliminates $x_{\phi s}$ by substituting $r_{\phi s}$ for it in any cached expressions containing it.

Note that `measure()`'s return type is not strictly a scalar (number), but an expression that may include cached variables, e.g., $0.6x_{\phi s} + 0.1x_{\psi t} + 0.3$. (See the examples in section 3.5.) However, each call to `measure()` that creates a variable $x_{\phi s}$ later replaces it with an expression $r_{\phi s}$. Therefore any top-level non-recursive call is guaranteed to return a variable-free expression, i.e., a numerical probability (Corollary 5.33).

3.4.2 step()

Definition 3.3

step(ϕ, s):
  if $\phi = \mathcal{T}$ or $\phi = \mathcal{F}$:
    return ϕ
  else, $\phi = (\alpha\ ?\ \psi : \omega)$:
    if $\alpha = a$ (an LTL atom node):
      $\alpha' := \gamma(s, a)$
    else if $\alpha = X\tau$ (a next node):
      $\alpha' := \tau$
    else if $\alpha = \tau \mathcal{U} \upsilon$ (an until node):
      if $\forall t$ reachable from s, measure(υ, t) = 0:
        $\alpha' := \mathcal{F}$
      else:
        α' := or(step(υ, s), and(step(τ, s), $\tau \mathcal{U} \upsilon$))
    return cond(α', step(ψ, s), step(ω, s))

step() is the procedure that does the work in the recursion. Its fundamental property is that it returns a formula ϕ' such that any *plausible path* $P = (p_1 = s, p_2, p_3, \ldots)$ satisfies ϕ iff its suffix $P_2 = (p_2, p_3, \ldots)$ satisfies ϕ' (Theorem 5.47).

Plausible paths are defined in Definition 5.11. By Proposition 5.40, any Markov chain almost surely (with probability 1) follows a plausible path.

step() starts by checking whether $\phi = \mathcal{T}$ or $\mathcal{F}$. If so, then of course $P \models \phi$ iff $P_2 \models \phi$, and we can just return ϕ.

If ϕ is not a boolean, it must be an atom LTL-BDD of the form $(\alpha\ ?\ \psi : \omega)$, where α represents either an LTL atom, a next, or an until. In this case step() first constructs a formula α' such that $P_2 \models \alpha'$ iff $P \models \alpha$, and then returns $\phi' =$ cond(α', step(ψ, s), step(ω, s)). (The useful cond() operator is explained in the BDD appendix on page 108.) Why is this ϕ' guaranteed to be satisfied by P_2 iff ϕ is satisfied by P? Remember that ϕ is equivalent to ψ when P satisfies α, and to ω when it doesn't. Consider these cases separately:

- $P \models \alpha$. Then $P_2 \models \alpha'$. So, by the definition of cond(), in this case $P_2 \models \phi'$ iff

$P_2 \models \texttt{step}(\psi, s)$, which holds iff $P \models \psi$, which in this case is true iff $P \models \phi$. Done.

- $P \not\models \alpha$. Therefore $P_2 \not\models \alpha'$. So, $(P_2 \models \phi') \Leftrightarrow (P_2 \models \texttt{step}(\omega, s)) \Leftrightarrow (P \models \omega) \Leftrightarrow (P \models \phi)$.

So the only missing piece is the construction of α'. Consider again the three cases: $\alpha = a$, $\alpha = X\tau$, $\alpha = \tau \mathcal{U} \upsilon$. Suppose $\alpha = a$. Because s is a parameter to `step()`, we can just let $\alpha' := \gamma(s, a) =$ the truth value of a in s. For instance, if we know a is false in s, then it follows trivially that $P \models a$ iff $P_2 \models \mathcal{F}$.

The $\alpha = X\tau$ case is even simpler: by the definition of X, $P \models X\tau$ iff $P_2 \models \tau = \alpha'$. So the only tricky case is until.

Suppose $\alpha = \tau \mathcal{U} \upsilon$. `step()` begins by checking whether the until is *unrealizable*:

Definition 3.4 *An LTL until formula $(\psi \mathcal{U} \omega \ ? \mathcal{T} : \mathcal{F})$ is* ***unrealizable*** *from a state s if, $\forall t$ reachable from s (including s itself), $\texttt{prob}(t \models \omega) = 0$.*

If $\tau \mathcal{U} \upsilon$ is unrealizable from s, we let $\alpha' = \mathcal{F}$. This is a trick we use to avoid an inconvenient case which could prevent `measure()` from terminating. Note that our definition of unrealizable is distinctly probabilistic: there may still be (0-probability) paths from s which satisfy the until! (See the example in section 3.5.3.) In fact, the existence of such paths is why eliminating the until is useful. However, Corollary 5.44 asserts that no unrealizable until is satisfied by any plausible path. So, replacing unrealizable untils with $\mathcal{F}$ is safe.

In order to check if an until is unrealizable, `step()` calls `measure()`. Since it compares the result to 0, these calls to `measure()` had better return numbers, rather than expressions containing variables. Lemma 5.37 asserts that they do.

The α' computed for realizable untils is simpler than it looks. According to the definition of the until operator $\mathcal{U}$, $P \models \alpha = \tau \mathcal{U} \upsilon$ iff either $P \models \upsilon$, or both $P \models \tau$ and $P_2 \models \tau \mathcal{U} \upsilon$. Now, $P \models \upsilon$ iff $P_2 \models \texttt{step}(\upsilon, s)$, and $P \models \tau$ iff $P_2 \models \texttt{step}(\tau, s)$. So, $P \models \tau \mathcal{U} \upsilon$ iff either $P_2 \models \texttt{step}(\upsilon, s)$ or both $P_2 \models \texttt{step}(\tau, s)$ and $P_2 \models \alpha$: that is, $\alpha' = \texttt{or}(\texttt{step}(\upsilon, s), \texttt{and}(\texttt{step}(\tau, s), \alpha))$.

So in every case we can construct α', and therefore we can always return ϕ'.

Because `measure()` calls `step()` and vice versa, showing that they terminate (Proposition 5.18) constitutes a single proof.

3.4.3 solve()

Definition 3.5

solve($x = e$):
    $k :=$ the coefficient of x in e
    if $k = 1$:
        abort with an error (division by 0)
    else:
        return $\frac{e-kx}{1-k}$

solve($x = e$) solves the given equation for x and returns the solution r.

For example, solve($x_{\phi s} = 0.6x_{\phi s} + 0.1x_{\psi t} + 0.3$) returns $r_{\phi s} = 0.25x_{\psi t} + 0.75$. The reasoning is as follows:

$$\begin{aligned} x_{\phi s} &= 0.6x_{\phi s} + 0.1x_{\psi t} + 0.3 \\ x_{\phi s} - 0.6x_{\phi s} &= 0.1x_{\psi t} + 0.3 \\ 0.4x_{\phi s} &= 0.1x_{\psi t} + 0.3 \\ x_{\phi s} = r &= \frac{0.1x_{\psi t}+0.3}{0.4} \\ &= 0.25x_{\psi t} + 0.75 \end{aligned}$$

Or, more generally, suppose the coefficient of x in e is k. Then solve($x = e$) returns $\frac{e-kx}{1-k}$:

$$\begin{aligned} x &= e \\ x - kx &= e - kx \\ (1-k)x &= e - kx \\ x = r &= \frac{e-kx}{1-k} \end{aligned}$$

(If e contains no x-term, then $k = 0$ and r simply works out to e).

This computation of r will not work if $k = 1$. This case is painful, but after a long proof Lemma 5.46 asserts that, because step() eliminates unrealizable untils, any time measure() calls solve($x = e$), $k < 1$.

3.5 Example

Here I sketch measure()'s operation on Markov chain $\mathcal{M}_1$ (page 10) and three formulas from Figure 1.2 (page 11): Xa, $X(b\mathcal{U}a)$, and $\neg(\mathcal{T}\mathcal{U}\neg a)$.

This sketch will not follow `measure()` line for line, but will accurately follow the reasoning it uses. As these examples show, much of `measure()`'s behavior just consists of repeatedly calling `step`(ϕ, s) to answer the question: "What must be true one step later, in order for ϕ to be satisfied in state s?"

3.5.1 `measure`(Xa, s_1)

First consider a call to `measure`(Xa, s_1), to compute `prob`($s_1 \models Xa$): the probability that a path from s_1 satisfies Xa. Execution proceeds as follows:

1. Create a new variable x_1 and cache it as the solution. This is so that any recursive calls to `measure`(Xa, s_1) return the cached variable; otherwise such a recursive call would lead to infinite regress.

2. Next `measure()` will compute an expression e_1, representing the probability that $\phi' =$ `step`(Xa, s_1) is satisfied in the next state, and equate it to x_1. The first step is to compute ϕ'. `step`(Xa, s_1) is simply a, by the following reasoning: for any path $P = (p_1 = s_1, p_2, p_3, \ldots)$ to satisfy Xa, its suffix $P_2 = (p_2, p_3, \ldots)$ must satisfy a.

3. Now we make recursive calls to compute `prob`($s' \models \phi'$) for every possible successor state s' to s_1: $x_2 =$ `measure`(a, s_1), $x_3 =$ `measure`(a, s_2), $x_4 =$ `measure`(a, s_3). e_1 can then be computed as a weighted sum of their return values, weighted by the edge weights from s_1 to s': $x_1 = e_1 = 0.5x_2 + 0.4x_3 + 0.1x_4$.

4. The first recursive call, $x_2 =$ `measure`(a, s_1), returns 1, since (by $\mathcal{M}_1$'s definition in Figure 1.1) a is true in s_1. (As shown in Definition 3.2, our real `measure()` algorithm does not detect this immediately, instead making a further call to `step()`. This is only to make some of our proofs simpler; the outcome is the same.)

5. Similarly, $x_3 =$ `measure`(a, s_2) also returns 1, and $x_4 =$ `measure`(a, s_3) returns 0.

6. So we have: $x_1 = e_1 = 0.5x_2 + 0.4x_3 + 0.1x_4 = 0.5 + 0.4 = 0.9$.

And we have correctly computed that `prob`($s_1 \models Xa$) = 0.9.

3.5.2 measure($X(b\mathcal{U}a)$, s_1)

Now, measure($X(b\mathcal{U}a)$, s_1). This example shows the purpose of the realizability check in step().

1. Cache a variable x_5 representing the solution.
2. Compute step($X(b\mathcal{U}a)$, s_1): what must be true after one step for $X(b\mathcal{U}a)$ to be true now? As seen in the previous example, for any ϕ and s, step($X\phi$, s) is just ϕ. So step($X(b\mathcal{U}a)$, s_1) = $b\mathcal{U}a$.
3. So, we make recursive calls x_6 = measure($b\mathcal{U}a$, s_1), x_7 = measure($b\mathcal{U}a$, s_2), x_8 = measure($b\mathcal{U}a$, s_3), and equate x_5 to a weighted sum as before: $x_5 = 0.5x_6 + 0.4x_7 + 0.1x_8$.
4. x_6 = measure($b\mathcal{U}a$, s_1) calls step($b\mathcal{U}a$, s_1), which begins by checking whether the until is realizable from s_1. To answer this requires calls to measure(a, t) for every state t reachable from s_1. measure(a, s_1) returns 1 (> 0), so the until is realizable.
5. Therefore, step($b\mathcal{U}a$, s_1) is computed to be just $\mathcal{T}$, by the following reasoning: By the definition of $\mathcal{U}$, $\phi\mathcal{U}\psi$ is true iff either ψ is true, or ϕ is true and in the next state $\phi\mathcal{U}\psi$ is true. That is, $P \models \phi\mathcal{U}\psi$ iff $P \models \psi \vee (\phi \wedge X(\phi\mathcal{U}\psi))$. So since a and b are both true in s_1, we have: step($b\mathcal{U}a$, s_1) = step($a \vee (b \wedge X(b\mathcal{U}a))$, s_1) = step(a, s_1) $\vee$ (step(b, s_1)$\wedge$ step($X(b\mathcal{U}a)$, s_1)) = $\mathcal{T} \vee (\mathcal{T} \wedge (b\mathcal{U}a)) = \mathcal{T}$.
6. So since step($b\mathcal{U}a$, s_1) = $\mathcal{T}$, x_6 = measure($b\mathcal{U}a$, s_1) = 1. And similarly, since a is also true in s_2, step($b\mathcal{U}a$, s_2) = $\mathcal{T}$ and x_7 = measure($b\mathcal{U}a$, s_2) = 1.
7. The case of x_8 = measure($b\mathcal{U}a$, s_3) is different, because the realizability check fails. The only state reachable from s_3 is s_3 itself, and a is false in s_3. That is, $b\mathcal{U}a$ is unrealizable from s_3. Therefore, step($b\mathcal{U}a$, s_3) is computed as just $\mathcal{F}$, and $x_8 = 0$.
8. So again we have: $x_5 = 0.5x_6 + 0.4x_7 + 0.1x_8 = 0.5 + 0.4 = 0.9$.

Note that the realizability check was essential here for the case of x_8 = measure($b\mathcal{U}a$, s_3). Without it, step($b\mathcal{U}a$, s_3) would be computed as step($a \vee (b \wedge X(b\mathcal{U}a))$, s_3), and since b is true and a false in s_3, this would reduce to $\mathcal{F} \vee (\mathcal{T} \wedge (b\mathcal{U}a)) = b\mathcal{U}a$. So measure($b\mathcal{U}a$, s_3) would end up trying to solve the degenerate equation $x_8 = x_8$.

3.5.3 measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_2)

This last example shows how measure() effectively ignores *implausible* paths (Definition 5.11). The formula $\neg(\mathcal{T}\mathcal{U}\neg a)$ asserts that $\neg a$ never occurs, i.e., that a is always true. There are paths in $\mathcal{M}_1$ satisfying this formula, such as $(s_1, s_1, s_1, \ldots)$, but they are all implausible; eventually any plausible path will enter s_3, where a is false.

1. Cache variable x_9, representing measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_2).

2. Check whether $\mathcal{T}\mathcal{U}\neg a$ is realizable from s_2. This requires recursive calls to check whether $\neg a$ is satisfied with non-zero probability from any state reachable from s_2. s_3 is such a state, so the until is realizable.

3. Therefore, step($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_2) = step($\neg((\neg a) \vee (\mathcal{T} \wedge X(\mathcal{T}\mathcal{U}\neg a)))$, s_2) = $\neg((\neg$ step(a, s_2)) $\vee$ ($\mathcal{T}\mathcal{U}\neg a$)) = $\neg((\neg\mathcal{T}) \vee (\mathcal{T}\mathcal{U}\neg a)) = \neg(\mathcal{T}\mathcal{U}\neg a)$. So, x_9 is computed as a weighted sum $x_9 = 0.7x_{10} + 0.3x_{11}$, where x_{10} = measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_1) and x_{11} = measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_3).

4. Since s_3 is also reachable from s_1, the until is realizable from s_1, and therefore step($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_2) is also $\neg(\mathcal{T}\mathcal{U}\neg a)$. So the call to x_{10} = measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_1) results in three recursive calls to measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s) for the three states s. All of these are calls that have already been assigned variables, so the solutions are retrieved from the cache. The resulting equation is: $x_{10} = 0.5x_{10} + 0.4x_9 + 0.1x_{11}$.

5. Finally, the call to x_{11} = measure($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_3). Again the until is realizable. So step($\neg(\mathcal{T}\mathcal{U}\neg a)$, s_3) = step($\neg((\neg a) \vee (\mathcal{T} \wedge X(\mathcal{T}\mathcal{U}\neg a)))$, s_3) = $\neg((\neg$ step(a, s_3)) $\vee$ ($\mathcal{T}\mathcal{U}\neg a$)) = $\neg((\neg\mathcal{F}) \vee (\mathcal{T}\mathcal{U}\neg a)) = \neg(\mathcal{T}) = \mathcal{F}$. Therefore, $x_{11} = 0$.

6. So we end up with the following system of equations: $x_9 = 0.7x_{10} + 0.3x_{11}$; $x_{10} = 0.5x_{10} + 0.4x_9 + 0.1x_{11}$; $x_{11} = 0$. Solving these just yields $x_9 = x_{10} = x_{11} = 0$, implying (correctly) that the probability that a is always true is 0 from any state in $\mathcal{M}_1$.

Chapter 4

Complexity

In this chapter I look for bounds on the running time of a top-level call to `measure`(ϕ, s). Proofs are again delegated to Chapter 5.

The main aim of this work has been to find an algorithm and prove it correct. The complexity results which follow, like the MCMC implementation described in Chapter 6, are unrefined and more likely to be useful as a basis for future work (see section 8.2) than as conclusive results in themselves.

4.1 Approach

A top-level call to `measure`(ϕ, s) results in a number of further recursive `measure`() calls. `measure`() caches its results: a second call with the same parameters returns immediately. Also, calls passing in $\mathcal{T}$ or $\mathcal{F}$ just return 1 or 0. So for the purpose of running time, we can ignore these types of calls and consider only the nontrivial calls, i.e., calls with distinct input pairs (ψ, t) where ψ is not a boolean. Then, by computing a bound on the maximum possible number of distinct nontrivial input pairs, and another on the amount of time spent in each nontrivial call (not counting its recursive calls), we can multiply these to get a bound on the total amount of time required by `measure`(ϕ, s).

The number of distinct input pairs is the number of states in the input Markov chain $\mathcal{M}$ (easy to count), times the number of distinct LTL-BDDs passed to `measure`() by recursive calls (not so easy). So most of the analysis which follows will look for bounds on:

- the number of distinct LTL-BDDs passed to recursive `measure()` calls, and
- the amount of time spent in each call to `measure()`.

4.2 Input measurements

First, we define some input measurements that will figure in our bounds. See also the examples in Figure 4.1 below. (Of course these measurements are dependent on the inputs ϕ and $\mathcal{M}$. Normally the formula or machine in question will be obvious, so for brevity we will simply write, e.g., u rather than $u(\phi)$.)

Definition 4.1

$\lvert\phi\rvert$	$\stackrel{def}{=}$	syntactic length of ϕ: # operators, booleans and atoms (not parentheses)
$\lvert\Sigma\rvert$	$\stackrel{def}{=}$	# distinct LTL atoms in ϕ
$\lvert A\rvert$	$\stackrel{def}{=}$	# distinct LTL-BDD atoms (= # distinct LTL atoms, nexts & untils) in ϕ
o	$\stackrel{def}{=}$	# (not necessarily distinct) LTL atom occurrences in ϕ
u	$\stackrel{def}{=}$	# distinct (unbounded) untils in ϕ
d_U	$\stackrel{def}{=}$	`udepth`(ϕ) (Defn 5.2): maximum nesting of (unbounded) untils in ϕ
d_X	$\stackrel{def}{=}$	`xdepth`(ϕ) (Defn 5.3): max nesting of nexts in ϕ, counting atoms as nexts
d_i	$\stackrel{def}{=}$	maximum nesting of nexts (as for d_X) inside any until in ϕ
$\lvert S\rvert$	$\stackrel{def}{=}$	# states in $\mathcal{M}$
E	$\stackrel{def}{=}$	# edges (prob > 0) in $\mathcal{M}$

Given an input ϕ and $\mathcal{M}$, these values are all easy to measure. Using them, we want to express big-O bounds on the following less easily measured values:

Definition 4.2

n	$\stackrel{def}{=}$	# nontrivial `measure()` calls
$\lvert e\rvert$	$\stackrel{def}{=}$	avg # variables per cached expression, over entire execution of `measure`(ϕ, s)
$\lvert B\rvert$	$\stackrel{def}{=}$	# distinct LTL-BDDs passed to `measure()` (apart from $\mathcal{T}$ and $\mathcal{F}$)
m	$\stackrel{def}{=}$	running time of a single `measure()` call, not counting recursive calls
R	$\stackrel{def}{=}$	total running time of `measure`(ϕ, s), for any s in $\mathcal{M}$

As argued above, $R = n \cdot m$ and $n \leq |B| \cdot |S|$, so R is $O(|B| \cdot |S| \cdot m)$. So we want bounds on $|B|$ and m.

ϕ	$\lvert\phi\rvert$	$\lvert\Sigma\rvert$	$\lvert A\rvert$	o	u	$d_{\mathcal{U}}$	d_X	d_i
$\mathcal{T}$	1	0	0	0	0	0	0	0
$\neg a$	2	1	1	1	0	0	1	0
$\mathcal{T}\mathcal{U}c$	3	1	2	1	1	1	1	1
$\mathcal{T}\mathcal{U}^{\leq 10}c$	5	1	11	11	0	0	11	0
$(Xa) \wedge a$	4	1	2	2	0	0	2	0
$X(a \wedge b)$	4	2	3	2	0	0	2	0
$X(b\mathcal{U}X\neg a)$	6	2	5	2	1	1	3	2
$(Xc)\mathcal{U}(a \wedge Xb)$	7	3	6	3	1	1	2	2
$(\neg c)\mathcal{U}\neg(\mathcal{T}\mathcal{U}b)$	7	2	4	2	2	2	1	1
$(\neg c)\mathcal{U}^{\leq 3}\neg(\mathcal{T}\mathcal{U}^{\leq 2}b)$	9	2	7	15	0	0	6	0
$(XXc)\mathcal{U}(b \wedge Xc)$	8	2	5	3	1	1	3	3
$(b\mathcal{U}c) \vee XX\neg(b\mathcal{U}c)$	10	2	5	4	1	1	3	1
$X((a\mathcal{U}b) \vee X(b\mathcal{U}Xc))$	10	3	8	4	2	1	4	2
$X((a\mathcal{U}b)\mathcal{U}X\neg(b\mathcal{U}Xc))$	11	3	9	4	3	2	4	3
$(a\mathcal{U}b)\mathcal{U}\neg(c\mathcal{U}XX(b\mathcal{U}Xa))$	13	3	10	5	4	3	4	4
$(c\mathcal{U}XXXXXXXXa)\mathcal{U}Xb$	13	3	13	3	2	2	8	8
$((XXXXc)\mathcal{U}XXa)\mathcal{U}XXXb$	14	3	14	3	2	2	5	5

Figure 4.1: The variables we use to measure formula size, defined above. For example, $|\Sigma|$ counts the number of distinct LTL atoms in ϕ.

4.3 Bounds on $|B|$

Here I describe three bounds on $|B|$: one fully general but unreassuring, one for the special case of formulas containing no untils (or only bounded untils), and one for formulas containing no nested untils.

4.3.1 Bound 1: a crude upper bound on $|B|$

The simplest upper bound on $|B|$, the number of nontrivial LTL-BDDs passed to `measure()`, is obtained by counting the total possible number of LTL-BDDs that can be created from the input formula ϕ. By Corollary 5.17, no more than $2^{2^{|A|}}$ LTL-BDDs are created during a call to `measure()`, where $|A|$ is the number of LTL-BDD

atoms in ϕ as defined above. Therefore we have **bound 1** on $|B|$:

$|B| \leq 2^{2^{|A|}}$. (Corollary 5.50)

$|A| \leq |\phi|$, so this bound says that the number of LTL-BDDs passed to `measure()` is no more than doubly exponential in the size of the formula.

Bound 1 is of limited use, because $2^{2^{|A|}}$ grows very fast. For example, if $\phi = (Xc)\mathcal{U}(a \wedge Xb)$, then $|A| = 6$, and $2^{2^{|A|}} = 2^{2^6} = 2^{64}$, about 18 trillion trillion. As shown in Figure 4.2, the actual number of LTL-BDDs passed to `measure()` on this input in MCMC is 7. In fact, variants like $(Xb)\mathcal{U}((Xa) \vee c)$ all pass between 5 and 7 distinct LTL-BDDs. This suggests that our $2^{2^{|A|}}$ bound is not tight.

Formula	# BDDs passed to `measure()`			
	Bound 1	Bound 2	Bound 3	MCMC
ϕ	$2^{2^{\|A\|}}$	$d_X 2^o$	$2^{(d_X+d_i-1)\|\Sigma\|}3^u$	
$\mathcal{T}$	2	0	1	1
$\neg a$	4	2	1	3
$\mathcal{T}\mathcal{U}c$	16	–	6	4
$\mathcal{T}\mathcal{U}^{\leq 10}c$	2^{2048}	22528	1024	13
$(Xa) \wedge a$	16	8	2	4
$X(a \wedge b)$	256	8	4	4
$X(b\mathcal{U}X\neg a)$	2^{32}	–	768	7
$(Xc)\mathcal{U}(a \wedge Xb)$	2^{64}	–	1536	7
$(\neg c)\mathcal{U}\neg(\mathcal{T}\mathcal{U}b)$	65536	–	–	6
$(\neg c)\mathcal{U}^{\leq 3}\neg(\mathcal{T}\mathcal{U}^{\leq 2}b)$	2^{128}	196608	1024	23
$(XXc)\mathcal{U}(b \wedge Xc)$	2^{32}	–	3072	8
$(b\mathcal{U}c) \vee XX\neg(b\mathcal{U}c)$	2^{32}	–	192	7
$X((a\mathcal{U}b) \vee X(b\mathcal{U}Xc))$	2^{256}	–	294912	11
$X((a\mathcal{U}b)\mathcal{U}X\neg(b\mathcal{U}Xc))$	2^{512}	–	–	17
$(a\mathcal{U}b)\mathcal{U}\neg(c\mathcal{U}XX(b\mathcal{U}Xa))$	2^{1024}	–	–	43
$(c\mathcal{U}XXXXXXXXa)\mathcal{U}Xb$	2^{8192}	–	–	135
$((XXXXc)\mathcal{U}XXa)\mathcal{U}XXXb$	2^{16384}	–	–	16

Figure 4.2: Our upper bounds on $|B|$ are not tight.

4.3.2 Bound 2: when $d_{\mathcal{U}} = 0$, $|B| \leq d_X 2^o$

In practice, MCMC can often handle large formulas (with 20+ LTL-BDD atoms). In particular, formulas with no untils ($u = d_{\mathcal{U}} = 0$ — recall that this includes formulas with only bounded untils) and formulas without nested untils ($d_{\mathcal{U}} \leq 1$) are generally solved quickly. Since these categories include most formulas used as model checking specifications, tighter bounds for these cases are worth having. We can obtain such bounds by the following reasoning:

1. Proposition 5.31 asserts that, if a top-level call `measure`(ϕ, s) results in a recursive call to `measure`(ψ, t), then ϕ is reducible to ψ (Definition 5.7), i.e., ψ is the result of some number of applications of `step`() to ϕ, or to the right-hand argument of an until within ϕ. For example, if $\phi = a \wedge (b\mathcal{U}XXc)$, then ψ must be the result of applying `step`() (repeatedly) to ϕ or to XXc.

2. The right-hand argument of any until within ϕ is no larger than ϕ, by any of the measures in Definition 4.1, so a bound on the number of LTL-BDDs resulting from applications of `step`() to ϕ will also lead to a big-O bound on $|B|$. Therefore it suffices to count the number of LTL-BDDs which can result from applying `step`() to ϕ, i.e., the number of different possible values of $\texttt{step}^k(\phi, P)$ for some path P and $k \geq 0$ (Definition 5.4).

First, suppose ϕ contains no unbounded untils ($u = d_{\mathcal{U}} = 0$). Then each application of `step`() to ϕ strips off at least one X (Proposition 5.25). Also, if `step`() is repeatedly applied to ϕ, each occurrence of an LTL atom in ϕ is checked only once (or not at all) by `step`(). Using these observations we can derive **bound 2**:

When $d_{\mathcal{U}} = 0$, $|B| \leq d_X 2^o$. (Proposition 5.51)

d_X and o are $\leq |\phi|$, so in loose terms this says that for until-free formulas, $|B|$ is singly exponential in $|\phi|$.

This is a fairly tight bound: we can construct formulas for which $|B|$ approaches it. For example, for a ϕ of the form $(a \wedge X^i a) \vee (b \wedge X^i b) \vee \ldots$ (where X^i stands for i applications of X), $|B|$ is about $d_X 2^{\frac{o}{2}}$.

4.3.3 Bound 3: when $d_{\mathcal{U}} \leq 1$, $|B|$ is $O(3^u 4^{|\phi|^2})$

Now suppose ϕ is an until, and ϕ's subformulas contain no untils ($u = d_{\mathcal{U}} = 1$). Then we can show that, for any path P and $k \geq d_X$, $\texttt{step}^k(\phi, P) = \mathcal{T}$, $\mathcal{F}$, or $\texttt{step}^{d_X}(\phi, Q)$

for some other path Q (Lemma 5.53). And in this case we can show that therefore $|B|$ is $O(2^{(d_X-1)|\Sigma|})$ (Corollary 5.55).

Next, suppose $d_{\mathcal{U}} = 1$ but ϕ itself is not an until. Then it must contain u (unnested) sub-untils. Using Lemma 5.53, we can derive **bound 3**:

$|B|$ is $O(2^{(d_X+d_i-1)|\Sigma|}3^u)$. (Theorem 5.56)

Since d_X, d_i, Σ and u are all $\leq |\phi|$, this bound says that for formulas with no nested untils, $|B|$ is no more than singly exponential in the size of the formula. As shown in Figure 4.2, this is a big improvement on bound 1 when it applies, but still not a reliable indicator of complexity on many inputs. However:

Conjecture 4.3 *When $d_{\mathcal{U}} = 1$, in the worst case, bound 3 (Theorem 5.56) is tight: no significantly tighter bound on $|B|$ exists in terms of d_X, d_i, $|\Sigma|$ and u.*

4.3.4 Conjecture: $|B|$ is $O(2^{|\phi|})$

Any bound is only as useful as the measurements in terms of which it is expressed. For example, even if worst-case examples exist for any given d_X, d_i, $|\Sigma|$ and u such that bound 3 is tight, it may be that any such worst cases involve very long formulas. It is reasonable to want a general bound directly in terms of $|\phi|$, the length of the formula. This I have not been able to find.

Conjecture 4.4 *There exists a general bound on $|B|$ expressible in terms of $|\phi|$ which is significantly tighter than bounds 1 or 3.*

And even (optimistically):

Conjecture 4.5 *On typical, non-degenerate inputs, the expected value of $|B|$ is $O(2^{|\phi|})$.*

A natural way to start looking for these improved bounds is to try to generalize bound 2. For example, if the effects on $|B|$ of each atom, boolean operator ($\neg/\wedge/\vee$), and temporal operator ($X/\mathcal{U}$) could be quantified, they could be combined into a bound on $|B|$ in terms of $|\phi|$.

4.4 A crude upper bound on m

m, the time taken by a nontrivial call to `measure`(ϕ, s) (not counting time spent within recursive calls), can be broken up into three parts:

1. time to compute `step`(ϕ, s)
2. time to make the recursive `measure`() calls, not counting the time spent within each call
3. time taken by `solve`($x_{\phi s} = e$) and `substitute`($x_{\phi s} := r_{\phi s}$)

We will analyze these separately.

4.4.1 step()

`step`(ϕ, s) performs three types of tasks: recursive calls to `step`(), calls to BDD operations (`and`(), `or`(), `cond`()), and determining whether an until $\tau \mathcal{U} v$ is realizable from s (Definition 3.4). Consider the time required for each task separately.

The recursive calls to `step`() are all on subformulas of ϕ. Therefore, these calls will result in one `step`() call per node in ϕ. By Proposition A.1, the number of nodes in ϕ is $O(2^{|A|})$. Therefore, the total time required by `step`(ϕ, s) is $O(2^{|A|} \cdot m')$, where m' is the time required by each recursive `step`() call (not counting its own recursive calls).

By Proposition A.3, `and`(), `or`(), and `cond`() are all $O(2^{|A|})$. Aside from its recursive calls, each call to `step`() makes at most four calls to these BDD operations. Therefore the time taken by BDD operations in each `step`() call is $O(2^{|A|})$.

Determining whether $\tau \mathcal{U} v$ is realizable from s is faster if we do some preprocessing. We can begin the top-level call to `measure`(ϕ, s) by calling `measure`(v, t) for each until $\tau \mathcal{U} v$ in ϕ and each state t in $\mathcal{M}$. It is then a standard graph reachability problem to determine, for each $\tau \mathcal{U} v$ and t, whether $\tau \mathcal{U} v$ is realizable from t; i.e., whether there is any state t' reachable from t such that `measure`(v, t') > 0. With this information stored, realizability can be checked by future `step`() calls in constant time.

How long does this preprocessing take? The time for the `measure`(v, t) calls can be ignored here, since m excludes the time for other `measure`() calls. For an implementation (like MCMC) which simply represents $\mathcal{M}$ as an $|S| \times |S|$ matrix, the time required for the reachability computation is at worst $O(|S|^2 \cdot u)$. This is a reasonable cost, since the time just to read in such a matrix is $O(|S|^2)$. In more efficient (e.g., graph-based or BDD-based) Markov chain representations, the reachability computation can be brought down proportional to E, the number of

edges in $\mathcal{M}$. In either case, as we will see, this preprocessing cost is not a limiting factor.

So apart from some preprocessing, the total time required for a call to `step`(ϕ, s) is: (the number of recursive `step`() calls)·(the time required for each call)= $O(2^{|A|}) \cdot O(2^{|A|}) = O(4^{|A|})$.

4.4.2 Recursive `measure`() calls

Recall again that we are not counting the time spent within these calls, only the time required to make them: that is, one time unit per call.

`measure`(ϕ, s) makes a recursive call to `measure`(ϕ', s') for each outgoing edge with non-zero probability ($\delta(s, s') > 0$). In MCMC's simple matrix representation, `measure`() needs to go through the entire s row to find the valid s' candidates, so finding them takes time $|S|$. In more efficient representations, the outgoing edges can be looked up directly, bringing the average time down to $O(\frac{E}{|S|})$, i.e., the average degree of $\mathcal{M}$'s graph.

4.4.3 `solve()/substitute()`

This is another part of the algorithm where we are still faced with a big gap between our worst-case bounds and observed performance on sample inputs.

The worst case occurs when $|e|$, the average number of variables in each cached solution expression, is proportional to the total number of variables created. The number of variables created is just n, the number of nontrivial calls to `measure`(). Suppose $|e| \approx n$. Then, on average, `solve`() must solve an equation in n variables, which is $O(n)$. In fact this can be improved to $O(1)$, but consider `substitute`(). The number of steps required to substitute an expression of n variables into each of n cached solution expressions is $O(n^2)$.

Recall that n can certainly be expected to be $\geq |S|$, the number of states in $\mathcal{M}$. So, if the running time for each `measure`() call, m, is n^2, then even for a simple formula ϕ, the total running time R is (# calls) · (time per call) = $\Omega(|S|) \cdot \Omega(|S|^2) = \Omega(|S|^3)$. This is terrible.

One way to look at this is to observe that these `substitute`() calls are effectively solving a system of n linear equations in n variables. Naive algorithms to do this are indeed $O(n^3)$, and even fast special-case algorithms only approach $O(n^2)$. From this

point of view a $O(|S|^3)$ running time, or at least $O(|S|^2)$, looks like a legitimate worst case.

However, in practice $|e|$ is usually much lower than n. In fact, we can show that on many realistic inputs $|e| \leq 1$, so that `solve()` and `substitute()` run in constant time and are not a significant factor in m.

Suppose ψ and $\psi' = \texttt{step}(\psi, t)$ are not mutually reducible. Then, by Corollary 5.35, $\texttt{measure}(\psi, t)$ returns a number. And Corollary 5.36 further asserts that the call made by $\texttt{measure}(\psi, t)$ to `substitute()` only performs a single substitution. This gives us Corollary 5.57: when ψ and ψ' are not mutually reducible, the running times of `solve()` and `substitute()` are $O(1)$.

A common case in which ψ and ψ' are not mutually reducible is when $u = 0$: ψ contains no unbounded untils. When $u = 0$, Proposition 5.26 implies that ψ and ψ' are not mutually reducible, unless $\psi' = \psi$. Either way `solve()` and `substitute()` are $O(1)$.

More generally, the mutually-reducible equivalence relation partitions the total set of created LTL-BDDs B into j equivalence classes, each containing $|B_j|$ mutually reducible LTL-BDDs. Lemma 5.32 implies that each call to `substitute()` or `solve()` deals only with variables whose formulas belong to the same equivalence class (where, e.g., the variable of $x_{\phi s}$ is ϕ). So, rather than solving a single system of $n \leq |B| \cdot |S|$ equations in n variables, `measure()` solves j systems, each of $n_j \leq |B_j| \cdot |S|$ equations in n_j variables.

The expected time to solve this type of sparse matrix varies widely (between $O(n^3)$ and $O(n)$) depending on the size of the B_j's. When $u \geq 1$, the worst case may indeed approach $O(n^3)$. However, we will conjecture that $|e|$ is usually $O(\frac{E}{|S|})$:

Conjecture 4.6 *On typical, non-degenerate inputs, the expected running time of each call to* `solve()` *and* `substitute()` *is* $O((\frac{E}{|S|})^2)$.

4.4.4 Adding it up

Our worst-case bound for an efficient implementation of m, then, comes to:

(preprocessing time, averaged over n calls)
$+$ (time for `step`())
$+$ (time to make recursive `measure`() calls)
$+$ (time for `solve`() & `substitute`())

$= O(\frac{E}{n}) + O(4^{|A|}) + O(\frac{E}{|S|}) + O(n^2)$
$\leq O(\frac{E}{|B|\cdot|S|}) + O(4^{|A|}) + O(\frac{E}{|S|}) + O((|B| \cdot |S|)^2)$
$\leq O(4^{|A|}) + O(\frac{E}{|S|}) + O((|B| \cdot |S|)^2)$

In general, this expression is dominated by the $O((|B|\cdot|S|)^2)$ time for `substitute`(): quadratic in the number of LTL-BDDs created, and in the number of states in $\mathcal{M}$.

In the $u = 0$ case, it reduces to $O(4^{|A|} + \frac{E}{|S|})$: exponential in the number of distinct LTL-BDD atoms (LTL atoms, nexts and untils) in ϕ, linear in the average degree of $\mathcal{M}$.

In the event that Conjecture 4.6 holds, the bound on m reduces to $O(4^{|A|} + (\frac{E}{|S|})^2)$: exponential in the number of LTL-BDD atoms, quadratic in the average degree of $\mathcal{M}$.

4.5 Conclusions

The main objective of a complexity analysis, apart from theoretical interest, is to estimate the practical limits on an algorithm's input size. As Figure 4.2 makes clear, our analysis contains far too many special cases, conjectures, loose bounds, and exponentials to be useful for this purpose.

A more sensible way to gauge the usefulness of the `measure`() algorithm is to try it out on some realistic examples. To do this properly will require a more mature implementation than MCMC, but even running such examples in MCMC seems unlikely to produce more pessimistic results than our theoretical worst-case bounds. Such tests are an obvious area for future work (Chapter 8.2).

In the absence of realistic tests, we can only summarize the analysis with the following bounds on R, the total running time of a call to `measure`(ϕ, s).

Recall that $R = n \cdot m$, and $n \leq |B| \cdot |S|$:

1. Worst case when $u = 0$: R is $O(d_X 2^o(|S| \cdot 4^{|A|} + E))$: in rough terms, linear in E, singly exponential in $|\phi|$.

2. Conjectured expected case when $u \geq 1$ (assuming Conjectures 4.5 and 4.6 hold): $O(|S| \cdot 2^{|\phi|}(4^{|A|} + (\frac{E}{|S|})^2))$: roughly, linear in $|S|$, quadratic in $\frac{E}{|S|}$, singly

exponential in $|\phi|$.

3. Theoretical worst case: R is $O(8^{2^{|A|}} \cdot |S|^3)$.

Chapter 5

Proofs

This chapter contains the proofs relied on by Chapters 3 and 4.

The number and hairiness of these proofs is unfortunate. There may be a simpler way to prove the correctness results (in particular, Lemma 5.46: in any call to `solve`$(x_{\phi s} = e)$. the coefficient of $x_{\phi s}$ in e is < 1), but so far it has escaped me.

The main results are in the following sections:

- 5.7 (pages 68-71): a plausible path has a prefix determining ϕ
- 5.8 (pages 71-80): the big combined induction, proving the correctness of `measure`() and `step`()

5.1 Some definitions

The precise definition of the subformulas of an LTL-BDD ϕ is important, because it will underpin our many structural inductions:

Definition 5.1 *The **subformulas** of an LTL-BDD are as follows:*

- $\mathcal{T}/\mathcal{F}$*: no subformulas*
- $(a\ ?\ \psi\ :\ \omega)$*:* ψ *and* ω
- $(X\tau\ ?\ \psi\ :\ \omega)$*:* τ*,* ψ *and* ω
- $(\tau \mathcal{U} \upsilon\ ?\ \psi\ :\ \omega)$*:* τ*,* υ*,* ψ *and* ω

Note that $X\tau$ is *not* a subformula of $(X\tau \ ? \ \psi : \omega)$, since then $(X\tau \ ? \ \mathcal{T} : \mathcal{F})$ would be a subformula of itself, killing our structural inductions. Similarly, $\tau \mathcal{U} \upsilon$ is not a subformula of $(\tau \mathcal{U} \upsilon \ ? \ \psi : \omega)$.

We will sometimes refer to "the subformulas of an LTL formula ϕ", referring of course to the subformulas of the LTL-BDD representing ϕ.

Definition 5.2

$$\begin{aligned}
\texttt{udepth}(\mathcal{T}) = \texttt{udepth}(\mathcal{F}) &\stackrel{def}{=} 0 \\
\texttt{udepth}(a \ ? \ \psi : \omega) &\stackrel{def}{=} \max(\texttt{udepth}(\psi), \texttt{udepth}(\omega)) \\
\texttt{udepth}(X\tau \ ? \ \psi : \omega) &\stackrel{def}{=} \max(\texttt{udepth}(\tau), \texttt{udepth}(\psi), \texttt{udepth}(\omega)) \\
\texttt{udepth}(\tau \mathcal{U} \upsilon \ ? \ \psi : \omega) &\stackrel{def}{=} \max(\texttt{udepth}(\tau) + 1, \texttt{udepth}(\upsilon) + 1, \texttt{udepth}(\psi), \texttt{udepth}(\omega))
\end{aligned}$$

$\texttt{udepth}(\phi)$ computes the maximum depth of nested untils in ϕ. Examples: $\texttt{udepth}(a \vee c) = 0$, $\texttt{udepth}(a \wedge (b \mathcal{U} c) \wedge (a \mathcal{U} c)) = 1$, $\texttt{udepth}((b \mathcal{U} a) \mathcal{U} c) = 2$.

As we saw in section 2.3.4, bounded untils are encoded without pure untils, so they are not counted by udepth(): for any k, $\texttt{udepth}(\tau \mathcal{U}^{\leq k} \upsilon) = \max(\texttt{udepth}(\tau), \texttt{udepth}(\upsilon))$.

Definition 5.3

$$\begin{aligned}
\texttt{xdepth}(\mathcal{T}) = \texttt{xdepth}(\mathcal{F}) &\stackrel{def}{=} 0 \\
\texttt{xdepth}(a \ ? \ \psi : \omega) &\stackrel{def}{=} \max(1, \texttt{xdepth}(\psi), \texttt{xdepth}(\omega)) \\
\texttt{xdepth}(X\tau \ ? \ \psi : \omega) &\stackrel{def}{=} \max(\texttt{xdepth}(\tau) + 1, \texttt{xdepth}(\psi), \texttt{xdepth}(\omega)) \\
\texttt{xdepth}(\tau \mathcal{U} \upsilon \ ? \ \psi : \omega) &\stackrel{def}{=} \max(\texttt{xdepth}(\tau), \texttt{xdepth}(\upsilon), \texttt{xdepth}(\psi), \texttt{xdepth}(\omega))
\end{aligned}$$

$\texttt{xdepth}(\phi)$ computes the maximum depth of nested nexts in ϕ, counting atoms as nexts: $\texttt{xdepth}(T) = 0$, $\texttt{xdepth}(a) = 1$, $\texttt{xdepth}(X(a \wedge Xb)) = 3$, $\texttt{xdepth}(X((Xa) \mathcal{U} (b \vee XXa))) = 4$.

Because a bounded until $\tau \mathcal{U}^{\leq k} \upsilon$ is encoded with k nexts ($k - 1$ around the τ), $\texttt{xdepth}(\tau \mathcal{U}^{\leq k} \upsilon) = \max(\texttt{xdepth}(\tau) - 1, \texttt{xdepth}(\upsilon)) + k$.

Definition 5.4 *For a given LTL-BDD ϕ and path $P = (p_1, p_2, p_3, \ldots)$ in a labeled Markov chain $\mathcal{M}$:*

$$\begin{aligned} \texttt{step}^0(\phi,\, P) &\stackrel{def}{=} \phi \\ \texttt{step}^{n>0}(\phi,\, P) &\stackrel{def}{=} \texttt{step}^{n-1}(\texttt{step}(\phi,\, p_1),\, P_2 = (p_2, p_3, \ldots)) \end{aligned}$$

So $\texttt{step}^1(\phi,\, P) = \texttt{step}(\phi,\, p_1)$, $\texttt{step}^2(\phi,\, P) = \texttt{step}(\texttt{step}(\phi,\, p_1),\, p_2)$, etc. Intuitively, $\texttt{step}^n(\phi,\, P)$ is the result of the n^{th} recursive call to `step()` made by $\texttt{measure}(\phi,\, p_1)$ as it traverses P. The following alternative form can also be useful:

Proposition 5.5 $\texttt{step}^{n>0}(\phi,\, P = (p_1, p_2, \ldots)) = \texttt{step}(\texttt{step}^{n-1}(\phi,\, P),\, p_n)$.

Proof From the above definition, by an obvious induction on n. ∎

Definition 5.6 *For a path P in a labeled Markov chain $\mathcal{M}$, and LTL-BDDs ϕ and ψ,* `step()` ***reduces ϕ to ψ along*** *P if $\exists n \geq 0$ such that $\texttt{step}^n(\phi,\, P) = \psi$.*

Intuitively, `step()` reduces ϕ to ψ along P if the sequence of recursive calls made by $\texttt{measure}(\phi,\, s)$ as it traverses P eventually includes a call to $\texttt{measure}(\psi,\, t)$ (for some t).

Note that $n \geq 0$ allows $n = 0$, so by this definition, ϕ reduces to itself along any path.

Definition 5.7 *For two LTL-BDDs ϕ and ψ, ϕ is* ***reducible to*** *ψ (or equivalently, ψ is reducible from ϕ) if either:*

1. *$\exists$ some path P such that* `step()` *reduces ϕ to ψ along P, or*
2. *ϕ contains an until $\tau \mathcal{U} v$, and $\exists$ a path P such that* `step()` *reduces v to ψ along P.*

Note that, unlike Definition 5.6, Definition 5.7 is independent of any specific path or Markov chain. This reducibility relation will be useful to us because of Corollary 5.16 and Proposition 5.31: any formula ϕ is reducible to finitely many other formulas ψ, and for any recursive call $\texttt{measure}(\psi, t)$ resulting from $\texttt{measure}(\phi, s)$, ϕ is reducible to ψ.

Definition 5.8 *Two LTL-BDDs ϕ and ψ are* ***mutually reducible*** *if ϕ is reducible to ψ and ψ is reducible to ϕ.*

Definition 5.9 *A **possible path** P in a labeled Markov chain $\mathcal{M}$ is a (finite or infinite) sequence of states $(p_1, p_2, \ldots)$ describing a possible sequence of state transitions in $\mathcal{M}$: i.e., such that $\forall i \geq 1$, $\delta(p_i, p_{i+1}) > 0$.*

For two states s and t, then, t is reachable from s iff there exists a possible path $(p_1 = s, p_2, \ldots, p_n = t)$.

Definition 5.10 *A state s **occurs infinitely often** in a path $P = (p_1, p_2, \ldots)$ if there are infinitely many i such that $p_i = s$. Similarly, a finite path $Q = (q_1, q_2, \ldots, q_m)$ occurs infinitely often in P if there are infinitely many i such that, $\forall 1 \leq j \leq m$, $p_{i-1+j} = q_j$.*

Definition 5.11 *A **plausible path** in a labeled Markov chain $\mathcal{M}$ is an infinite possible path $P = (p_1, p_2, \ldots)$ such that for any state s occurring infinitely often in P, if $Q = (q_1 = s, q_2, q_3, \ldots, q_m)$ is a finite possible path starting from s, then Q occurs infinitely often in P.*

The idea of a plausible path is crucial to our algorithm. It formalizes the intuition once expressed by my mother: How is it you call so often and I'm not home, if you never call when I'm home? In other words, given enough chances to happen, it is implausible that a possible event should never happen.

An example of an implausible path in $\mathcal{M}_1$ from page 10 is $R = (s_1, s_1, s_1, \ldots)$, which never visits s_2 or s_3 despite infinitely many opportunities. Section 3.5.3 illustrates how `measure()` ignores these implausible paths.

Definition 5.12 *A finite possible path $Q = (q_1, q_2, \ldots, q_m)$ **determines** an LTL formula ϕ in a labeled Markov chain $\mathcal{M}$ if either every plausible path $P = (p_1 = q_1, \ldots, p_m = q_m, p_{m+1}, \ldots)$ beginning with Q satisfies ϕ, or no such P satisfies ϕ.*

In other words, the course $\mathcal{M}$ follows beyond $p_m = q_m$ doesn't matter: the first m states of P determine whether or not it satisfies ϕ.

The following examples are worth tracing through for a full understanding of plausible and determining paths:

1. Note that $\mathcal{M}$ is an essential parameter in Definition 5.12. For example, referring again to $\mathcal{M}_1$ from Figure 1.1, it is obvious that the prefix $Q' = (s_1, s_1)$

determines a (as true) and $X\neg b$ (as false); but not so obvious that it also determines $\psi = \mathcal{TU}(\neg b \wedge X\neg b)$ ("Eventually b is false in successive states"), as false, since no possible path in $\mathcal{M}_1$ starting with Q' satisfies this formula. In fact, the empty path () determines ψ in $\mathcal{M}_1$.

2. The stipulation that P be plausible is also essential to the definition. For example, Q' also determines $\omega = \mathcal{TU}\neg a$ ("Eventually a is false") in $\mathcal{M}_1$, as true, despite the existence of paths such as $R = (s_1, s_1, s_1, \ldots)$ which don't satisfy ω. Such paths are possible, but not plausible; any plausible path in $\mathcal{M}_1$ eventually enters s_3, where a is false, satisfying ω.

3. Theorem 5.41 asserts that, for any plausible path P and formula ϕ, P has a finite prefix determining ϕ. But note that this theorem does not hold in general for possible paths: there exist possible paths P such that no finite prefix of P determines certain formulas ϕ, but every such P is implausible. No such P exists in $\mathcal{M}_1$, but the 6-state Markov chain $\mathcal{M}$ in the sample input to MCMC (Chapter 6) contains examples, such as $S = (s_5, s_5, s_5, \ldots)$. No finite prefix of S determines $\omega = \mathcal{TU}b$.

Definition 5.13 `step()` ***resolves*** ϕ ***along*** P *if it reduces* ϕ *to* ω *along* P *(Definition 5.6), where* $\omega = \mathcal{T}/\mathcal{F} =$ *whether* $P \models \phi$.

Definition 5.14 *An expression e, made up of a scalar term* k_0 *and n variable terms* $k_1 x_{\phi_1 t_1} \ldots k_n x_{\phi_n t_n}$, ***represents*** *a number k if replacing the variables with the probabilities they stand for leads to a sum of k:* $k_0 + \sum_{i=1}^{n} k_i \cdot \texttt{prob}(t_i \models \phi_i) = k$.

5.2 `measure()` and `step()` terminate

Proposition 5.15 *For any LTL-BDD* ϕ *and state s in a Markov chain* $\mathcal{M}$, *the return values of* `measure`(ϕ, s) *and* `step`(ϕ, s) *contain no LTL-BDD atoms not already present in* ϕ.

Proof In the LTL-BDD representation, there is a distinct LTL-BDD atom for each distinct LTL atom, next, or until. Examining the definitions of `measure()` and `step()` (pages 31 and 33), we find that they create no new LTL atoms, nexts or untils not already present in ϕ. ∎

Corollary 5.16 *For any LTL-BDD ϕ, if $|A|$ is the number of distinct LTL-BDD atoms occurring in ϕ, then there exist no more than $2^{2^{|A|}}$ distinct ψ such that ϕ is reducible to ψ.*

Proof Follows from Propositions 5.15 and A.2. ∎

Corollary 5.17 *For any LTL-BDD ϕ and state s in a Markov chain $\mathcal{M}$, if $|A|$ is the number of distinct LTL-BDD atoms occurring in ϕ, then a top-level call to* `measure`*(ϕ, s) generates no more than $2^{2^{|A|}}$ LTL-BDDs.*

Proof Again, follows from Propositions 5.15 and A.2. ∎

Proposition 5.18 `measure`*(ϕ, s) and* `step`*(ϕ, s) terminate.*

Proof Structural induction. Assume that, for all subformulas ψ of ϕ, and for all t, `measure`(ψ, t) and `step`(ψ, t) terminate. We need to show that therefore `measure`(ϕ, s) and `step`(ϕ, s) terminate.

That `step`(ϕ, s) terminates follows immediately by the induction hypothesis: its recursive calls to `measure`() and `step`() all pass in subformulas as arguments.

Next, `measure`(ϕ, s). `measure`() has three cases, of which the first two ($\mathcal{T}/\mathcal{F}$, or a solution in the cache) terminate immediately. There remains the third, nontrivial case. In this case, `measure`(ϕ, s) makes a recursive call to `step`(ϕ, s) and multiple recursive calls to `measure`(ϕ', s'). The call to `step`(ϕ, s) terminates by the reasoning above. But we cannot infer termination of the recursive calls to `measure`(ϕ', s') from the induction hypothesis, since ϕ' is not necessarily a subformula of ϕ.

However, note that the first thing a nontrivial call to `measure`() does is store a new variable in the cache; new because the nontrivial case is only entered if nothing was previously stored in the cache for (ϕ, s). The number of states is finite, and by Corollary 5.17, so is the possible number of distinct LTL-BDDs. Therefore the number of possible input pairs (ψ, t) is bounded. So, since each nontrivial call to `measure`() adds a new input pair's variable to the cache, and cached solutions are global and never removed, it follows that only finitely many of the recursive calls to `measure`() are nontrivial.

Consider the last of these nontrivial calls. Since it is the last, all its recursive calls must be to the trivial cases, which terminate. Therefore it also terminates. And we

can similarly reason that so must the 2nd-last nontrivial call, the 3rd-last, and so on. So all the nontrivial calls resulting from a call to `measure`(ϕ, s) terminate, as well as the trivial calls. Therefore, `measure`(ϕ, s) terminates.

So by induction, every call to `measure`() or `step`() terminates. ∎

5.3 step() distributes over BDD operations

Lemma 5.19 `step`() *commutes with* `not`() *and distributes over* `and`(), `or`(), *and* `cond`()*:*

- `step(not(`ϕ`),` s`) = not(step(`ϕ`,` s`))`
- `step(and(`ϕ_1`,` ϕ_2`),` s`) = and(step(`ϕ_1`,` s`), step(`ϕ_2`,` s`))`
- `step(or(`ϕ_1`,` ϕ_2`),` s`) = or(step(`ϕ_1`,` s`), step(`ϕ_2`,` s`))`
- `step(cond(`ϕ_1`,` ϕ_2`,` ϕ_3`),` s`) = cond(step(`ϕ_1`,` s`), step(`ϕ_2`,` s`), step(`ϕ_3`,` s`))`

Proof We show these by structural induction, using some of the BDD operation identities from Figure A.15 (page 116) and straightforward (if somewhat laborious) case analysis.

1. `not`(). Assume the commutativity identity holds for all subformulas of o (as defined in Definition 5.1). We show that therefore it also holds for ϕ, whether ϕ is a boolean or an atom LTL-BDD.

Suppose $\phi = \mathcal{T}$ ($\phi = \mathcal{F}$ is exactly parallel):

	`step(not(`$\mathcal{T}$`),` s`)`	
=	`step(`$\mathcal{F}$`,` s`)`	(defn of `not`())
=	$\mathcal{F}$	(defn of `step`())
=	`not(`$\mathcal{T}$`)`	(defn of `not`() again)
=	`not(step(`$\mathcal{T}$`,` s`))`	(defn of `step`() again)

Or suppose $\phi = (\alpha\ ?\ \psi : \omega)$ (covering the $\alpha = a$, $\alpha = X\tau$, and $\alpha = \tau \mathcal{U} \upsilon$ cases all together):

$$
\begin{aligned}
& \texttt{step}(\texttt{not}(\alpha\ ?\ \psi : \omega),\ s) \\
= \ & \texttt{step}((\alpha\ ?\ \texttt{not}(\psi) : \texttt{not}(\omega)),\ s) && \text{(defn of not())} \\
= \ & \texttt{cond}(\alpha',\ \texttt{step}(\texttt{not}(\psi),\ s),\ \texttt{step}(\texttt{not}(\omega),\ s)) && \text{(defn of step())} \\
= \ & \texttt{cond}(\alpha',\ \texttt{not}(\texttt{step}(\psi,\ s)),\ \texttt{not}(\texttt{step}(\omega,\ s))) && \text{(ind hyp)} \\
= \ & \texttt{not}(\texttt{cond}(\alpha',\ \texttt{step}(\psi,\ s),\ \texttt{step}(\omega,\ s))) && \text{(Figure A. 5)} \\
= \ & \texttt{not}(\texttt{step}((\alpha\ ?\ \psi : \omega),\ s)) && \text{(defn of step())}
\end{aligned}
$$

In either case the induction holds. So step() commutes with not().

2. $\texttt{and}(\phi_1,\ \phi_2)$. Structural induction again. This time assume the claim holds for any and() operation where both operands are subformulas (of ϕ_1 or ϕ_2), or one operand is a subformula and the other is ϕ_1 or ϕ_2.

Suppose one of the operands, say ϕ_1, is $\mathcal{F}$:

$$
\begin{aligned}
& \texttt{step}(\texttt{and}(\mathcal{F},\ \phi_2),\ s) \\
= \ & \texttt{step}(\mathcal{F},\ s) && \text{(defn of and())} \\
= \ & \mathcal{F} && \text{(defn of step())} \\
= \ & \texttt{and}(\mathcal{F},\ \texttt{step}(\phi_2,\ s)) && \text{(defn of and())} \\
= \ & \texttt{and}(\texttt{step}(\mathcal{F},\ s),\ \texttt{step}(\phi_2,\ s)) && \text{(defn of step())}
\end{aligned}
$$

Or suppose $\phi_1 = \mathcal{T}$:

$$
\begin{aligned}
& \texttt{step}(\texttt{and}(\mathcal{T},\ \phi_2),\ s) \\
= \ & \texttt{step}(\phi_2,\ s) && \text{(defn of and())} \\
= \ & \texttt{and}(\mathcal{T},\ \texttt{step}(\phi_2,\ s)) && \text{(defn of and())} \\
= \ & \texttt{and}(\texttt{step}(\mathcal{T},\ s),\ \texttt{step}(\phi_2,\ s)) && \text{(defn of step())}
\end{aligned}
$$

If neither ϕ_1 nor ϕ_2 is a boolean, then both are atom LTL-BDDs: $\phi_1 = (\alpha_1\ ?\ \psi_1 : \omega_1)$, $\phi_2 = (\alpha_2\ ?\ \psi_2 : \omega_2)$. Now there are four cases, based on whether $\alpha_1 = \alpha_2$, and whether and() enters the $(n_{a_1} = n_{\neg a_1})$ case (see the definition on page 107):

Case 2a: $\alpha_1 \neq \alpha_2$ (say, $\alpha_1 < \alpha_2$), $\texttt{and}(\psi_1,\ \phi_2) \neq \texttt{and}(\omega_1,\ \phi_2)$:

$$
\begin{aligned}
& \texttt{step}(\texttt{and}((\alpha_1\ ?\ \psi_1 : \omega_1),\ (\alpha_2\ ?\ \psi_2 : \omega_2)),\ s) \\
= \ & \texttt{step}((\alpha_1\ ?\ \texttt{and}(\psi_1,\ \phi_2) : \texttt{and}(\omega_1,\ \phi_2)),\ s) && \text{(defn of and())} \\
= \ & \texttt{cond}(\alpha_1',\ \texttt{step}(\texttt{and}(\psi_1,\ \phi_2),\ s),\ \texttt{step}(\texttt{and}(\omega_1,\ \phi_2),\ s)) && \text{(defn of step())} \\
= \ & \texttt{cond}(\alpha_1',\ \texttt{and}(\texttt{step}(\psi_1,\ s),\ \texttt{step}(\phi_2,\ s)), \\
& \qquad \texttt{and}(\texttt{step}(\omega_1,\ s),\ \texttt{step}(\phi_2,\ s))) && \text{(ind hyp)} \\
= \ & \texttt{and}(\texttt{cond}(\alpha_1',\ \texttt{step}(\psi_1,\ s),\ \texttt{step}(\omega_1,\ s)),\ \texttt{step}(\phi_2,\ s)) && \text{(Figure A.15)} \\
= \ & \texttt{and}(\texttt{step}((\alpha_1\ ?\ \psi_1 : \omega_1),\ s),\ \texttt{step}(\phi_2,\ s)) && \text{(defn of step())}
\end{aligned}
$$

Case 2b: $\alpha_1 < \alpha_2$, $\texttt{and}(\psi_1,\ \phi_2) = \texttt{and}(\omega_1,\ \phi_2)$:

$\texttt{step}(\texttt{and}((\alpha_1\ ?\ \psi_1 : \omega_1), (\alpha_2\ ?\ \psi_2 : \omega_2)), s)$

$= \texttt{step}(\texttt{and}(\psi_1, \phi_2), s)$ (defn of and())

$= \texttt{cond}(\alpha_1', \texttt{step}(\texttt{and}(\psi_1, \phi_2), s), \texttt{step}(\texttt{and}(\psi_1, \phi_2), s))$ (defn of cond())

$= \texttt{cond}(\alpha_1', \texttt{step}(\texttt{and}(\psi_1, \phi_2), s), \texttt{step}(\texttt{and}(\omega_1, \phi_2), s))$ (case 2b)

(...as in case 2a)

Case 2c: $\alpha_1 = \alpha_2 = \alpha$, $\texttt{and}(\psi_1, \psi_2) \neq \texttt{and}(\omega_1, \omega_2)$:

$\texttt{step}(\texttt{and}((\alpha\ ?\ \psi_1 : \omega_1), (\alpha\ ?\ \psi_2 : \omega_2)), s)$

$= \texttt{step}((\alpha\ ?\ \texttt{and}(\psi_1, \psi_2) : \texttt{and}(\omega_1, \omega_2)), s)$ (defn of and())

$= \texttt{cond}(\alpha', \texttt{step}(\texttt{and}(\psi_1, \psi_2), s), \texttt{step}(\texttt{and}(\omega_1, \omega_2), s))$ (defn of step())

$= \texttt{cond}(\alpha', \texttt{and}(\texttt{step}(\psi_1, s), \texttt{step}(\psi_2, s)),$
$\texttt{and}(\texttt{step}(\omega_1, s), \texttt{step}(\omega_2, s)))$ (ind hyp)

$= \texttt{and}(\texttt{cond}(\alpha', \texttt{step}(\psi_1, s), \texttt{step}(\omega_1, s)),$
$\texttt{cond}(\alpha', \texttt{step}(\psi_2, s), \texttt{step}(\omega_2, s)))$ (Figure A.15)

$= \texttt{and}(\texttt{step}((\alpha\ ?\ \psi_1 : \omega_1), s), \texttt{step}((\alpha\ ?\ \psi_2 : \omega_2), s))$ (defn of step())

Case 2d: $\alpha_1 = \alpha_2 = \alpha$, $\texttt{and}(\psi_1, \psi_2) = \texttt{and}(\omega_1, \omega_2)$:

$\texttt{step}(\texttt{and}((\alpha\ ?\ \psi_1 : \omega_1), (\alpha\ ?\ \psi_2 : \omega_2)), s)$

$= \texttt{step}(\texttt{and}(\psi_1, \psi_2), s)$ (defn of and())

$= \texttt{cond}(\alpha', \texttt{step}(\texttt{and}(\psi_1, \psi_2), s), \texttt{step}(\texttt{and}(\psi_1, \psi_2), s))$ (defn of cond())

$= \texttt{cond}(\alpha', \texttt{step}(\texttt{and}(\psi_1, \psi_2), s), \texttt{step}(\texttt{and}(\omega_1, \omega_2), s))$ (case 2d)

(...as in case 2c)

In every case the induction holds, so step() distributes over and().

And **3.** or() and **4.** cond() follow immediately, since or() is defined in terms of and() and not() (Definition A.6), and cond() in terms of or(), and() and not() (Definition A.7). ∎

Corollary 5.20 $\texttt{step}^n()$ *commutes with* not() *and distributes over* and(), or(), *and* cond()*:*

- $\texttt{step}^n(\texttt{not}(\phi), P) = \texttt{not}(\texttt{step}^n(\phi, P))$
- $\texttt{step}^n(\texttt{and}(\phi_1, \phi_2), P) = \texttt{and}(\texttt{step}^n(\phi_1, P), \texttt{step}^n(\phi_2, P))$
- $\texttt{step}^n(\texttt{or}(\phi_1, \phi_2), P) = \texttt{or}(\texttt{step}^n(\phi_1, P), \texttt{step}^n(\phi_2, P))$
- $\texttt{step}^n(\texttt{cond}(\phi_1, \phi_2, \phi_3), P) = \texttt{cond}(\texttt{step}^n(\phi_1, P), \texttt{step}^n(\phi_2, P), \texttt{step}^n(\phi_3, P))$

Proof Follows from Lemma 5.19 by simple induction on n. Example (for the non-trivial $n > 0$ case):

$$\begin{aligned}
& \mathtt{step}^n(\mathtt{and}(\phi_1, \phi_2), P) \\
= \ & \mathtt{step}^{n-1}(\mathtt{step}(\mathtt{and}(\phi_1, \phi_2), p_1), P_2) && \text{(defn of } \mathtt{step}^n()) \\
= \ & \mathtt{step}^{n-1}(\mathtt{and}(\mathtt{step}(\phi_1, p_1), \mathtt{step}(\phi_2, p_1)), P_2) && \text{(Lemma 5.19)} \\
= \ & \mathtt{and}(\mathtt{step}^{n-1}(\mathtt{step}(\phi_1, p_1), P_2), \mathtt{step}^{n-1}(\mathtt{step}(\phi_2, p_1), P_2)) && \text{(ind hyp)} \\
= \ & \mathtt{and}(\mathtt{step}^n(\phi_1, P), \mathtt{step}^n(\phi_2, P)) && \text{(defn of } \mathtt{step}^n())
\end{aligned}$$

And the others operators follow similarly. ■

5.4 Mutually reducible LTL-BDDs contain the same atoms

Proposition 5.21

1. *The reducible-to and reducible-from relations on formulas (Definition 5.7) are preorders.*

2. *The mutually-reducible relation on formulas (Definition 5.8) is an equivalence relation.*

Proof In domain theory, a preorder is defined as a relation that is reflexive and transitive. Consider the reducible-to relation first. Reflexivity follows from Definition 5.6: ϕ reduces to itself along any path P, so ϕ is reducible to (and from) itself. Transitivity is also clear, from Definition 5.7: if ϕ is reducible to ψ, and ψ is reducible to ω, then ϕ must be reducible to ω. And the same holds for reducible-from.

An equivalence relation is a preorder which is also symmetric. The reflexivity and transitivity of mutually-reducible follow straightforwardly from the fact that these properties hold for reducible-to and reducible-from, and the symmetry of mutually-reducible is obvious. ■

We can now identify some constraints on which pairs of LTL-BDDs satisfy these relations:

Proposition 5.22 *If one LTL-BDD, ϕ, is reducible to another, ψ, then every LTL-BDD atom α occurring in ψ (of one of the three forms a, $X\tau$, or $\tau \mathcal{U} \upsilon$) also occurs in ϕ.*

Proof By Proposition 5.15, $\texttt{step}(\phi, s)$ introduces no new LTL-BDD atoms α not already present in ϕ. So any α occurring in a ψ of the form $\texttt{step}^n(\phi, P)$ also occurs in ϕ. Similarly, for any $\tau \mathcal{U} \upsilon$ occurring in ϕ, any α occurring in $\texttt{step}^n(\upsilon, P)$ occurs in υ, and therefore of course also in ϕ. Therefore, by Definition 5.7, any α occurring in ψ also occurs in ϕ. ∎

Corollary 5.23 *If one LTL-BDD, ϕ, is reducible to another, ψ, then* $\texttt{udepth}(\phi) \geq \texttt{udepth}(\psi)$.

Proof Follows from Proposition 5.22: since ϕ contains every LTL-BDD atom in ψ, it must contain every until in ψ, including those of greatest depth. ∎

Corollary 5.24 *Any two mutually reducible LTL-BDDs ϕ and ψ each contain exactly the same set of LTL-BDD atoms, and* $\texttt{udepth}(\phi) = \texttt{udepth}(\psi)$.

Proof Follows immediately from Definition 5.8, Proposition 5.22, and Corollary 5.23. ∎

We can also prove some stronger constraints on mutual reducibility:

Proposition 5.25 *For any until-free LTL-BDD ϕ, and any state s in a Markov chain $\mathcal{M}$,* $\texttt{xdepth}(\texttt{step}(\phi, s)) \leq \max(\texttt{xdepth}(\phi) - 1, 0)$.

Proof Straightforward structural induction on ϕ. Assume the claim holds for ϕ's subformulas. If ϕ is $\mathcal{T}$ or $\mathcal{F}$, $\texttt{xdepth}(\texttt{step}(\phi, s)) = \texttt{xdepth}(\phi) = 0$ and the claim is obviously true. Otherwise, ϕ must be of the form $(\alpha \ ? \ \psi : \omega)$, where α is either an LTL atom a or a next $X\tau$.

Case 1: $\alpha = a$. Then by the definition of `step()` (page 33), $\texttt{step}(\phi, s)$ is either $\texttt{step}(\psi, s)$ or $\texttt{step}(\omega, s)$. Both cases are equivalent, so suppose $\texttt{step}(\phi, s) = \texttt{step}(\psi, s)$:

$$
\begin{aligned}
& \texttt{xdepth}(\texttt{step}((a \ ? \ \psi : \omega), s)) \\
= \ & \texttt{xdepth}(\texttt{step}(\psi, s)) \\
\leq \ & \max(\texttt{xdepth}(\psi) - 1, 0) && \text{(ind hyp)} \\
\leq \ & \max(\texttt{xdepth}(\phi) - 1, 0) && \text{(defn of xdepth())}
\end{aligned}
$$

Case 2: $\alpha = X\tau$:

$$
\begin{aligned}
& \texttt{xdepth}(\texttt{step}((X\tau\ ?\ \psi : \omega),\ s)) \\
= \ & \texttt{xdepth}(\texttt{cond}(\tau,\ \texttt{step}(\psi,\ s),\ \texttt{step}(\omega,\ s))) && \text{(defn of step())} \\
\leq \ & \max(\texttt{xdepth}(\tau),\ \texttt{xdepth}(\texttt{step}(\psi,\ s)),\ \texttt{xdepth}(\texttt{step}(\omega,\ s))) && \text{(defn of xdepth())} \\
\leq \ & \max(\texttt{xdepth}(\tau),\ \max(\texttt{xdepth}(\psi) - 1,\ 0), \\
& \quad \max(\texttt{xdepth}(\omega) - 1,\ 0)) && \text{(ind hyp)} \\
= \ & \max(\max(\texttt{xdepth}(\tau),\ \texttt{xdepth}(\psi) - 1,\ \texttt{xdepth}(\omega) - 1),\ 0) \\
= \ & \max(\max(\texttt{xdepth}(\tau) + 1,\ \texttt{xdepth}(\psi),\ \texttt{xdepth}(\omega)) - 1,\ 0) \\
= \ & \max(\texttt{xdepth}(\phi) - 1,\ 0) && \text{(defn of xdepth())}
\end{aligned}
$$

So in either case the claim holds, completing the induction. ∎

Proposition 5.26 *Any two distinct mutually reducible LTL-BDDs both contain at least one until.*

Proof It is easy to show that no two distinct until-free LTL-BDDs $\phi \neq \psi$ are mutually reducible; the claim then follows immediately.

Suppose without loss of generality that $\texttt{xdepth}(\psi) \leq \texttt{xdepth}(\phi)$. If ϕ and ψ are $\mathcal{T}$ and $\mathcal{F}$ they are obviously not mutually reducible. Otherwise, at least one of them must contain an LTL atom or next, so $\texttt{xdepth}(\phi) \geq 1$. We will show that ψ is not reducible to ϕ. Since ψ contains no untils, we only need to show that, for any path $P = (p_1, p_2, \ldots)$ and $n \geq 0$, $\texttt{step}^n(\phi,\ P) \neq \phi$.

$\texttt{step}^0(\psi,\ P) = \psi \neq \phi$, so assume $n \geq 1$. $\texttt{step}^1(\psi,\ P) = \texttt{step}(\psi,\ p_1)$. By Proposition 5.25, $\texttt{xdepth}(\texttt{step}(\phi,\ p_1))$ is either 0 or $\texttt{xdepth}(\psi) - 1$. In either case, this depth is $< \texttt{xdepth}(\phi)$, so $\texttt{step}^1(\psi,\ P) \neq \phi$. And, since the depth of nested nexts can only decrease with further applications of step(), we have our result that $\forall n \geq 0$, $\texttt{step}^n(\phi,\ P) \neq \phi$. Therefore, ψ is not reducible to ϕ.

Therefore, if any distinct pair ϕ and ψ are mutually reducible, at least one of them contains an until, and therefore by Corollary 5.24 they both do. ∎

Lemma 5.27 *For any two distinct mutually reducible LTL-BDDs $\phi \neq \psi$, every LTL atom a or next $X\tau$ occurring in ϕ also occurs inside an until in ψ.*

Proof Follows from an extension of the reasoning used to prove Proposition 5.26. Let $A_{\phi\psi}$ be the set of LTL-BDD atoms occurring in ϕ but not inside any until in ψ, or in ψ but not inside any until in ϕ. Suppose $A_{\phi\psi}$ is nonempty. Then let α be the element (or one of the elements) of this set with the greatest number of nested nexts.

i.e.. the greatest `xdepth()`. Suppose without loss of generality that α occurs in ϕ, but not inside any until in ψ.

Using α, we will show that ψ is not reducible to ϕ. It will follow that if ϕ and ψ are mutually reducible, then $A_{\phi\psi}$ is empty, and the claim holds.

First we dispense with simple cases. If any until occurs in one of ϕ and ψ but not the other, then Corollary 5.24 is enough to show that ϕ and ψ are not mutually reducible. So we need only consider the case where both contain the same untils.

Also, the second clause of the definition of reducibility (Definition 5.7) is satisfied only if ψ contains some until $\tau \mathcal{U} v$ such that v is reducible to ϕ. But by hypothesis, no until in ψ contains α. So by Proposition 5.22, no such v is reducible to ϕ.

So it remains only to prove that the first clause of the definition is not satisfied: $\forall$ paths P and $n \geq 0$, $\texttt{step}^n(\psi, P) \neq \phi$. The $n = 0$ case is trivial: $\texttt{step}^0(\psi, P) = \psi$, and by hypothesis $\psi \neq \phi$. So assume $n \geq 1$.

Suppose $\texttt{step}^n(\psi, P) = \phi$. Since ϕ contains α, so does $\texttt{step}^n(\psi, P)$, and by Proposition 5.22, therefore so does ψ. Furthermore, since no until in ψ contains α, we can see from the definition of `step()` (page 33) that the α in $\texttt{step}^n(\psi, P)$ can only have been obtained by applying `step()` n times to an LTL-BDD α_{-n} in ψ in which α was enclosed by n nexts. That is, $\texttt{xdepth}(\alpha_{-n}) \geq \texttt{xdepth}(\alpha) + n > \texttt{xdepth}(\alpha)$.

Now, since no until in ψ contains α, and we are assuming ϕ and ψ contain the same untils, it follows that no until in ϕ contains α. So, since α_{-n} contains α, no until in ϕ can contain α_{-n}. Therefore, α_{-n} belongs to $A_{\phi\psi}$. That is, we have another element of $A_{\phi\psi}$ with more deeply nested untils than α — contradicting the definition of α.

Therefore our assumption, that for some P and $n > 0$ $\texttt{step}^n(\psi, P) = \phi$, was false. So if $\phi \neq \psi$ and $A_{\phi\psi}$ is nonempty, then ϕ and ψ are not mutually reducible. The claim follows immediately. ∎

Corollary 5.28 *If an LTL-BDD ϕ is mutually reducible with any other LTL-BDD ψ ($\phi \neq \psi$), then every LTL atom a or next $X\tau$ occurring in ϕ also occurs inside an until in ϕ.*

Proof Follows from Lemma 5.27 and Corollary 5.24: any a or $X\tau$ in ϕ must occur in an until in ψ, and the same until must also occur in *phi*. ∎

Remark 5.29 *The above constraints on reducibility and mutual reducibility are necessary but not sufficient: there exist LTL-BDDs which satisfy these constraints and yet are not (mutually) reducible.*

For example, although they are made up of the same LTL-BDD atoms and satisfy the constraint of Lemma 5.27, the two formulas $\mathcal{T}\mathcal{U}a$ and $a \wedge (\mathcal{T}\mathcal{U}a)$ are not mutually reducible: $\mathcal{T}\mathcal{U}a$ is not reducible to $a \wedge (\mathcal{T}\mathcal{U}a)$. However, for many pairs (ϕ, ψ) these constraints are enough to show that ϕ is not reducible to ψ, or that ϕ and ψ are not mutually reducible.

5.5 For every variable $x_{\psi t}$ in return value $r_{\phi s}$, ϕ and ψ are mutually reducible

Proposition 5.30 *Any call to* `measure()` *eliminates any variables it creates before returning.*

Proof `measure()` returns an expression consisting of some variable terms and a scalar (numerical) part. Variables are only created in one place in the algorithm: the first line of `measure()`'s nontrivial case (page 31). But this case then eliminates the created variable from all cached expressions a few lines below. Since all recursive calls `measure()` makes complete before it returns, eliminating any variables they created, the set of variables in existence at the beginning of any call to `measure()` is the same as the set in existence when that same call returns. ∎

Proposition 5.31 *If a call to* `measure`(ϕ, s) *or* `step`(ϕ, s) *results in a (possibly nested) recursive call to* `measure`(ψ, t)*, then ϕ is reducible to ψ.*

Proof Follows from the definitions of `measure()` and `step()` (pages 31 and 33). There are only two types of recursive `measure()` calls resulting from `measure`(ϕ, s): those made by `measure()`, and those made by `step()`. The first type occurs when some recursive call `measure`(ψ, t) calls `measure`(ψ', t'), where $\psi' =$ `step`(ψ, t). The second type is of the form `measure`(υ, t), where $\tau \mathcal{U} \upsilon$ is an until occurring in ϕ. So any sequence of these can only lead to recursive calls of the form `measure`(ψ, t), where ψ is either `step`$^n(\phi, P)$ for some path P and $n \geq 0$, or `step`$^n(\upsilon, P)$. In other words, by Definition 5.7, ϕ is reducible to ψ. ∎

Lemma 5.32 *If $r_{\phi s}$ is the expression returned by* `measure`(ϕ, s)*, or the solution cached for (ϕ, s), then for every variable $x_{\psi t}$ occurring (with coefficient > 0) in $r_{\phi s}$, ϕ and ψ are mutually reducible.*

Proof Induction on call return order. Suppose the claim holds for all expressions returned by calls to `measure`() terminating before `measure`(ϕ, s), including recursive calls. We will show that it therefore holds for this call as well.

Consider the three cases in `measure`():

Case 1: $\phi = \mathcal{T}/\mathcal{F}$. So $r_{\phi s} = 1$ or 0 and the claim is trivially true.

Case 2: $r_{\phi s}$ retrieved from the cache. Then a previous call to `measure`(ϕ, s) must have cached $x_{\phi s}$. If this previous call hasn't yet completed, then it also hasn't yet substituted for $x_{\phi s}$, so the cached expression is still just $x_{\phi s}$. Therefore $r_{\phi s} = x_{\phi s}$, and again the claim is trivially satisfied (since mutual reducibility is reflexive).

If the previous call did complete, then the expression $r'_{\phi s}$ it left in the cache was the same expression as it returned, and by the induction hypothesis $r'_{\phi s}$ satisfied the claim. Since then, $r'_{\phi s}$ may have been modified by variable substitutions, performed by calls of the form `substitute`$(x_{\psi t} := r_{\psi t})$. But again, any such substitution occurred just before `measure`(ψ, t) returned $r_{\psi t}$, so by the induction hypothesis, any variable $x_{\omega u}$ in $r_{\psi t}$ was mutually reducible with ψ. And since ψ occurred in $r'_{\phi s}$, and $r'_{\phi s}$ satisfied the claim, ψ and ϕ are mutually reducible. So by the transitivity of mutual reducibility (Proposition 5.21), for any variable $x_{\omega u}$ introduced by substitution into $r'_{\phi s}$, ϕ and ω are mutually reducible. Therefore the claim still holds for $r_{\phi s}$.

Case 3: no cached solution. Then, apart from $x_{\phi s}$, $r_{\phi s}$ contains the same variables as e, where e is the expression computed as a weighted sum of recursive calls to `measure`(ϕ', s'). By the induction hypothesis, for any variable $x_{\psi t}$ in one of the recursively computed expressions, ϕ' and *psi* are mutually reducible. And since $\phi' =$ `step`(ϕ, s), ϕ is reducible to ϕ', and by transitivity therefore also to ψ. So it remains only to show that ψ is reducible to ϕ.

By Proposition 5.30, all variables created during the call to `measure`(ϕ, t) are eliminated before it returns. So $x_{\psi t}$ must have been created, but not eliminated, before the call began. Therefore the `measure`(ψ, t) call which created $x_{\psi t}$ began before and ended after the call to `measure`(ϕ, s). The only such calls are recursive calls waiting for `measure`(ϕ, s) to terminate. That is, `measure`(ϕ, s) is a recursive call

resulting from measure(ψ, t). So by Proposition 5.31, ψ is reducible to ϕ. Therefore, for any variable $x_{\psi t}$ occurring in $r_{\phi s}$, ϕ and ψ are mutually reducible. ∎

5.6 measure() returns a number

Corollary 5.33 *A top-level call to* measure() *returns a number.*

Proof Of course, no variables exist at the beginning of the top-level call. So by Proposition 5.30, none will exist when it returns, and therefore its return value must be variable-free, i.e., a number. ∎

Lemma 5.34 *If ϕ and $\phi' =$ step(ϕ, s) are not mutually reducible, then every call from* measure(ϕ, s) *to* measure(ϕ', s') *returns a number.*

Proof Follows from Lemma 5.32, and the reasoning used to prove it. We will show that if the expression $r_{\phi' s'}$ returned by measure(ϕ', s') contains any variable $x_{\psi t}$, then ϕ and ϕ' are mutually reducible. The claim follows immediately.

Suppose some such $x_{\psi t}$ exists. By Lemma 5.32, ϕ' and ψ are mutually reducible. And, following case 3 from our proof of Lemma 5.32, $x_{\psi t}$ must have been created, but not eliminated, before the measure(ϕ', s') call began. So either $\psi = \phi$, or the measure(ϕ, s) call resulted from a prior call to measure(ψ, t). In either case, ψ is reducible to ϕ; in the first case trivially, in the second case by Proposition 5.31.

But by the definition of ϕ', ϕ is immediately reducible to ϕ'. And by the mutual reducibility of ϕ' and ψ, and transitivity of reducible-to (Proposition 5.21), ϕ is reducible to ψ.

So if $r_{\phi' s'}$ contains a variable, ϕ and ψ are mutually reducible. Therefore, if they are not mutually reducible then $r_{\phi' s'}$ must be variable-free, i.e., a number. ∎

Corollary 5.35 *If ϕ and $\phi' =$ step(ϕ, s) are not mutually reducible, then* measure(ϕ, s) *returns a number.*

Proof ϕ cannot be $\mathcal{T}$ or $\mathcal{F}$, since in these cases $\phi' = \phi$ and therefore ϕ and ϕ' are mutually reducible. So the first call to measure(ϕ, s) returns the sum of some calls to measure(ϕ', s'). By Lemma 5.34, these calls all return numbers, and therefore so does measure(ϕ, s). ∎

Corollary 5.36 *If ϕ and $\phi' =$* `step`(ϕ, s) *are not mutually reducible, then the call from* `measure`(ϕ, s) *to* `substitute`$(x_{\phi s} := r_{\phi s})$ *performs exactly one substitution.*

Proof We can show that, at the time of the substitution, the only cached solution expression containing a term for $x_{\phi s}$ is the one-term $x_{\phi s}$ expression cached by `measure`(ϕ, s) itself.

First, consider expressions cached before the call to `measure`(ϕ, s). Any variable in these expressions must have been created before the call. Therefore, no substitution performed by `measure`(ϕ, s) or its recursive calls affects these expressions, because all such substitutions are for variables created after the `measure`(ϕ, s) call began.

The only other cached expressions at the time of the `substitute`$(x_{\phi s} := r_{\phi s})$ call are those cached by recursive `measure`(ψ, t) calls resulting from `measure`(ϕ, s). Any such call must have resulted either from one of the calls to `measure`(ϕ', s'), or from the call made by `measure`(ϕ, s) to `step`(ϕ, s). It is easy to show that, in either case, ψ is not reducible to ϕ. It will follow that ϕ and ψ are not mutually reducible, and therefore by Lemma 5.32 that $x_{\phi s}$ does not occur in $r_{\psi t}$.

Case 1: `measure`(ψ, t) results from `measure`(ϕ', s'). ϕ is immediately reducible to ϕ', and by Proposition 5.31, ϕ' is reducible to ψ. So, by transitivity of reducible-to (Proposition 5.21), if ψ was reducible to ϕ then ϕ and ϕ' would be mutually reducible, which is given to be false. Therefore ψ is not reducible to ϕ.

Case 2: `measure`(ψ, t) results from `step`(ϕ, s). Then it must have resulted from one of `step`()'s calls to `measure`(v, u), for some $\tau \mathcal{U} v$ occurring in ϕ. Therefore, by Proposition 5.31, v is reducible to ψ. But by Proposition 5.22, v is not reducible to ϕ, since ϕ contains $\tau \mathcal{U} v$ whereas v (being finite) does not. Therefore ψ cannot be reducible to ϕ.

So the only cached occurrence of $x_{\phi s}$ at the time of substitution, and therefore the only substitution, is in the expression cached by `measure`(ϕ, s). ∎

Lemma 5.37 *A recursive call to* `measure`() *from* `step`() *returns a number.*

Proof Parallels the proof of Lemma 5.34. First, recall that the only such recursive calls occur when an LTL-BDD ϕ contains an until $\tau \mathcal{U} v$, and a call to `step`(ϕ, s) makes calls of the form `measure`(v, u).

Suppose the expression r_{vu} returned by `measure`(v, u) contains a variable, $x_{\psi t}$. We will derive a contradiction, proving that r_{vu} contains no such variable.

By Lemma 5.32, v and ψ must be mutually reducible. LTL-BDDs are finite structures, so v cannot contain itself and therefore cannot contain the until $\tau \mathcal{U} v$. Therefore, by Corollary 5.24, ψ cannot contain $\tau \mathcal{U} v$.

Now, following case 3 from our proof of Lemma 5.32, $x_{\psi t}$ must have been created, but not eliminated, before the `measure`(v, u) call began. `measure`(v, u) was called by `step`(ϕ, s), so it follows that `step`(ϕ, s) resulted from an earlier, unfinished call to `measure`(ψ, t). Since ϕ contains $\tau \mathcal{U} v$, and neither `measure`() nor `step`() introduces any new LTL-BDD atoms (Proposition 5.15), ψ must also have contained $\tau \mathcal{U} v$ — contradicting our previous inference that ψ does not contain $\tau \mathcal{U} v$.

Therefore, our hypothesis that r_{vu} contained a variable $x_{\psi t}$ was false. So no $x_{\psi t}$ occurs in r_{vu}, and therefore r_{vu} is variable-free, i.e., a number. ∎

Lemma 5.38 *If* `measure`(ϕ, s) *returns an expression* $r_{\phi s} = k_0 + \sum_{i=1}^{n} k_i x_i$, *then:*

- *The coefficients are all between 0 and 1:* $\forall 0 \leq i \leq n,\ 0 \leq k_i \leq 1$.
- *So is their sum:* $0 \leq \sum_{i=0}^{n} k_i \leq 1$.

Proof Induction on return order. Assume both claim holds for all calls to `measure`() terminating before this one does. We want to deduce that they hold for $r_{\phi s}$.

Consider again the three cases in `measure`(ϕ, s):

Case 1: $\phi = \mathcal{T}/\mathcal{F}$. Returns 1 or 0, trivially satisfying both claims.

Case 2: $r_{\phi s}$ retrieved from the cache. The only place `cache`() is called caches an expression consisting of a single variable, $x_{\phi s}$ (coef sum: 1), satisfying both claims. This cached expression may have been modified by calls of the form `substitute`$(x_{\psi t} := r_{\psi t})$ before being retrieved as $r_{\phi s}$. But any such `substitute`() call was just prior to `measure`() returning $r_{\psi t}$, so by the induction hypothesis, $r_{\psi t}$ satisfied both claims. And it is easy to verify that therefore the substitution of $r_{\psi t}$ (coef sum: ≤ 1) for $x_{\psi t}$ (coef sum: 1) couldn't have violated either claim. Therefore, any $r_{\phi s}$ retrieved from the cache still satisfies both claims.

Case 3: no cached solution. Makes a number of recursive calls to `measure`(), computing an expression e as a weighted sum of the returned expressions, and then returning $r_{\phi s} =$ `solve`$(x_{\phi s} = e)$. We can show that e satisfies both claims, and that therefore so does $r_{\phi s}$.

By the induction hypothesis, both claims hold for each of the $r_{\phi' s'}$ expressions making up e. So, since every $r_{\phi' s'}$ has positive coefficients, so does e. And even in the

maximal case, where each $r_{\phi' s'}$ has a coefficient sum of 1, e's coefficients only sum to $\sum_{s'} \delta(s, s')$, which by the definition of a Markov chain (Chapter 2.2.5) is simply 1.

Then, by the definition of `solve()` (page 35), $r_{\phi s} = \frac{e - k x_{\phi s}}{1-k}$, where k is the coefficient of $x_{\phi s}$ in e. By hypothesis above, `measure()` does return successfully, so we can ignore the error case $k = 1$. Therefore, since e's coefficients are between 0 and 1, $0 \leq k < 1$. So the coefficients of $r_{\phi s}$ are just the coefficients of e divided by $1 - k$, a positive number, and therefore are positive as well. Furthermore, since the elimination of the $k x_{\phi s}$-term reduces e's coefficient sum by k, the sum of the coefficients in $r_{\phi s}$ is $\leq \frac{1-k}{1-k} = 1$. Therefore any $r_{\phi s}$ computed from e satisfies both claims.

So by induction, both claims always hold for $r_{\phi s}$. ∎

5.7 A plausible path has a prefix determining ϕ

The proofs in this section refer to Definitions 5.11 (plausible path P) and 5.12 (P determines ϕ).

Proposition 5.39 *If a path $P = (p_1, p_2, \ldots)$ is plausible, then so is any suffix $P_{i>1} = (p_i, p_{i+1}, \ldots)$.*

Proof Easily verified from the definition above. ∎

Proposition 5.40 *A labeled Markov chain $\mathcal{M}$ follows a plausible path with probability 1: $\forall s$, $\mu_s(\{P \mid P \text{ is plausible}\}) = 1$.*

Proof If $Q = (q_1 = s, q_2, q_3, \ldots, q_m)$ is a finite possible path in $\mathcal{M}$, then $\mathcal{M}$ has a positive probability k $(= \Pi_1^{m-1} \delta(q_i, q_{i+1}))$ of following Q each time it visits s. So if s occurs infinitely often in a path P followed by $\mathcal{M}$, then clearly with probability 1 Q occurs infinitely often in P. ∎

Note that the above proof depends on the Markovian (memoryless) property: $\mathcal{M}$ has the same fixed probability k of following Q each time it enters state s, independent of other visits.

Theorem 5.41 *For any plausible path $P = (p_1, p_2, \ldots)$ in a labeled Markov chain $\mathcal{M}$, and LTL formula ϕ, P has a finite prefix $P^n = (p_1, p_2, \ldots, p_n)$ which determines ϕ.*

Note that, for a given P, the n may be different for different ϕ.

Proof Structural induction. Suppose, for any subformula ψ of ϕ and plausible path $Q = (q_1, q_2, \ldots)$, there exists an integer m such that $Q^m = (q_1, q_2, \ldots, q_m)$ determines ψ. Using this hypothesis, we will produce a corresponding n such that P^n determines ϕ.

(Note that here we are only proving the existence of such an n, not that we can compute it or that no smaller n exists. Later, in Lemma 5.45, we will in effect prove the stronger claim that repeated applications of `step()` compute such an n.)

If $\phi = \mathcal{T}$ or $\mathcal{F}$, then any path determines ϕ, even the empty path; so we have $n = 0$. Otherwise, ϕ must be an LTL-BDD of the form $(\alpha\ ?\ \psi : \omega)$, where α is one of a, $X\tau$, or $\tau \mathcal{U} \upsilon$. Now, by the induction hypothesis, $\exists k$ and ℓ such that P^k determines ψ and P^ℓ determines ω. Suppose we also had a j such that P^j determined α. Then, letting $n = \max(j, k, \ell)$, it would follow that P^n determined α, ψ and ω, and therefore also ϕ. So we just need to find such a j.

Consider the three cases for α:

Case 1: $\alpha = a$. We only need to examine p_1 to determine whether $P \models a$. So $j = 1$.

Case 2: $\alpha = X\tau$. Note that by Proposition 5.39, suffix $P_2 = (p_2, p_3, \ldots)$ is plausible. So by the induction hypothesis, $\exists g$ such that $P_2^g = (p_2, \ldots, p_g)$ determines τ. But by the definition of X, $P \models X\tau$ iff $P_2 \models \tau$; so if P_2^g determines τ, then $P^g = (p_1, \ldots, p_g)$ determines $X\tau$. So $j = g$.

Case 3: $\alpha = \tau \mathcal{U} \upsilon$. We can show that, if any suffix $P_{x \geq 1} = (p_x, p_{x+1}, \ldots)$ of P has a prefix determining $\tau \mathcal{U} \upsilon$, then so does P. We do this by showing that if $P_{x>1}$ has a prefix determining $\tau \mathcal{U} \upsilon$, then so does P_{x-1}. Then we can show by case analysis that some such x always exists. It will follow by induction on $x - i$ that some P^j determines $\tau \mathcal{U} \upsilon$.

Suppose, for some $x \geq 1$ and z, P_x^z determines $\tau \mathcal{U} \upsilon$. if $x = 1$, then P^z determines $\tau \mathcal{U} \upsilon$ and we're done. Otherwise, consider $P_{x-1} = (p_{x-1}, p_x, p_{x+1}, \ldots)$. By the definition of X, $P_{x-1} \models X(\tau \mathcal{U} \upsilon)$ iff $P_x \models \tau \mathcal{U} \upsilon$. So, since P_x^z determines $\tau \mathcal{U} \upsilon$, P_{x-1}^z determines $X(\tau \mathcal{U} \upsilon)$. Also, by our original induction hypothesis, there exist g and h such that P_{x-1}^g determines τ and P_{x-1}^h determines υ. It follows that, for $y = \max(g, h, z)$, P_{x-1}^y determines τ, υ and $X(\tau \mathcal{U} \upsilon)$, and therefore also $\upsilon \vee (\tau \wedge X(\tau \mathcal{U} \upsilon))$. But by the definition of $\mathcal{U}$, $\upsilon \vee (\tau \wedge X(\tau \mathcal{U} \upsilon)) \equiv \tau \mathcal{U} \upsilon$. Therefore, if P_x^z determines $\tau \mathcal{U} \upsilon$,

then so does P^y_{x-1}, completing the induction.

So all we need to complete our proof are an x and z such that P^z_x determines $\tau \mathcal{U} v$.

Since there are finitely many states in $\mathcal{M}$, there must be at least one state s occurring infinitely often in P. So we proceed by asking: does there exist a plausible path $Q = (q_1 = t, q_2, \ldots)$ in $\mathcal{M}$, such that t is reachable from s, and $Q \models v$?

Case 3a: No. That is, no plausible path from any state reachable from s satisfies v. Therefore no plausible path starting from s satisfies $\tau \mathcal{U} v$. So, let $p_f = s$ be the first occurrence of s in P. Then the one-state path $P^f_f = (p_f)$ determines $\tau \mathcal{U} v$ (as false). So $x = z = f$.

Case 3b: Yes. By the induction hypothesis, some prefix $Q^m = (q_1 = t, \ldots, q_m)$ of Q determines v; as true, since $Q \models v$. Therefore, since any path satisfying v also satisfies $\tau \mathcal{U} v$, Q^m also determines $\tau \mathcal{U} v$ as true.

Now, P visits s infinitely often, and t is reachable from s, i.e., there is a finite possible path from s to t. So by the plausible path property, P also visits t infinitely often. And then again, by the same property, Q^m occurs infinitely often in P. So let $P^{f-1+m}_f = (p_f, p_{f+1}, \ldots, p_{f-1+m})$ be the first occurrence of Q^m in P: $\forall 1 \leq i \leq m$, $p_{f-1+i} = q_i$. That is, $P^{f-1+m}_f = Q^m$, and therefore P^{f-1+m}_f determines $\tau \mathcal{U} v$ as true. So we have $x = f$, $z = f - 1 + m$.

So in both subcases we have a finite path P^z_x which determines $\tau \mathcal{U} v$. As we saw, it follows that there exists a prefix P^j which does too. Therefore, any plausible path P in $\mathcal{M}$ has a prefix P^n which determines ϕ. ∎

Corollary 5.42 *For any state s in a labeled Markov chain $\mathcal{M}$, and any LTL formula ϕ, $\texttt{prob}(s \models \phi) = 0$ iff no plausible path $P = (p_1 = s, p_2, \ldots)$ starting from s satisfies ϕ.*

Proof Suppose no plausible P starting in s satisfies ϕ. By Proposition 5.40, from any state $\mathcal{M}$ follows a plausible path with probability 1. So with probability 1, the path $\mathcal{M}$ follows starting from s is plausible and therefore doesn't satisfy ϕ. Therefore, $\texttt{prob}(s \models \phi) = 0$.

Conversely, suppose some plausible P from s satisfies ϕ. By Theorem 5.41, P has a determining prefix $P^n = (p_1 = s, p_2, \ldots, p_n)$ such that any plausible path in $\mathcal{M}$ starting with P^n satisfies ϕ. Consider a path $P' = (p'_1 = s, p'_2, \ldots)$ followed by $\mathcal{M}$ from s. Since P^n is possible and finite, the probability k that P' starts with P^n is > 0. And by Proposition 5.40, the probability that $\mathcal{M}$ continues along a plausible

path from p_n is 1. So, the probability that P' both starts with P^n and is plausible, and therefore satisfies ϕ, is also k. Therefore, $\texttt{prob}(s \models \phi) \geq k > 0$. ∎

Corollary 5.43 *For any state s in a labeled Markov chain $\mathcal{M}$, and any LTL formula ψ, $\texttt{prob}(s \models \psi) = 1$ iff every plausible path $P = (p_1 = s, p_2, \ldots)$ starting from s satisfies ψ.*

Proof Follows directly from Corollary 5.42, letting $\phi = \neg\psi$, since $\texttt{prob}(s \models \psi) = 1$ iff $\texttt{prob}(s \models \neg\psi) = 0$, and $P \models \psi$ iff $P \not\models \neg\psi$. ∎

Corollary 5.44 *For any plausible path $P = (p_1 = s, p_2, \ldots)$ in a labeled Markov chain $\mathcal{M}$, and for any LTL until formula $\phi = (\tau \mathcal{U} \upsilon \ ? \ \mathcal{T} \ : \ \mathcal{F})$ unrealizable from s, $P \not\models \phi$.*

Proof By the definition of an unrealizable until (page 34), $\forall t$ reachable from s (including s), $\texttt{prob}(t \models \upsilon) = 0$. So by Corollary 5.42, no plausible path from such a t satisfies υ. But every suffix $P_{i \geq 1} = (p_i, p_{i+1}, \ldots)$ of P is such a path. Therefore no suffix of P (including P itself) satisfies υ, and therefore $P \not\models \phi$. ∎

5.8 measure() and step() are correct

We are now ready for our main results, in Lemmas 5.45 and 5.46 and Theorems 5.47 and 5.48. For ease of exposition the four proofs are presented separately, but in fact they are all parts of a single induction proof on $d = \texttt{udepth}(\phi)$. That is, we assume all four claims hold $\forall\psi$ such that $\texttt{udepth}(\psi) < d$, and infer that each must hold for any ϕ of depth d.

This combined approach is necessary because all four results are interdependent: each needs at least one of the others as an induction hypothesis.

We will refer to this combined induction as "Ind1", and its hypothesis as "IH1" (or, e.g., "IH1 (5.45)", when invoking the Lemma 5.45 part of the hypothesis), to distinguish it from other induction hypotheses we will make.

This section uses most of the definitions from section 5.1, especially: $\texttt{udepth}(\phi)$, a **plausible path** P, P **determines** ϕ, step() **resolves** ϕ along P, expression e **represents** constant k. You may want to review these definitions before proceeding.

The overall structure of the Ind1 proof is as follows:

1. `step()` **resolves any** ϕ **along any plausible** P (Lemma 5.45): Using IH1 (5.47), we show that if this claim holds for all subformulas of ϕ, it must also hold for ϕ. So by a second induction Ind2, this one structural rather than on d, the claim holds $\forall\phi$ of depth d. (The proof also uses a third induction, Ind3, to show that if an until $\tau\mathcal{U}v$ is resolved along some suffix P_x of P, then it is resolved along P.)

2. **Whenever** `measure`(ϕ, s) **calls** `solve`$(x_{\phi s} = e)$**, the coefficient of** $x_{\phi s}$ **in** e **is** < 1 (Lemma 5.46): By the previous proof, Lemma 5.45 holds for ϕ and any plausible path. And by Proposition 5.40, there exists at least one plausible path P which $\mathcal{M}$ can follow from s. We show by another induction Ind4 on the structure of P that Lemma 5.46 holds for the call to `solve`$(x_{\phi s} = e)$.

3. $P \models \phi$ **iff** $P_2 \models$ `step`(ϕ, s) (Theorem 5.47): Another nested structural induction, Ind5. Using IH1 (5.48), we show that if the claim holds for ϕ's subformulas, then it holds for ϕ.

4. `measure`(ϕ, s) **represents** `prob`$(s \models \phi)$ (Theorem 5.48): In the previous two proofs we proved that Lemma 5.46 and Theorem 5.47 hold for all formulas of depth $\leq d$. Using these results, we can complete a final nested induction Ind6, this time on return order: assuming the claim holds for all calls `measure`(ψ, t) returning before the call to `measure`(ϕ, s), where `udepth`$(\psi) \leq$ `udepth`(ϕ), we infer that it holds for `measure`(ϕ, s).

Lemma 5.45 `step()` *resolves any LTL formula* ϕ *along any plausible path* $P = (p_1, p_2, \ldots)$.

Intuitively, this lemma says that `step()` computes the determining prefix asserted to exist by Theorem 5.41. Consequently the proof mirrors the proof to Theorem 5.41.

Proof IH1 (5.48) hypothesis lets us assume that Theorem 5.48 holds for any formula ψ such that `udepth`$(\psi) < d =$ `udepth`(ϕ). If we additionally assume that the claim holds for subformulas of ϕ, we can show that it holds for ϕ as well. It will follow by a second, nested induction Ind2 that the claim holds for all ϕ of depth d.

So, we want to find an n such that `step`$^n(\phi, P) = (P \models \phi) = \mathcal{T}/\mathcal{F}$, assuming that such an n exists for any subformula of ϕ.

If $\phi = \mathcal{T}/\mathcal{F}$, then $n = 0$ and we are done. So consider the remaining case: $\phi = (\alpha\ ?\ \psi : \omega)$, where α is one of a, $X\tau$, or $\tau\mathcal{U}\upsilon$. By IH2, $\exists k$ such that $\mathtt{step}^k(\psi, P) = (P \models \psi)$. And since for any state t, $\mathtt{step}(\mathcal{T}, t) = \mathcal{T}$ and $\mathtt{step}(\mathcal{F}, t) = \mathcal{F}$, it follows that $\forall i \geq k$, $\mathtt{step}^i(\psi, P) = \mathtt{step}^k(\psi, P) = (P \models \psi)$. Similarly, $\exists \ell$ such that $\forall i \geq \ell$, $\mathtt{step}^i(\omega, P) = (P \models \omega)$.

Note that we *cannot* likewise apply IH2 to α, since α is not a subformula (Definition 5.1). Still, suppose somehow we also find a j such that $\mathtt{step}^j(\alpha, P) = (P \models \alpha)$. Then, letting $n = \max(j, k, \ell)$:

$$
\begin{aligned}
& \mathtt{step}^n(\phi, P) \\
= \ & \mathtt{step}^n((\alpha\ ?\ \psi : \omega), P) \\
= \ & \mathtt{step}^n(\mathtt{cond}(\alpha, \psi, \omega), P) && \text{(Prop A.8)} \\
= \ & \mathtt{cond}(\mathtt{step}^n(\alpha, P), \mathtt{step}^n(\psi, P), \mathtt{step}^n(\omega, P)) && \text{(Cor 5.20)} \\
= \ & \mathtt{cond}((P \models \alpha), (P \models \psi), (P \models \omega)) && \text{(defn of } n\text{)} \\
= \ & (P \models \phi) && \text{(defn of } \mathtt{cond}()\text{)}
\end{aligned}
$$

In other words, if α is resolved along P, then so is ϕ, completing Ind2. So to show that $\mathtt{step}()$ resolves ϕ along P, we only need to find a j such that $\mathtt{step}^j(\alpha, P) = (P \models \alpha)$.

Consider the three cases for α:

Case 1: $\alpha = a$. So we have $j = 1$:

$$
\begin{aligned}
& \mathtt{step}^j(\alpha, P) \\
= \ & \mathtt{step}^1(a, P) \\
= \ & \mathtt{step}^0(\mathtt{step}(a, p_1), P_2) && \text{(defn of } \mathtt{step}^1()\text{)} \\
= \ & \gamma(s, a) = (P \models a) && \text{(defns of } \mathtt{step}^0(), \mathtt{step}()\text{)}
\end{aligned}
$$

Case 2: $\alpha = X\tau$. By IH2, $\exists g$ such that $\mathtt{step}^g(\tau, P_2) = (P_2 \models \tau)$. Let $j = g + 1$:

$$
\begin{aligned}
& \mathtt{step}^j(\alpha, P) \\
= \ & \mathtt{step}^{g+1}(X\tau, P) \\
= \ & \mathtt{step}^g(\mathtt{step}(X\tau, p_1), P_2) && \text{(defn of } \mathtt{step}^{g+1}()\text{)} \\
= \ & \mathtt{step}^g(\tau, P_2) && \text{(defn of } \mathtt{step}()\text{)} \\
= \ & (P_2 \models \tau) && \text{(defn of } g\text{)} \\
= \ & (P \models X\tau) && \text{(defn of } X\text{)}
\end{aligned}
$$

Case 3: $\alpha = \tau\mathcal{U}\upsilon$. This is a more subtle case. We can show that, if $\tau\mathcal{U}\upsilon$ is resolved along some suffix $P_{x>1} = (p_x, p_{x+1}, \ldots)$ of P, then it is also resolved along P. We do this by showing that if $\tau\mathcal{U}\upsilon$ is resolved along P_x, then it is resolved along P_{x-1}. Then we can show by case analysis that such an x always exists. It will follow

by a simple (third!) induction Ind3, on $x - i$, that $\texttt{step}()$ resolves $\tau \mathcal{U} \upsilon$ along P.

Suppose there exist x and z such that $\texttt{step}^z(\tau \mathcal{U} \upsilon, P_x) = (P_x \models \tau \mathcal{U} \upsilon)$. Consider $P_{x-1} = (p_{x-1}, p_x, p_{x+1}, \ldots)$. By IH2, $\exists g, h$ such that $\forall i \geq g$, $\texttt{step}^i(\tau, P_{x-1}) = (P_{x-1} \models \tau)$, and $\forall i \geq h$, $\texttt{step}^i(\upsilon, P_{x-1}) = (P_{x-1} \models \upsilon)$. So let $y = \max(g, h, z+1)$. Then $\texttt{step}()$ resolves $\tau \mathcal{U} \upsilon$ along P_{x-1} within y steps, completing Ind3:

$$
\begin{aligned}
& \texttt{step}^y(\tau \mathcal{U} \upsilon, P_{x-1}) \\
= \; & \texttt{step}^{y-1}(\texttt{step}(\tau \mathcal{U} \upsilon, p_{x-1}), P_x) && \text{(defn of } \texttt{step}^y()) \\
= \; & \texttt{step}^{y-1}(\texttt{or}(\texttt{step}(\upsilon, p_{x-1}), \texttt{and}(\texttt{step}(\tau, p_{x-1}), \tau \mathcal{U} \upsilon)), P_x) && \text{(defn of } \texttt{step}()) \\
= \; & \texttt{or}(\texttt{step}^{y-1}(\texttt{step}(\upsilon, p_{x-1}), P_x), \\
& \quad \texttt{and}(\texttt{step}^{y-1}(\texttt{step}(\tau, p_{x-1}), P_x), \texttt{step}^{y-1}(\tau \mathcal{U} \upsilon, P_x))) && \text{(Cor 5.20)} \\
= \; & \texttt{or}(\texttt{step}^y(\upsilon, P_{x-1}), \texttt{and}(\texttt{step}^y(\tau, P_{x-1}), \texttt{step}^{y-1}(\tau \mathcal{U} \upsilon, P_x))) && \text{(defn of } \texttt{step}^y()) \\
= \; & \texttt{or}((P_{x-1} \models \upsilon), \texttt{and}((P_{x-1} \models \tau), (P_x \models \tau \mathcal{U} \upsilon))) && \text{(defn of } y) \\
= \; & (P_{x-1} \models \tau \mathcal{U} \upsilon) && \text{(defn of } \mathcal{U})
\end{aligned}
$$

So all we need to complete our proof are an x and z such that $\texttt{step}^z(\tau \mathcal{U} \upsilon, P_x) = (P_x \models \tau \mathcal{U} \upsilon)$.

Since there are finitely many states in $\mathcal{M}$, there must be at least one state s occurring infinitely often in P. So we proceed by asking: does there exist a plausible path $Q = (q_1 = t, q_2, \ldots)$ in $\mathcal{M}$, such that t is reachable from s, and $Q \models \upsilon$?

Case 3a: No. That is, no plausible path from any state reachable from s satisfies υ. Therefore, letting $P_f = (p_f = s, p_{f+1}, \ldots)$ be the first occurrence of s in P, $P_f \not\models \tau \mathcal{U} \upsilon$.

Also, by Corollary 5.42, for every t reachable from s, $\texttt{prob}(t \models \upsilon) = 0$. That is, $\tau \mathcal{U} \upsilon$ is unrealizable from s (Definition 3.4).

So by the special clause in $\texttt{step}()$ handling unrealizable untils, $\texttt{step}^1(\tau \mathcal{U} \upsilon, P_f) = \texttt{step}(\tau \mathcal{U} \upsilon, p_f) = \mathcal{F}$. The clause relies on calls to $\texttt{measure}(\upsilon, t)$, and therefore on Theorem 5.48, but since $\texttt{udepth}(\upsilon) < \texttt{udepth}(\phi)$ (Definition 5.2), IH1 lets us assume Theorem 5.48 holds for υ. (This is in fact our only use of IH1 in proving Lemma 5.45.) So $x = f$, $z = 1$ and we are done.

Case 3b: Yes. By Theorem 5.41, some prefix $Q^m = (q_1 = t, \ldots, q_m)$ of Q determines υ; as true, since $Q \models \upsilon$.

Since P visits s infinitely often, and t is reachable from s, it follows by the plausibility of P that P also visits t infinitely often. And then again, by the same property, Q^m occurs infinitely often in P. So let $P_f = (p_f, p_{f+1}, \ldots)$ be the first occurrence of Q^m in P: $\forall 1 \leq i \leq m$, $p_{f-1+i} = q_i$. Since P_f begins with Q^m, and Q^m determines υ

as true, $P_f \models v$, and therefore $P_f \models \tau \mathcal{U} v$.

Meanwhile, by IH2, $\exists h'$ such that $\texttt{step}^{h'}(v, P_f) = (P_f \models v) = \mathcal{T}$. So:

$$
\begin{aligned}
& \texttt{step}^{h'}(\tau \mathcal{U} v, P_f) \\
= \; & \texttt{step}^{h'-1}(\texttt{step}(\tau \mathcal{U} v, p_f), P_{f+1}) && \text{(defn of } \texttt{step}^{h'}()) \\
= \; & \texttt{step}^{h'-1}(\texttt{or}(\texttt{step}(v, p_f), \texttt{and}(\texttt{step}(\tau, p_f), \tau \mathcal{U} v)), P_{f+1}) && \text{(defn of } \texttt{step}()) \\
= \; & \texttt{or}(\texttt{step}^{h'-1}(\texttt{step}(v, p_f), P_{f+1}), \\
& \quad \texttt{step}^{h'-1}(\texttt{and}(\texttt{step}(\tau, p_f), \tau \mathcal{U} v), P_{f+1})) && \text{(Cor 5.20)} \\
= \; & \texttt{or}(\texttt{step}^{h'}(v, P_f), \texttt{step}^{h'-1}(\texttt{and}(\texttt{step}(\tau, p_f), \tau \mathcal{U} v), P_{f+1})) && \text{(defn of } \texttt{step}^{h'}()) \\
= \; & \texttt{or}(\mathcal{T}, \texttt{step}^{h'-1}(\texttt{and}(\texttt{step}(\tau, p_f), \tau \mathcal{U} v), P_{f+1})) && \text{(defn of } h') \\
= \; & \mathcal{T} && \text{(defn of } \texttt{or}()) \\
= \; & (P_f \models \tau \mathcal{U} v) && \text{(defn of } f)
\end{aligned}
$$

And we have $x = f$, $z = h'$. So in both subcases, $\texttt{step}()$ resolves $\tau \mathcal{U} v$ along some P_x, and therefore along P. Therefore for every case of α, α is resolved along P, and it follows as we saw above that therefore, so is ϕ. ∎

Lemma 5.46 *Whenever* $\texttt{measure}(\phi, s)$ *calls* $\texttt{solve}(x_{\phi s} = e)$, *any* $x_{\phi s}$*-term in* e *has coefficient* < 1.

Proof (Part of the combined Ind1 proof outlined on page 71.)

We can show that for any given ϕ and s, there exists at least one finite possible path $P = (p_1 = s, p_2, \ldots, p_n)$ from s such that $\texttt{step}^n(\phi, P) = \mathcal{T}/\mathcal{F}$. Letting P be the shortest such path, and letting ϕ_i be shorthand for $\texttt{step}^i(\phi, P)$, we will show that every (p_i, ϕ_i) pair is unique; i.e., $\forall 1 \leq i < j \leq n$, either $p_i \neq p_j$ or $\phi_i \neq \phi_j$ (or both). Then, using P and this property, we can prove the claim by an induction Ind4 on ϕ_{n-j}.

1. P **exists.** By Proposition 5.40, from any state (including s), $\texttt{prob}(\mathcal{M}$ follows a plausible path$) = 1$. So there exists at least one plausible path $Q = (q_1 = s, q_2, \ldots)$ from s, and by the previous proof (Lemma 5.45), $\exists m$ such that $\texttt{step}^m(\phi, Q) = (Q \models \phi) = \mathcal{T}/\mathcal{F}$. That is, $Q^m = (q_1, \ldots, q_m)$ meets the definition of P.

2. P **contains no loops.** We will show that, if P contains an identical pair (p_i, ϕ_i) and (p_j, ϕ_j), i.e., a loop between i and j, we can cut out the loop to get a shorter possible path P' along which $\texttt{step}()$ still resolves ϕ. Since P was defined to be the shortest such path, it will follow that P contains no such loops.

Suppose for some $1 \leq i < j \leq n$ in P, $p_i = p_j$ and $\phi_i = \phi_j$. Let $P' = (p'_1 = p_1, p'_2 = p_2, \ldots, p'_i = p_i = p_j, p'_{i+1} = p_{j+1}, \ldots p'_{m=n-j+i} = p_n)$. Now, the definition of

$\texttt{step}^n(\phi, P)$ (page 51) only depends on the first n states of P, so $\phi'_i = \phi_i = \phi_j$. Then, using Proposition 5.5: $\phi'_{i+1} = \texttt{step}(\phi'_i, p'_{i+1}) = \texttt{step}(\phi_j, p_{j+1}) = \phi_{j+1}$. Similarly, $\phi'_{i+2} = \texttt{step}(\phi'_{i+1}, p'_{i+2}) = \texttt{step}(\phi_{j+1}, p_{j+2}) = \phi_{j+2}, \ldots, \phi'_m = \phi_n = \mathcal{T}/\mathcal{F}$. So we have constructed a shorter possible path P' along which step() resolves ϕ, contradicting the definition of P.

Therefore, $\forall 1 \leq i < j \leq n$, $p_i \neq p_j$ or $\phi_i \neq \phi_j$.

3. Ind4. Let $r_j = r_{\phi_j p_j}$ be an expression returned by a call to $\texttt{measure}(\phi_j, p_j)$, and let $k_{i,j}$ be the coefficient of a variable $x_i = x_{\phi_i p_i}$ in r_j. We show by induction Ind4 on $n - j$ that: $\forall 1 \leq i < j \leq n$, $k_{i,j} < 1$, and this inequality continues to hold through all subsequent variable substitutions in r_j. Lemma 5.46 will then follow by a short argument.

First, suppose $j = n$. Then $\phi_j = \mathcal{T}/\mathcal{F}$, and $\texttt{measure}(\phi_j, p_j)$ returns 1 or 0; so $\forall i$, $k_{i,j} = 0$ and the Ind4 claim holds trivially.

So suppose $1 \leq j < n$, and assume IH4 holds for $j + 1$. We will show that it therefore holds for j. Consider the three cases for $r_j = \texttt{measure}(\phi_j, p_j)$:

Case 1: $\phi_j = \mathcal{T}/\mathcal{F}$. As for ϕ_n, returns 1 or 0, trivially satisfying Ind4.

Case 2: r_j retrieved from the cache. So it must have originally been cached as the one-variable expression $r_j = x_j$ by an earlier call to $\texttt{measure}(\phi_j, p_j)$. Since then, or in the future, r_j may be modified by variable substitutions, but we will show that it satisfies the Ind4 claim through all such substitutions.

At first, $r_j = x_j$. And because we eliminated all loops in P, $\forall i < j$, $x_i \neq x_j$, and therefore $k_{i,j} = 0$.

So suppose variable substitutions take place in r_j. The first such substitution occurs when $\texttt{measure}(\phi_j, p_j)$ calls $\texttt{substitute}(x_j, r'_j)$, right before returning the expression r'_j. In case 3 we will show that the Ind4 claim holds for any such r'_j, and continues to hold throughout all variable substitutions. So, this case reduces to case 3.

Case 3: no cached solution. Then $r_j = \texttt{solve}(x_j = e)$, where e is computed as a weighted sum: $e = \sum_{s'} \delta(p_j, s') \cdot r'_{s'}$, where $r'_{s'} = \texttt{measure}(\phi_{j+1}, s')$.

Now, by Lemma 5.38, the coefficients in each $r'_{s'}$ sum to ≤ 1. And by the definition of a Markov chain (Chapter 2.2.5), $\sum_{s'} \delta(s, s') = 1$. So for the coefficient of any variable in e to be 1, its coefficient would have to be exactly 1 in every $r'_{s'}$.

But because P is a possible path, one of the recursive calls is to $\texttt{measure}(\phi_{j+1}, p_{j+1})$. That is, $r'_{p_{j+1}} = r_{j+1}$. And by IH4, $\forall 1 \leq i < j + 1$ (i.e., $\forall 1 \leq i \leq j$), the

coefficient $k_{i,j+1}$ of x_i in r_{j+1} is < 1. Therefore, $\forall 1 \leq i \leq j$, the coefficient of x_i in e is < 1.

By the definition of `solve()`, $r_j = \texttt{solve}(x_j = e) = \frac{e - kx_j}{1-k}$, where k is the coefficient of x_j in e. We just saw that $k < 1$, so the error case $k = 1$ does not occur. And since $\forall 1 \leq i < j$ the coefficient of x_i in e is < 1, it follows that the coefficient of x_i in $e - kx_j$ is $< 1 - k$. Therefore, the coefficient $k_{i,j}$ of x_i in r_j is $< \frac{1-k}{1-k} = 1$. And since by IH4 the $k_{i,j+1} < 1$ inequality continues to hold throughout all variable substitutions, so does $k_{i,j} < 1$.

So in all three cases, the Ind4 claim holds, completing the induction.

4. Lemma 5.46. Finally, using the Ind4 result and the same reasoning as in case 3 above, we can prove Lemma 5.46. If `measure`(ϕ, s) calls `solve`$(x_{\phi s} = e)$, then e was computed as a weighted sum of recursive calls, one of which was to $r_2 =$ `measure`(ϕ_2, p_2). So by Ind4, the coefficient of $x_1 = x_{\phi s}$ in r_2 is < 1, and therefore so is its coefficient in e. ∎

Theorem 5.47 *For any plausible path $P = (p_1, p_2, p_3, \ldots)$ in a labeled Markov chain $\mathcal{M}$, and LTL-BDD ϕ, $P \models \phi$ iff suffix $P_2 = (p_2, p_3, \ldots) \models$* `step`$(\phi, s)$.

Proof (Third part of the combined Ind1 proof outlined on page 71.)

This proof was sketched in section 3.4.2.

Suppose the claim holds for all subformulas of ϕ. Using IH1 (5.48), we can show that it holds for ϕ as well. It follows by structural induction Ind5 that the claim holds for all ϕ of depth d.

If $\phi = \mathcal{T}/\mathcal{F}$, then `step`$(\phi, s) = \phi$, and trivially $P \models \phi$ iff $P_2 \models \phi$. So assume ϕ is not a boolean but an LTL-BDD of the form $(\alpha\ ?\ \psi : \omega)$, where α is one of a, $X\tau$, or $\tau \mathcal{U} \upsilon$. `step()` begins by computing an α', and then returns $\phi' =$ `cond`$(\alpha', \psi', \omega')$, where $\psi' =$ `step`(ψ, s) and $\omega' =$ `step`(ω, s). Suppose we can prove that for any α' computed by `step()`, $P \models \alpha$ iff $P_2 \models \alpha'$. Then we can show that, whether or not $P \models \alpha$, $P \models \phi$ iff $P_2 \models \phi'$:

- Suppose $P \models \alpha$. Then by supposition, $P_2 \models \alpha'$. And by the definition of ϕ, $P \models \phi$ iff $P \models \psi$. Now, by IH5, $P \models \psi$ iff $P_2 \models \psi'$. But since $P_2 \models \alpha'$, it follows from the definition of ϕ' that $P_2 \models \psi'$ iff $P_2 \models \phi'$. So $P \models \phi$ iff $P_2 \models \phi'$.

- Suppose $P \not\models \alpha$. Then similarly, $P_2 \not\models \alpha'$, and: $(P \models \phi) \Leftrightarrow (P \models \omega) \Leftrightarrow (P_2 \models \omega') \Leftrightarrow (P_2 \models \phi')$.

So to complete the Ind5 induction, and therefore this third part of Ind1, it only remains to show that $P \models \alpha$ iff $P_2 \models \alpha'$.

Consider the cases in `step()` (page 33):

Case 1: $\alpha = a$. So $\alpha' = \gamma(s, a) =$ the truth value of a in s. If $\gamma(s, a) = \mathcal{T}$, then $P \models a$, and trivially $P_2 \models \alpha' = \mathcal{T}$. Or if $\gamma(s, a) = \mathcal{F}$, then $P \not\models a$ and $P_2 \not\models \alpha' = \mathcal{F}$. So in each case, $P \models \alpha$ iff $P_2 \models \alpha'$.

Case 2: $\alpha = X\tau$. $\alpha' = \tau$. So by the definition of X, $P \models \alpha$ iff $P_2 \models \alpha'$.

Case 3: $\alpha = \tau \mathcal{U} \upsilon$. In this case `step()` begins by calling `measure`(υ, t) on every state t reachable from s. (Computing the states in $\mathcal{M}$ reachable from s is easy.) Then there are two subcases:

Case 3a: $\forall t$, `measure`$(\upsilon, t) = 0$. In this subcase, $\alpha' = \mathcal{F}$. Now, `udepth`$(\upsilon) <$ `udepth`(ϕ), so by IH1 (5.48), $\forall t$, `prob`$(t \models \upsilon) = 0$. Therefore $\tau \mathcal{U} \upsilon$ is unrealizable from s (Definition 3.4). And by Corollary 5.44, $P \not\models \tau \mathcal{U} \upsilon$. So it follows that $P \models \alpha$ (false) iff $P_2 \models \alpha'$ (also false).

Case 3b: $\exists t$ such that `measure`$(\upsilon, t) > 0$. So $\alpha' =$ `or`$(\upsilon',$ `and`$(\tau', \tau \mathcal{U} \upsilon))$, where $\upsilon' =$ `step`(υ, s) and $\tau' =$ `step`(τ, s). By IH5, $P \models \tau$ iff $P_2 \models \tau'$, and $P \models \upsilon$ iff $P_2 \models \upsilon'$. And now, by the definition of $\mathcal{U}$, $(P \models \tau \mathcal{U} \upsilon) \Leftrightarrow ((P \models \upsilon) \vee ((P \models \tau) \wedge (P_2 \models \tau \mathcal{U} \upsilon))) \Leftrightarrow ((P_2 \models \upsilon') \vee ((P_2 \models \tau') \wedge (P_2 \models \tau \mathcal{U} \upsilon))) \Leftrightarrow (P_2 \models \alpha')$.

So in every case, $P \models \alpha$ iff $P_2 \models \alpha'$, and as we saw it follows that $P \models \phi$ iff $P_2 \models \phi'$. ∎

Theorem 5.48 *The expression* $r_{\phi s}$ *returned by* `measure`(ϕ, s) *represents* `prob`$(s \models \phi)$.

Proof (Final part of the combined Ind1 proof outlined on page 71.)

The main idea here was sketched in section 3.4.1.

By the previous two proofs, we know that Lemma 5.46 and Theorem 5.47 hold for all ψ such that `udepth`$(\psi) \leq d =$ `udepth`(ϕ), including ϕ itself. Using these results, we can show that if the claim holds for all calls `measure`(ω, u) returning before a call `measure`(ψ, t), where `udepth`$(\omega) \leq$ `udepth`(ψ), then it holds for `measure`(ψ, t). Since `measure`(ϕ, s) terminates (Proposition 5.18), i.e., results in finitely many recursive calls, it follow by induction Ind6 on return order that the claim holds for `measure`(ϕ, s).

For brevity let $z =$ `prob`$(s \models \phi)$. Consider the three cases in `measure()` (page 31):

Case 1: $\phi = \mathcal{T}/\mathcal{F}$. So $r_{\phi s} = 1$ or 0, respectively. Since any path from s satisfies $\mathcal{T}$, and none satisfies $\mathcal{F}$, these return values trivially represent z.

Case 2: solution retrieved from the cache. The cached solution must have originally been stored as $x_{\phi s}$, and then possibly modified by subsequent calls to `substitute()`. It is obvious from definition 5.14 that $x_{\phi s}$ represents z. So we only need to show that this equality wasn't broken by the substitutions.

Suppose a call `substitute`$(x_{\psi t} := r_{\psi t})$ performed a substitution in the cached solution, $r'_{\phi s}$. Then, because $x_{\psi t}$ occurred in $r'_{\phi s}$, it follows by Lemma 5.32 that ϕ and ψ were mutually reducible, and by Corollary 5.24 that `udepth`(ϕ) = `udepth`(ψ). Also, the substitution must have occurred right before `measure`(ψ, t) returned $r_{\psi t}$. So by IH6, $r_{\psi t}$ represents `prob`$(t \models \psi)$. But by Definition 5.14, $x_{\psi t}$ also represents `prob`$(t \models \psi)$. That is, the variable being substituted and the expression being substituted for it represent the same value, and therefore the substitution does not affect the value represented by $r'_{\phi s}$.

So since the cached solution originally represented z, and any subsequent substitutions didn't change the value it represents, it still represents z when retrieved as $r_{\phi s}$.

Case 3: no cached solution. From state s, the Markov chain $\mathcal{M}$ must proceed to some state s', following a path $P = (s, s', p_3, p_4, \ldots)$. By Proposition 5.40, with probability 1 P is plausible. So by the previous proof (Theorem 5.47), $P \models \phi$ iff $P_2 = (s', p_3, p_4, \ldots) \models \phi' =$ `step`(ϕ, s). Now, by the Markovian property, $\mathcal{M}$'s behavior once it enters s' is independent of which s it came from. Therefore, $z = \sum_{s'} \delta(s, s') \cdot$`prob`$(s' \models \phi')$.

By Corollary 5.23, `udepth`$(\phi') \leq$ `udepth`(ϕ). So by IH6, $z = \sum_{s'} \delta(s, s') \cdot$`measure`$(\phi', s')$. This is what the third case of `measure`(ϕ, s) computes as e. Therefore e represents z.

Finally, by the proof of Lemma 5.46, the coefficient k of $x_{\phi s}$ in e is < 1. Therefore $r_{\phi s} =$ `solve`$(x_{\phi s} = e) = \frac{e - k x_{\phi s}}{1-k}$ is computed without error. And since, by Definition 5.14 above, $k x_{\phi s}$ represents $k \cdot z$, $r_{\phi s}$ represents $\frac{z - kz}{1-k} = z$. ∎

Corollary 5.49 *A top-level call to* `measure`(ϕ, s) *returns* `prob`$(s \models \phi)$.

Proof By Corollary 5.33, a top-level call to `measure()` returns a number k. And by Theorem 5.48, k represents `prob`$(s \models \phi)$. Therefore, by Definition 5.14 (page 54), $k =$ `prob`$(s \models \phi)$. ∎

5.9 Complexity

Corollary 5.50 *If $|A|$ is the number of distinct LTL-BDD atoms in an LTL-BDD ϕ, and $|B|$ is the number of distinct non-$\mathcal{T}/\mathcal{F}$ LTL-BDDs passed to* `measure()` *during a call to* `measure`(ϕ, s) *for some s, then:*

$|B| \leq 2^{2^{|A|}}$.

Proof Straight from Corollary 5.17. ∎

Proposition 5.51 *If ϕ is an LTL formula containing no untils, d_X and o are the depth of nested nexts and the number of LTL atom occurrences in ϕ as defined in Definition 4.1, and $|B|$ is the number of LTL-BDDs passed to* `measure()` *as defined in Definition 4.2, then:*

$|B| \leq d_X 2^o$.

Proof Induction on d_X. Assume the claim holds for all formulas ψ such that `xdepth`$(\psi) < d_X =$ `xdepth`(ϕ). We will show that therefore it holds for ϕ.

If ϕ is $\mathcal{T}$ or $\mathcal{F}$, $|B|$ is 0 (since it only counts non-$\mathcal{T}/\mathcal{F}$ LTL-BDDs) and the claim holds trivially. Otherwise, since ϕ contains no untils, it must be a boolean combination of LTL atom nodes of the form $(a\ ?\ \psi : \omega)$ and next nodes of the form $(a\ ?\ \psi : \omega)$. So $d_X \geq 1$.

For any state s, consider $\phi' =$ `step`(ϕ, s). Let $|B|(\phi')$ be the number of distinct LTL-BDDs passed to `measure()` as a result of a call to `measure`(ϕ', t), and $o(\phi')$ be the number of LTL atom occurrences in ϕ'. By Proposition 5.25, `xdepth`$(\phi') < d_X$. So by the induction hypothesis, $|B|(\phi') \leq (d_X - 1)2^{o(\phi')}$.

But note that, since ϕ contains no untils, the value of ϕ' depends entirely on the boolean values of the LTL top-level atoms in ϕ (i.e., not counting those nested within nexts). Let k be the number of these top-level atoms. Then there are at most 2^k possible values of ϕ'. Furthermore, since every occurrence of an atom in ϕ occurs either at the top level or within a next, for any ϕ', $o(\phi') \leq o - k$.

Now, $|B|$ just counts ϕ itself, plus the number of ϕ' and the formulas reducible from them:

$$
\begin{aligned}
& |B| \\
= \; & 1 + \textstyle\sum_{\phi'} |B|(\phi') \\
\leq \; & 1 + 2^k (d_X - 1) 2^{o(\phi')} \\
\leq \; & 1 + 2^k (d_X - 1) 2^{o-k} \\
\leq \; & d_X 2^o
\end{aligned}
$$

Completing the induction. ∎

Proposition 5.52 *If:*

- ϕ *is any LTL-BDD*
- D_k *is the set of LTL-BDDs reducible from* ϕ *within* k *steps, i.e., the set of* ψ *such that for some path* P *and* $0 \leq i \leq k$, $\texttt{step}^i(\phi, P) = \psi$
- $|D_k|$ *is the size of* D_k
- $|\Sigma|$ *is the number of distinct LTL atoms in* ϕ *as defined in Definition 4.1*

then $|D_k| < 2^{k|\Sigma|+1}$.

Proof By the definition of $\texttt{step}^i()$, $\phi_i = \texttt{step}^i(\phi, P)$ depends only on the truth values of atoms in the first i states of P. There are $|\Sigma|$ atoms in P, giving a total of $i|\Sigma|$ boolean variables determining ϕ_i. Therefore, there are no more than $2^{i|\Sigma|}$ different values of ϕ_i.

If $|\Sigma| = 0$, ϕ reduces only to $\mathcal{T}$ or $\mathcal{F}$ and the claim holds trivially. So assume $|\Sigma| \geq 1$. Now, adding up the numbers of ϕ_i for all $0 \leq i \leq k$:

$$
\begin{aligned}
& |D_k| \\
\leq \; & \textstyle\sum_{i=0}^{k} 2^{i|\Sigma|} \\
< \; & 2 \cdot 2^{k|\Sigma|} \\
= \; & 2^{k|\Sigma|+1}
\end{aligned}
$$

∎

Lemma 5.53 *If* ϕ *is an LTL until whose subformulas contain no untils, and* d_X *is the depth of nested nexts in* ϕ *as defined in Definition 4.1, then:*

For any path $P = (p_1, p_2, p_3, \ldots)$ *and* $k \geq d_X - 1$, $\texttt{step}^k(\phi, P) =$ *either* $\mathcal{T}$, $\mathcal{F}$, *or* $\texttt{step}^{d_X - 1}(\phi, P_{k-d_X+2})$, *where* $P_{k-d_X+2} = (p_{k-d_X+2}, p_{k-d_X+3}, \ldots)$.

Proof Induction on k: assuming the claim holds $\forall i$ such that $d_X - 1 \leq i < k$, we will show that it must hold for k.

Let $\phi = \tau \mathcal{U} v$. First, note that Proposition 5.25 implies that, $\forall j \geq d_X$, $\mathtt{step}^j(\tau, P)$ and $\mathtt{step}^j(v, P)$ are both booleans $(\mathcal{T}/\mathcal{F})$.

Now, if $d_X = 0$, τ and v must both be booleans and the claim holds trivially. Similarly, if $k = d_X - 1$, then $\mathtt{step}^k(\phi, P) = \mathtt{step}^{d_X-1}(\phi, P_1) = \mathtt{step}^{d_X-1}(\phi, P_{k-d_X+2})$. So assume $k \geq d_X \geq 1$. Then:

$$
\begin{aligned}
& \mathtt{step}^k(\phi, P) \\
= \; & \mathtt{step}^k(\tau \mathcal{U} v, P) \\
= \; & \mathtt{step}^{k-1}(\mathtt{step}(\tau \mathcal{U} v, p_1), P_2) && \text{(defn of } \mathtt{step}^k()) \\
= \; & \mathtt{step}^{k-1}(\mathtt{or}(\mathtt{step}(v, p_1), \mathtt{and}(\mathtt{step}(\tau, p_1), \tau \mathcal{U} v)), P_2) && \text{(defn of } \mathtt{step}()) \\
= \; & \mathtt{or}(\mathtt{step}^k(v, P), \mathtt{and}(\mathtt{step}^k(\tau, P), \mathtt{step}^{k-1}(\tau \mathcal{U} v, P_2))) && \text{(Cor 5.20)} \\
= \; & \mathtt{or}(\mathcal{T}/\mathcal{F}, \mathtt{and}(\mathcal{T}/\mathcal{F}, \mathtt{step}^{d_X-1}(\tau \mathcal{U} v, P_{k-d_X+2}))) && \text{(Prop 5.25, ind hyp)}
\end{aligned}
$$

Which must work out to either $\mathcal{T}$, $\mathcal{F}$ or $\mathtt{step}^{d_X-1}(\tau \mathcal{U} v, P_{k-d_X+2})$. So the induction holds. ∎

Corollary 5.54 *If:*

- ϕ *is an LTL until whose subformulas contain no untils*
- d_X *is the depth of nested nexts in* ϕ*, as defined in Definition 4.1*
- ϕ_k *is* $\mathtt{step}^k(\phi, P)$ *for some path* P *and* $k \geq d_X - 1$
- $\phi_k \neq \mathcal{T}$ *or* $\mathcal{F}$

then ϕ_k *is determined by* $P^k_{k-d_X+2} = (p_{k-d_X+2}, p_{k-d_X+3}, \ldots, p_k)$*, the last* $d_X - 1$ *states of* P^k*:* $\phi_k = \mathtt{step}^{d_X-1}(\phi, P^k_{k-d_X+2})$.

Proof Follows from (essentially a restatement of) Lemma 5.53. ∎

Corollary 5.55 *If* ϕ *is an LTL until whose subformulas contain no untils, and* d_X*,* $|\Sigma|$ *and* $|B|$ *are as defined in Definitions 4.1 and 4.2, then:*
$|B|$ *is* $O(2^{(d_X-1)|\Sigma|})$.

Proof Proposition 5.31 asserts that every recursive `measure`() call resulting from $\mathtt{measure}(\phi, s)$ is reducible from ϕ. Therefore, a bound on the number of formulas reducible from ϕ is a bound on $|B|$.

Let $\phi = \tau \mathcal{U} v$. By Proposition 5.25 and Lemma 5.53, every formula reducible from ϕ (apart from $\mathcal{T}$ and $\mathcal{F}$) is reducible within $d_X - 1$ steps, i.e., of the form $\phi_k = \mathtt{step}^k(\phi, P)$ or $v_k = \mathtt{step}^k(v, P)$, for some path P and $kleq d_X - 1$. So we only need to count these formulas.

By Proposition 5.52, the number of LTL-BDDs reducible from ϕ within $d_X - 1$ steps is $< 2^{(d_X-1)|\Sigma|+1}$. And similarly so is the number reducible from v. Therefore, $|B| < 2 \cdot 2^{(d_X-1)|\Sigma|+1} = 4 \cdot 2^{(d_X-1)|\Sigma|}$, and therefore $|B|$ is $O(2^{(d_X-1)|\Sigma|})$. ∎

Theorem 5.56 *If ϕ is an LTL formula such that* $\mathtt{udepth}(\phi) \leq 1$, *and d_X, d_i, $|\Sigma|$, u and $|B|$ are as defined in Definitions 4.1 and 4.2, then $|B|$ is $O(2^{(d_X+d_i-1)|\Sigma|}3^u)$.*

Proof As explained in the proof of Corollary 5.55, we can count $|B|$ by counting the number of distinct LTL-BDDs ψ reducible from ϕ. Every such ψ is of one of two forms: $\phi_k = \mathtt{step}^k(\phi, P)$, or $v_k = \mathtt{step}^k(v, P)$ for some until $\tau \mathcal{U} v$ in ϕ.

We can further break down the ϕ_k's to get three types: v_k, $\phi_{k<d_X}$, and $\phi_{k \geq d_X}$. By getting bounds on the number of each type, we can get a bound on $|B|$.

Type 1: v_k. Since $d_U = \mathtt{xdepth}(\phi) \leq 1$, ϕ contains no nested untils, and therefore no v in ϕ contains an until. So by Proposition 5.25, v reduces to $\mathcal{T}$ or $\mathcal{F}$ within d_X steps, and therefore by Proposition 5.52 the number of distinct non-$\mathcal{T}/\mathcal{F}$ formulas each v can reduce to is $O(2^{(d_X-1)|\Sigma|})$. So the total number of LTL-BDDs reducible from an v in ϕ is $O(u2^{(d_X-1)|\Sigma|})$.

Type 2: $\phi_{k<d_X}$. Again, by Proposition 5.52, the number of such ϕ_k is $O(2^{(d_X-1)|\Sigma|})$.

Type 3: $\phi_{k \geq d_X}$. The interesting case.

For any until $\tau \mathcal{U} v$ in *phi*, let $d_e(\tau \mathcal{U} v)$ be the number of nested nexts enclosing $\tau \mathcal{U} v$. For example, if $\phi = X(a \wedge XX(b \mathcal{U} c))$, then $d_e(b \mathcal{U} c) = 3$.

Now, since $k \geq d_X$, every LTL atom or next not contained within an until in ϕ is reduced to $\mathcal{T}$ or $\mathcal{F}$ in ϕ_k. Specifically, these non-until terms are all determined (as $\mathcal{T}/\mathcal{F}$) by the first d_X states in P, i.e., by $P_1^{d_X}$.

It follows that ϕ_k is a boolean combination of until terms, one for each until in ϕ, where the term ζ_i for a given until $\mathcal{U}_i = \tau \mathcal{U} v$ is: $\mathtt{step}^{k-d_e}(\mathcal{U}_i, P_{1+d_e}^k)$ (writing d_e as shorthand for $d_e(\tau \mathcal{U} v)$).

Recall that d_i is the greatest depth of nested nexts in any until in ϕ. So by Corollary 5.54, any such until term ζ_i which is not $\mathcal{T}$ or $\mathcal{F}$ is determined by the last $d_i - 1$ states in P, i.e., by $P_{k-d_i+2}^k$).

So for any path P_1^k, the first d_X states $P_1^{d_X}$ determine the boolean combination of until terms ζ_i, and the last $d_i - 1$ states $P_{k-d_i+2}^k$) determine the possible non-$\mathcal{T}/\mathcal{F}$ value of each ζ_i. All these $d_X + d_i - 1$ states don't determine in ϕ_k is which ζ_i reduce to $\mathcal{T}/\mathcal{F}$ and which don't. So knowing these states, we are left with up to 3^u possible values of ϕ_k, based on choosing whether to reduce each ζ_i to $\mathcal{T}$, $\mathcal{F}$, or its only possible non-$\mathcal{T}/\mathcal{F}$ value.

Therefore, counting the $(d_X + d_i - 1) \cdot |\Sigma|$ possible atom assignments as in the proof of Proposition 5.52, we have the following bound on the total number of possible values of $\phi_{k \geq d_X}$: $2^{(d_X+d_i-1)|\Sigma|}3^u$.

Adding up:

$$\begin{aligned} & |B| \\ = \; & (\#\ \texttt{step}^k(v,\ P)) + (\#\ \texttt{step}^{k<d_X}(\phi,\ P)) + (\#\ \texttt{step}^{k \geq d_X}(\phi,\ P)) \\ = \; & O(u2^{(d_X-1)|\Sigma|}) + O(2^{(d_X-1)|\Sigma|}) + 2^{(d_X+d_i-1)|\Sigma|}3^u \\ = \; & O(2^{(d_X+d_i-1)|\Sigma|}3^u) \end{aligned}$$

■

Corollary 5.57 *If ϕ is an LTL-BDD containing no untils, then the total running time of any calls made by* `measure`$(\phi,\ s)$ *to* `solve()` *and* `substitute()` *is* $O(1)$.

Proof If $\phi = \mathcal{T}$ or $\mathcal{F}$, `measure`$(\phi,\ s)$ returns 0 or 1 immediately and the claim holds trivially. Otherwise, `xdepth`$(\phi) \geq 1$. Consider phi' = `step`$(\phi,\ s)$. By Proposition 5.25, `xdepth`(ϕ') < `xdepth`(ϕ), so $\phi' \neq \phi$. Therefore, Proposition 5.26 implies that ϕ and ϕ' are not mutually reducible. Then, by Corollaries 5.35 and 5.36, `measure`$(\phi,\ s)$ returns a number and performs only a single substitution. It follows that `solve()` and `substitute()` are $O(1)$. ■

Chapter 6

An Implementation: MCMC

MCMC is a straightforward command-line Java implementation of the `measure()` algorithm described by this thesis. You enter a Markov chain $\mathcal{M}$ and some LTL formulas, and it computes the probability of each formula in each state of $\mathcal{M}$.

This chapter goes through a simple example run and outlines the design. Additional material is in Appendix B:

- the README file from the distribution
- a commented sample input file
- a full transcript from a longer run
- a code excerpt: `Checker.java`

The latest version of MCMC (as of this writing, `mcmc-0.9.5`), along with API documentation, is kept at: `www.cs.mcgill.ca/~jacob/mcmc`.

6.1 A sample run

The following transcript shows MCMC being used to solve the example problems traced through in section 3.5.

Note the satisfying/contradicting traces produced for each formula and state. A "satisfying trace" is just shorthand for a path determining the formula as true, and a "contradicting trace" is a path determining it as false (see Definition 5.12).

```
% java MCMC
Please enter a Markov chain, as n lines of n outgoing probabilities, followed
by some lines of atom probabilities (0/1), followed by a blank line:

     0.5  0.4  0.1
     0.7  0    0.3
     0    0    1

a:   1    1    0
b:   1    0    1

Parsed the Markov chain:

          -1-      -2-      -3-
Edge weights:
  1:     0.5      0.4      0.1
  2:     0.7      0        0.3
  3:     0        0        1
Atoms:
  a:     1        1        0
  b:     1        0        1


Now enter formulas to model check, one per line.  Examples:

  "a&c"         (a and c)                a and c are both true.
  "X(!a)"       (next not a)             a is false in the next state.
  "TU(b|c)"     (true until (b or c))    Eventually b is true, or c (or both).
  "aU[5]c"      (a until c within 5)     a is true until, within 5 steps, c is.

"trace" toggles traces, "debug" toggles debug output, "quit"/"exit" quits.

> Xa

Parsed the formula: Xa = Xa
Calculating...
  Prob in state 1:        0.9
    Satisfying trace:         (1,1)
    Contradicting trace:      (1,3)
  Prob in state 2:        0.7
```

```
    Satisfying trace:          (2,1)
    Contradicting trace:       (2,3)
  Prob in state 3:         0
    Satisfying trace:          none
    Contradicting trace:       (3)
Took 0.013 seconds.
So far:
  5 BDDs created, 4 passed to measure().
  35 measure() calls (6 nontriv), 19 step() (6).
  6 var subs, avg 0.17 subs (0.0 var terms) per substitute() call.

> X(bUa)

Parsed the formula: X(bUa) = XU(b,a)
Calculating...
  Prob in state 1:         0.9
    Satisfying trace:          (1,2)
    Contradicting trace:       (1,3)
  Prob in state 2:         0.7
    Satisfying trace:          (2,1)
    Contradicting trace:       (2,3)
  Prob in state 3:         0
    Satisfying trace:          none
    Contradicting trace:       (3)
Took 0.0070 seconds.
So far:
  9 BDDs created, 6 passed to measure().
  73 measure() calls (12 nontriv), 44 step() (14).
  12 var subs, avg 0.16 subs (0.0 var terms) per substitute() call.

> !(TU!a)

Parsed the formula: !(TU!a) = !U(T,!a)
Calculating...
  Prob in state 1:         0
    Satisfying trace:          none
    Contradicting trace:       (1)
  Prob in state 2:         0
    Satisfying trace:          none
    Contradicting trace:       (2)
```

```
  Prob in state 3:        0
    Satisfying trace:          none
    Contradicting trace:       (3)
Took 0.0050 seconds.
So far:
  11 BDDs created, 8 passed to measure().
  101 measure() calls (18 nontriv), 72 step() (23).
  21 var subs, avg 0.21 subs (0.0099 var terms) per substitute() call.

>
```

6.2 Design

MCMC consists of 4300 lines of Java code, divided into 31 classes in 6 packages, but most of these are libraries of no special interest. The main algorithm is in `mcmc.checker.Checker` (Appendix B.4), which contains the implementations of `measure()` and `step()`.

The six packages and their main contents are:

- `mcmc.checker`: `Checker`, containing `measure()`, `step()`, and the top-level `computeProbability()` method; and `Shell`, a simple command-line passing user inputs to `Checker`
- `mcmc.equation`: `Variable` and `Expression`, used to represent the variables ($x_{\phi s}$) and expressions (e, $r_{\phi s}$) manipulated by `measure()`, `solve()` and `substitute()`
- `mcmc.ltl`: `LTLBDDFactory`, containing the `next()`, `until()` and `boundedUntil()` operations on LTL-BDDs; and `LTLParser()`
- `mcmc.bdd`: a generic BDD (ROBDD) implementation
- `mcmc.markov`: `MarkovChain`, a basic matrix-based implementation, and `MarkovChainParser`
- `mcmc.util`: some simple utility classes not specific to MCMC (e.g., `TwoKeyHashMap`)

Four simple test programs are also included, `ExpressionTest`, `LTLTest`, `BDDTest` and `MarkovTest`, and a short `MCMC` wrapper class for ease of compilation and startup.

6.3 Features & limitations

The main priorities in writing MCMC were to make it easy to understand and extend. For ease of comparison, `Checker`'s `measure()` and `step()` methods correspond quite closely to the pseudocode in Definitions 3.2 and 3.3.

(As a glance will indicate, `LTLParser` is one class where the aim of clear readability was not achieved.)

The modularity of the packages is aimed at making improvements as painless as possible. One natural improvement would be a more sophisticated Markov chain implementation (see section 8.2).

Little effort has been devoted to optimization.

The design is loosely object-oriented.

There are fairly extensive comments in the code, though of course not to the level of detail of this thesis.

The bounded until operator (e.g., $a\mathcal{U}^{\leq 5}b$) from section 2.3.4 is supported, with the syntax "`aU[5]b`".

MCMC uses floating-point arithmetic (Java `doubles`), with values very close to 0 treated as 0. Potentially this can result in drastic roundoff errors, particularly in the divisions performed by `solve()` when solving systems of equations. The simplest solution would be to switch to (presumably much slower) exact rational arithmetic. More testing will be needed to determine whether such a switch is worthwhile, i.e., how often and in what situations these errors are serious in practice.

As shown in the sample run above, for each formula ϕ and state s MCMC outputs satisfying and contradicting traces. The algorithm used to find these traces is particularly susceptible to roundoff error, since it must detect 0 and 1 probabilities. Consequently the trace generation feature may not be reliable, or may only produce traces with very low or very high probabilities of satisfying ϕ. Like the roundoff error itself, this is an implementation issue rather than a profound problem.

See also the README file in Appendix B.1.

Chapter 7

Related Work

7.1 Courcoubetis & Yannakakis

The problem addressed by this thesis is one of several related problems solved by Costas Courcoubetis and Mihalis Yannakakis in [CY95]. So their algorithm is the most natural point of comparison. A condensed overview of the algorithm is in [CT97].

Their solution shares with this work the approach of iteratively transforming the input formula until it is reduced to a trivial (until-free) formula. However the approaches are otherwise quite different. The most conspicuous difference is that each of their transformations results not only in a new formula, but also a new Markov chain.

They give two transformations, one for $\mathcal{U}$ and one for X. The $\mathcal{U}$ transformation replaces one "innermost" until in the input formula (an until containing no other untils, i.e., of `udepth`() 1) with a new atom ξ, then modifies the Markov chain so that probabilities of satisfying the formula are preserved. The transformation for X is similar.

Each transformation eliminates one occurrence of the indicated temporal operator in the input formula, so the total number of transformations necessary is linear in the size of the formula. Using this fact and standard graph algorithm results, they are able to prove that their algorithm runs in time singly exponential in the formula size and polynomial in the size (number of nodes and edges) of the Markov chain.

So, their complexity results are stronger than those achieved here in Chapter 4,

where for the general case we have no better bound than doubly exponential in the size of the formula. Furthermore, [CY95] goes on to solve several generalizations, including more expressive specification languages (Büchi automata and ETL, extended temporal logic) and models (concurrent Markov chains, introducing nondeterministic choice). The main advantage of the present work would seem to be that it requires no transformation of the Markov chain. This is significant because in practice model size is the most common limiting factor. Their algorithm may also be difficult to implement, explaining why I have not been able to find an implementation (see also section 8.1).

7.2 Other related work

Formal verification, and even the specific study of probabilistic model checking, is an active field and a full review of contributions is beyond the scope of this work. The following is a selection of results directly related to the problem of LTL verification on probabilistic models.

The most common approach in practice is that of [ASBBS95] and [BCHKR97] (see section 2.5), in which the probability is not computed but encoded in the specification with a language such as PCTL or pCTL*. The output is then simply true or false, and can be computed efficiently using non-probabilistic model checking techniques. The obvious disadvantage is that the probability is not computed, though it can be approximated by binary search. So these techniques are most useful when the exact probability is not needed.

The first algorithm for verifying LTL specifications in probabilistic automata is Vardi's 1985 paper [Var85], which showed how to check whether a formula was satisfied with probability 1. Vardi has also given a more recent overview in [Var99].

Baier's 1998 habilitation [Bai98] includes a detailed survey of verification algorithms for probabilistic systems, including those mentioned above. Kwiatkowska's [Kwi02] tutorial gives a briefer but more recent overview of probabilistic model checking techniques, including references to several current implementations, such as PRISM [KNP02].

The result that every plausible path has a determining prefix (Definitions 5.11 and 5.12, Theorem 5.41) almost surely exists in the literature but I have not yet found it.

Chapter 8

Conclusions

8.1 Summary of results

The main contribution of this work is the `measure()` algorithm for computing the probability that a state s in a labeled Markov chain $\mathcal{M}$ satisfies an LTL formula ϕ. and the accompanying proof of correctness. The complexity bounds, though loose, do establish that the algorithm is no worse than doubly exponential in the size of ϕ and cubic in the size of $\mathcal{M}$.

Three possible advantages of this algorithm over the earlier and more general solution of [CY95] are:

1. Only the (normally small) formula is transformed, rather than the (big) model.

2. Results from MCMC suggest that the bounds in Chapter 4 are pessimistic, and that advantage 1 may make `measure()` more useful for practical verification problems.

3. An implementation exists: MCMC. I have not been able to find an implementation of the [CY95] algorithm.

Additional contributions are the proof that every plausible path has a determining prefix (Theorem 5.41), and the MCMC implementation from Chapter 6.

8.2 Future Work

This work has concentrated on simply proving the algorithm correct and building a trial implementation. Many possible improvements and extensions were left for future investigation. The three most pressing questions seem to be:

1. How does this algorithm compare with the original algorithm of [CY95], in practice?

2. How does this algorithm compare with the PCTL/pCTL* model checking algorithms of work such as [ASBBS95] and [BCHKR97], in practice?

3. What demand is there for a practical algorithm which can compute an exact probability, rather than approximate it with true/false queries via binary search?

These questions immediately suggest some avenues for future work. In rough order of descending urgency:

1. **More efficient implementation.** This is urgent because it is a prerequisite for realistic testing

   The Markov chain representation in MCMC is primitive. Tools such as PRISM [KNP02] use BDD representations and are therefore much more efficient on the large models common in practice. Such a representation could be incorporated into MCMC, or (more likely) the `measure()` algorithm could be incorporated into these more mature tools.

   The technique MCMC uses to solve systems of equations is also crude and might be fruitfully replaced with a more sophisticated linear algebra package.

   As discussed in section 6.3, floating-point handling in MCMC is sloppy and prone to roundoff error, especially in trace generation. Either more careful floating-point usage or substitution of exact rational arithmetic would improve numerical stability.

   There are also more basic potential optimizations, for example, porting the code to C.

2. **Realistic testing.** As discussed in section 4.5, the most productive way of evaluating the algorithm's usefulness would seem to be testing on realistic inputs. The simplest comparison would be to take problems currently being passed to PCTL/pCTL* model checkers, of the form "Does state s satisfy ϕ with probability $\geq k$?", and compare the running time required by a tool like MCMC to answer the corresponding question, "With what probability does s satisfy ϕ?"

   Particularly desirable would be to perform such tests with anyone currently using existing model checkers to do "binary search" probability checking (repeated calls to approximate the exact probability).

3. **Better complexity bounds.** Aside from actual performance improvements, it might be possible to improve many of the bounds in Chapter 4 purely by better analysis. In particular, the conjectures (4.3, 4.4, 4.5, 4.6) need to be resolved.

4. **Extension to more powerful formalisms.** This thesis has dealt only with a single type of model (labeled Markov chains) and specification (LTL). Several variants and generalizations are considered in work such as [CY95], [CT97], and [ASSB96], including models such as concurrent Markov chains and continuous-time Markov processes, and specifications in ETL or as Büchi automata. Which of these the `measure()` algorithm would be easily adaptable to is an open question. One clear restriction is that the caching `measure()` uses to ensure termination would not work for infinite-state systems.

Appendix A

Additional Background

This appendix reviews some background topics more basic than those covered in Chapter 2: the original non-probabilistic model checking algorithm, and BDDs (binary decision diagrams).

A.1 Traditional CTL model checking

We saw in section 2.1 what the purpose of a model checking algorithm is: to take a system description (e.g., some kind of automaton) and a specification (e.g., a formula in a temporal logic), and determine whether the system satisfies the specification. Here I go through the original model checking algorithm developed around 1981 by E Allan Emerson and Ed Clarke. Many of the ideas used by this algorithm will also be useful to us in the probabilistic setting, though not as many as we might hope.

Emerson & Clarke's algorithm takes a specification in CTL, a branching-time temporal logic, as contrasted with the linear-time LTL logic used by this thesis. So I begin with a discussion of branching-time vs linear-time logics, and define CTL syntax. Then I describe Emerson & Clarke's algorithm and go through an example involving an automatic door.

A.1.1 Linear-time vs branching-time logics

Both LTL and CTL are used to phrase assertions about the future states of atoms. The essential difference between them comes down to the distinction between linear-time and branching-time logics. A formula in a **linear-time** logic like LTL makes an

assertion about a single sequence of future states. A single formula in a **branching-time** logic like CTL, however, asserts something about a number of possible futures. That is, whereas an LTL formula describes a single sequential **path** in time, a CTL formula has as its subject a **tree** of possible future paths (or **branches**).

For example, the LTL formula Xa asserts that one step into the future, the atom a will be true. In CTL, some future paths may satisfy a and others may not, so we need to clarify our claim. We might assert for example that a is true in the next step along *all* possible paths: AXa. Or we might assert that *at least one* future path satisfies Xa: EXa. See Figures A.1 and A.2.

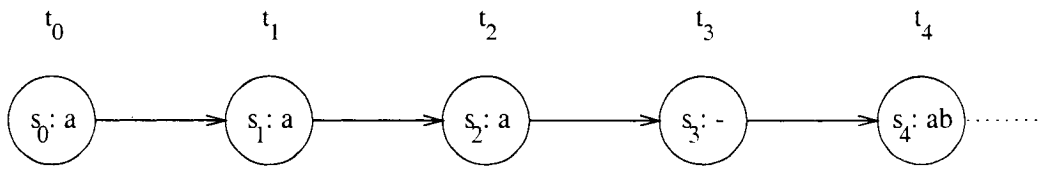

Figure A.1: The linear computation path from Figure 2.5. This path satisfies the LTL formula Xa (but not $XXXa$).

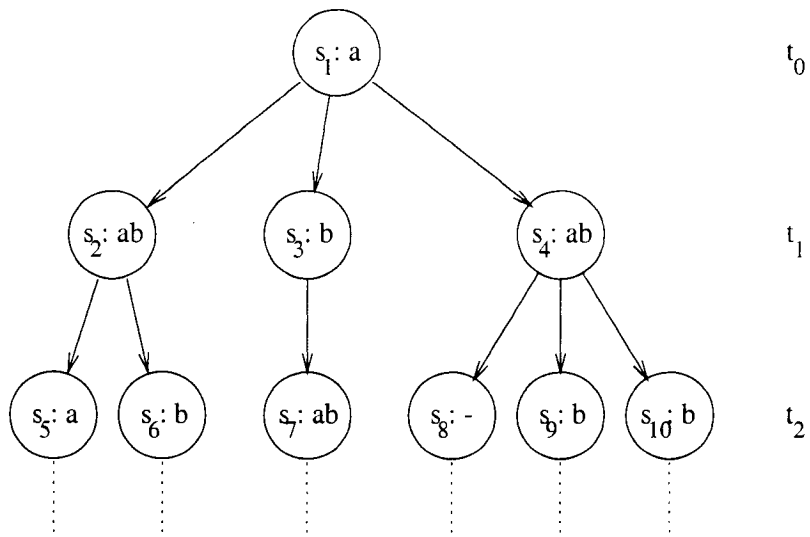

Figure A.2: A computation tree, T. T satisfies the CTL formula EXa, but also $EX\neg a$. It does not satisfy AXa. Meanwhile, EXb and AXb are both satisfied; $EX\neg b$ is not.

The multiplicity of different trees mean there are other meaningful claims we could want to make in a branching-time logic, such as "*exactly one* path satisfies Xa", or "*at least two* paths satisfy Xa." A simple logic like CTL gives us no way to express these assertions: we have only A and E — all, or at least one. However, adding negation lets us derive $\neg E$ ("along no paths") and $\neg A$ ("not along all paths") as well, and these four alone allow us to express many useful claims.

A.1.2 CTL syntax & semantics

CTL syntax (Figure A.3) is close to LTL's. The only new idea is the use of the E/A quantifiers. Otherwise the basic operators are the familiar operators X and $\mathcal{U}$ from LTL, and the usual $\neg$ and $\wedge$ boolean combinators. (The useful temporal operators F ("eventually") and G ("always") can be derived from CTL's $\mathcal{U}$, just as we saw for LTL in section 2.3.4.)

$$
\begin{aligned}
\phi \stackrel{def}{=} \quad & \mathcal{T} \mid \mathcal{F} \\
\mid \quad & a \\
\mid \quad & \neg\psi \\
\mid \quad & \psi \wedge \omega \\
\mid \quad & EX\psi \\
\mid \quad & AX\psi \\
\mid \quad & E(\psi \mathcal{U} \omega) \\
\mid \quad & A(\psi \mathcal{U} \omega)
\end{aligned}
$$

Figure A.3: CTL syntax: the eight types of basic LTL formulas, not counting the derived operators F and G.

Each CTL temporal operator is made up of a **path quantifier** (E/A) followed by a **modality** ($X/\mathcal{U}/F/G$). The modality says what's being asserted about the future paths; the quantifier says which paths the assertion is being made about. For example, the CTL operator $EG\psi$ asserts that there exists at least one path along which ψ is always satisfied. Conversely, $AF\psi$ asserts that ψ is eventually satisfied along every path. Figure A.4 shows the values of some CTL formulas on the computation tree from Figure A.2.

It can be helpful to think of the path quantifier and the modality of a CTL

temporal operator as describing the two dimensions of the assertion. In a computation tree drawn like the one in Figure A.2, the modality specifies the *vertical* extent of the claim, and the path quantifier specifies its *horizontal* extent: you look *down* the tree to check if a property like Xa is satisfied, but you look *across* the tree to see which computation paths satisfy it.

A.1.3 Computation trees and nondeterministic choice

One may wonder when it's useful to make assertions about computation trees, as branching-time logics allow us to do. In particular, constructs such as "always" and "eventually" seem best suited to describing infinite computation trees, and one might well wonder how often these come up. The answer is that they come up often, if one happens to be analyzing the behavior of nondeterministic systems, in which several different next steps may be possible from any given state.

Consider for example the nondeterministic finite automata (NFA) $\mathcal{A}'$ from Figure A.5. Started in state s_1, $\mathcal{A}'$ may move to s_2 at the next step, or stay in s_1. As we unwind further into the future, the (finite) NFA generates an infinite computation tree, part of which is shown in Figure A.6.

So, branching-time logic makes sense in the context of nondeterministic systems. Later we will examine whether it makes sense in probabilistic systems as well.

A.1.4 Example: an automatic door

The original model checking algorithm took a state s in an NFA as the model and a CTL formula ϕ as the specification, and produced a boolean (true/false) value: whether $s \models \phi$ (i.e., whether the tree of possible paths from s satisfies ϕ). Let's look at an example problem, and see how the model checker solves it.

Consider an automatic door, like the door that opens for you when you leave the supermarket. The door has a sensor plate a built into the ground in front of it, and another sensor b in the ground behind it. The door swings open when someone steps on a, and swings shut after they step off b. See Figure A.7.

We, the designers of the door, must make it follow some *protocol* which tells it when to open or close, depending on which sensors currently register someone standing on them. There are certain properties we want our protocol to guarantee: for example,

that the door never swings open, or shut, while b is on. We will model this system as an NFA, and model check it to verify that it satisfies the properties we want.

A.1.5 The NFA model

We can see three state variables, on or off: $a/\neg a$ (sensor a does/doesn't have someone standing on it), similarly $b/\neg b$, and $d/\neg d$ (the door is/isn't open). These three atoms combine to form eight (2^3) possible states for the entire system, as shown in Figure A.8. Our design effort goes into choosing the edges — which transitions our protocol will permit.[1]

A.1.6 The CTL specifications

Now, let's model check for two properties: safety and liveness. By *safety* here we mean that the door never bangs into someone standing on sensor b. *Liveness* is preserved if anyone who steps up to the door eventually gets through.

We need to encode these properties in CTL. Safety consists of two requirements: the door never swings open when b is on, and the door never swings shut when b is on. The first can be expressed as $AG((b \wedge \neg d) \rightarrow \neg EX(b \wedge d))$: "It is always true that, if b is on and the door is closed, it cannot not be true at the next step that b is on and the door has opened." Similarly, the second safety requirement can be written as $AG((b \wedge d) \rightarrow \neg EX(b \wedge \neg d))$. We can combine these to form: $AG(((b \wedge \neg d) \rightarrow \neg EX(b \wedge d)) \wedge ((b \wedge d) \rightarrow \neg EX(b \wedge \neg d)))$.

Liveness can be expressed as: any time a is on and b is on, b will eventually be off (when the person on b leaves), and any time a is on and b is off, b will eventually be on (when the person on a steps through). In CTL this again has two parts, $AG((a \wedge b) \rightarrow AF(\neg b))$ and $AG((a \wedge \neg b) \rightarrow AF(b))$. Again, we can combine these to form $AG(((a \wedge b) \rightarrow AF(\neg b)) \wedge ((a \wedge \neg b) \rightarrow AF(b)))$.

[1]Note that states s_3 and s_4, in which b is true but d is false, are unreachable from the initial state. This makes sense: the only way we allow b to be on is if someone has stepped through the door, in which case the door must be open, since it can't close again until the person has stepped off b. There would be no harm in removing these unreachable states from our system, except that later they might become reachable if we changed our protocol.

A.1.7 Converting to primitive operators

Before proceeding, we will want to get rid of the non-primitive operators in these formulas. Recall that $\rightarrow$, AG, and AF are not part of our technical definition of CTL; we need to replace them with operators defined in Figure A.3, using the helpful equivalences described in section 2.3.

Doing this conversion, we obtain $\neg E(\mathcal{TU}\neg(\neg((b \wedge \neg d) \wedge EX(b \wedge d)) \wedge \neg((b \wedge d) \wedge EX(b \wedge \neg d))))$ for safety, and $\neg E(\mathcal{TU}\neg(\neg((a \wedge b) \wedge \neg A(\mathcal{TU}\neg b)) \wedge \neg((a \wedge \neg b) \wedge \neg A(\mathcal{TU}b))))$ for liveness. These formulas are unreadable. However, they mean the same thing as the old ones, and this conversion can be easily automated, so in general only the computer will need to work with the uglier form.

A.1.8 Checking the properties

Now that we have the model and the specifications, how do we check them? The essential idea is to take advantage of the *compositional* structure of the specifications: a CTL formula is either basic and therefore easy to check directly (booleans and atoms), or made up of a combination of smaller formulas. We will see how to compute whether a formula is satisfied, if we know whether its subformulas are satisfied. This gives us a recursive algorithm for computing the truth of any specification in the model. Figures A.10 and A.11 below show the algorithm being used to check safety in $\mathcal{D}$.

As a simple example, if we can check whether (the tree of possible paths from) a state s satisfies two formulas ψ and ω, we can certainly check whether s satisfies $\psi \wedge \omega$: just check if it satisfies both subformulas. $\neg\psi$ is even easier: any given state s satisfies $\neg\psi$ iff it doesn't satisfy ψ.

More precisely, to model check a (state, formula) pair (ϕ, s), we first compute the truth of all ϕ's subformulas at every state in the system. That is, we check all pairs (ψ, t), where ψ is a subformula of ϕ and t is any state in the system. Since there are finitely many subformulas (which keep getting smaller as you recurse), and finitely many states, this algorithm terminates.

When run with a large input formula ϕ, this model checking algorithm will begin by checking small formulas nested deep within ϕ (at every state in the system), and use them to build up bigger and bigger combinations within ϕ, until finally ϕ itself is computed.

A worthwhile exercise is to use this algorithm to check the liveness property, as defined above, in $\mathcal{D}$. You'll find that the model does not satisfy liveness, because as defined in Figure A.8, our protocol doesn't force people to move: the self-looping edges of states s_4, s_6 and s_8 allow paths such as $s_1 s_2 s_6 s_6 s_6 \ldots$ along which someone never gets through the door. To fix this, we could remove the self-loops. The model checking algorithm would then show that liveness was satisfied.

A.1.9 Checking nexts

Checking formulas constructed with the next operators EX/AX is almost as easy as checking $\neg$ and $\wedge$. Suppose we want to check whether the following proposition is true in $\mathcal{D}$ above: from state s_7, it is possible in two steps to reach a state where d is false. In CTL we would write this $EX(EX\neg d)$. This formula has one subformula: $EX\neg d$. Now, suppose we already knew whether each state satisfied this subformula. Then it would be easy to compute the truth of $EX(EX\neg d)$: we would just see if any of the nodes s_7 can reach in one step (s_5, s_7 and s_8) satisfied $EX\neg d$. (In fact, $s_5 \models EX\neg d$, since s_5 can go to s_1; therefore $s_7 \models EX(EX\neg d)$.)

A.1.10 Checking untils

Computing untils takes a bit more cleverness. To check a formula of the form $E(\psi \mathcal{U} \omega)$, we start as usual by checking the truth values of subformulas ψ and ω at every state. Then we perform a labeling loop to determine which states satisfy the until, $E(\psi \mathcal{U} \omega)$. Figure A.9 shows an example.

The labeling loop proceeds as follows. First of all, any state where ω is true immediately satisfies the until, so we start with these states labeled true and all other states labeled false. Next, notice that any state t labeled false which satisfies ψ and has a *next* state s which has been labeled true, itself satisfies the until; this follows from the definition of the until operator. So we can change the label of t from false to true. Having done so, there may now be another state u where ψ is true and which has an edge to t; so u needs to be labeled true as well.

Repeating this relabeling process, we will eventually reach a point where no further states can be labeled true. (This is guaranteed to happen, since there are finitely many states, and at each step we are only increasing the number of states labeled true.) At

this point, the states labeled true are exactly those which satisfy the original until formula, $E(\psi \mathcal{U} \omega)$. Computing the truth of $A\mathcal{U}$ formulas is very similar.

A.1.11 Producing a counterexample

Model checking algorithms like this have the benefit tha if they determine that the model doesn't satisfy the specification, they generally produce a counterexample computation path. For example, in the course of verifying that liveness does not hold, the algorithm would produce a trace such as the $s_1 s_2 s_6 s_6 s_6 \ldots$ above. This benefit will not concern us much in our work here. However, in practice it does present a serious advantage to the system verifier, who needs to know not just that his system is broken, but how.

For a fuller explanation of the original CTL model-checking algorithm, consult a standard reference such as [CGP00]. The essential idea is that a formula can be decomposed into subformulas, whose model-checked truth values in each state can be used to compute the truth values for the original formula.

A.2 BDDs

BDDs (binary decision diagrams) are an efficient way to represent boolean combinations of variables, such as those represented in traditional propositional logic as $a \wedge b$ or $c \vee \neg(a \rightarrow \neg b)$. In this work, we use BDDs to represent LTL formulas.

This appendix reviews the basic ROBDD structure introduced in [Bry86].

Figure A.12 shows the BDD for $a \wedge b$. There are two types of nodes: those containing atoms (a/b), and the leaf nodes at the bottom containing $\mathcal{T}$ or $\mathcal{F}$. Each atom node has two branches: the (right) one followed if the atom is true, and the (left) one followed if it's false. All paths from the root node at the top eventually reach one of the terminal leaf nodes. So this BDD could be read as follows: "Check a. If it's false, return false. If a is true, check b, and return whether it's true." We can express this in C-like syntax as: $(a\ ?\ \mathcal{T} : (b\ ?\ \mathcal{T} : \mathcal{F}))$, or, using a as shorthand for $(a\ ?\ \mathcal{T} : \mathcal{F})$ and $\neg a$ for $(a\ ?\ \mathcal{F} : \mathcal{T})$, more compactly as $(a\ ?\ \mathcal{T} : b)$. Similarly, the BDD for $c \vee (a \wedge b)$ (Figure A.13) would be transcribed as $(a\ ?\ (b\ ?\ \mathcal{T} : c) : c)$.

A.2.1 ROBDDs

The proper full name of the BDD model we will work with is **ROBDDs**: *reduced, ordered* BDDs. This is the most common variant, and we will just call them BDDs. They are *reduced* in that identical subtrees are merged, to save space; see Figure A.14 for the unreduced form of the $a \wedge b$ BDD from Figure A.12. They are *ordered* in that any path from the root at the top to one of the leaf nodes passes through the atoms in a fixed order. Typically the ordering is lexicographic: all nodes containing a occur above all nodes containing b, a before aa before b, and so on. Enforcing an order on the atom nodes is important for keeping the representation canonical.

A.2.2 Benefits of BDDs

BDDs have a number of nice properties which lead them to be used extensively in model checking, as well as other fields. First of all, they are **compact**: stored on a computer, the BDD representation of a large formula tends to take up much less space than most other forms (including written forms like "$a \wedge b$"). In the worst case, a BDD can be exponential in the number of atoms:

Proposition A.1 *If n_1 is a BDD, A is the set of BDD atoms occurring in n_1, $|n_1|$ is the number of nodes in n_1, and $|A|$ is the size of A, then in the worst case, $|n_1|$ is $O(2^{|A|})$.*

Proof Follows from the observation that n_1 has the structure of a binary tree of height at most $|A|$. ∎

But in practice BDD size is often far smaller than this, closer to polynomial than exponential in $|A|$. This efficiency is probably the most widely appreciated virtue of BDDs. For our purposes, however, a more important benefit is that BDDs are **canonical**: there is only one BDD which represents any given boolean combination of variables. In other words, two formulas with the same meaning (for example, $a \wedge b$ and $\neg(\neg a \vee \neg b)$) always produce the same BDD (Figure A.12).

The fact that BDDs are canonical is handy in various ways. It keeps down the total number of BDDs constructed, since only one BDD will be constructed per boolean combination:

Proposition A.2 *From $|A|$ BDD atoms, exactly $2^{2^{|A|}}$ distinct BDDs can be constructed.*

Proof Each BDD represents a function from the set of atom truth assignments to $\mathcal{T}/\mathcal{F}$. There are $2^{|A|}$ different atom assignments, and therefore $2^{2^{|A|}}$ different functions from these assignments to $\mathcal{T}/\mathcal{F}$. ∎

Canonicity also makes it easy to check equality: given two logical formulas like the above, you can just create the BDD for each and check if they're (syntactically) the same. This ease of checking equality can be helpful when writing algorithms, not only to make them run faster, but to ensure that they terminate at all.

A.2.3 Operations on BDDs

Another useful property of BDDs is that standard boolean operators like `not()`, `and()`, and `or()` can be efficiently computed on them. Here we go through how you could recursively compute `and`(n_1, n_2) for two BDDs n_1 and n_2. Then we give the formal algorithms for all the above operators, and for another useful operator, `cond()`.

The full algorithm for `and`(n_1, n_2) is shown in Definition A.5. Recall that there are only two kinds of BDDs: those containing an atom, of the form $n_1 = (a\ ?\ n_2 : n_3)$, and the primitive booleans $\mathcal{T}$ and $\mathcal{F}$. So, the idea is that there are really only three cases for `and`(n_1, n_2):

1. One of the inputs may be a boolean, $\mathcal{T}$ or $\mathcal{F}$. In either case, the answer is then trivial to compute. For example, if $n_1 = \mathcal{T}$, then `and`(n_1, n_2) is simply n_2.

2. If neither node contains a boolean, than each must contain an atom. Suppose each contains the same atom, a: $n_1 = (a\ ?\ n_3 : n_4)$, $n_2 = (a\ ?\ n_5 : n_6)$. Then, in the case where a is true, the answer will be $n_a =$ `and`(n_3, n_5), which we can compute recursively. Similarly, in the case where a is false, the answer is $n_{\neg a} =$ `and`(n_4, n_6). So, if $n_a \neq n_{\neg a}$, the final answer is the a-node with each of these possible answers as its children: $(a\ ?\ n_a : n_{\neg a})$. If $n_a = n_{\neg a}$, then a's value doesn't matter and we just return n_a.

3. There remains only the case where n_1 and n_2 contain different atoms. Without loss of generality suppose that n_1's atom comes before n_2's alphabetically, and

thus (by the ordered property of ROBDDs) belongs above it in the BDD: $n_1 = (a\ ?\ n_3 : n_4)$, $n_2 = (b\ ?\ n_5 : n_6)$. Then we can recurse as above by considering two cases: either a is true, or it is false. If a is true, the answer is $n_a = \mathbf{and}(n_3, n_2)$. If a is false, the answer is $n_{\neg a} = \mathbf{and}(n_4, n_2)$. So the final answer is: $(a\ ?\ n_a : n_{\neg a})$ (again, unless $n_a = n_{\neg a}$).

The formal definitions of $\mathtt{not}(n_1)$, $\mathtt{and}(n_1, n_2)$, $\mathtt{or}(n_1, n_2)$ and $\mathtt{cond}(n_1, n_2, n_3)$ are below.

The complexity of these algorithms is easy to analyze:

Proposition A.3 *If A is a set of BDD atoms, n_1, n_2 and n_3 are BDDs made up of these atoms, and $|A|$ is the size of A, then in the worst case, the following operations are all $O(2^{|A|})$:*

- $\mathtt{not}(n_1)$
- $\mathtt{not}(n_1, n_2)$
- $\mathtt{or}(n_1, n_2)$
- $\mathtt{cond}(n_1, n_2, n_3)$

Proof A call to $\mathtt{not}(n_1)$ just makes two recursive calls on n_1's child nodes. So, one recursive call is made per node in n_1, and therefore not() is $O(|n_1|)$, where $|n_1|$ is the number of nodes in n_1. By Proposition A.1, $|n_1|$ is $O(2^{|A|})$ in the worst case. So, worst case, $\mathtt{not}(n_1)$ is $O(2^{|A|})$.

Similarly, each nontrivial call to $\mathtt{and}(n_1, n_2)$ makes recursive calls on either n_1's or n_2's children, and therefore every node in n_1 and n_2 is passed to at most one recursive call. So and() is $O(|n_1|+|n_2|)$, and therefore again $O(2^{|A|})$. And since or() and cond() simply make a fixed number of calls to not() and and(), so are they. ∎

Finally, some useful identities relating these operations are listed in Figure A.15.

All are straightforward to prove from the definitions.

Definition A.4

$\texttt{not}(n_1)$:
    if $n_1 = \mathcal{T}$:
        return $\mathcal{F}$
    else if $n_1 = \mathcal{F}$:
        return $\mathcal{T}$
    else, $n_1 = (a\ ?\ n_2 : n_3)$:
        return $(a\ ?\ \texttt{not}(n_2) : \texttt{not}(n_3))$

Definition A.5

```
and(n1, n2):
    if n1 = F or n2 = F:
        return F
    else if n1 = T:
        return n2
    else if n2 = T:
        return n1
    else, n1 = (a1 ? n3 : n4), n2 = (a2 ? n5 : n6):
        if a1 > a2:
            return and(n2, n1)
        else:
            if a1 < a2:
                n_a1 = and(n3, n2)
                n_¬a1 = and(n4, n2)
            else, a1 = a2:
                n_a1 = and(n3, n5)
                n_¬a1 = and(n4, n6)
            if (n_a1 = n_¬a1):
                return n_a1
            else:
                return (a1 ? n_a1 : n_¬a1)
```

`or`(n_1, n_2) can then be defined in terms of the primitive operators `not`() and `and`():

Definition A.6

```
or(n1, n2):
    return not(and(not(n1), not(n2)))
```

Finally we have $\texttt{cond}(n_1, n_2, n_3)$, which returns the BDD which is equivalent to n_2 when n_1 is true, and to n_3 when n_1 is false:

Definition A.7

$\texttt{cond}(n_1, n_2, n_3)$:
    return $\texttt{or}(\texttt{and}(n_1, n_2), \texttt{and}(\texttt{not}(n_1), n_3))$

The following proposition can also be useful:

Proposition A.8 *For any node* $n_1 = (a\ ?\ n_2 : n_3)$, $n_1 = \texttt{cond}(a, n_2, n_3)$ *(shorthand for* $\texttt{cond}((a\ ?\ \mathcal{T} : \mathcal{F}), n_2, n_3)$*).*

Proof Follows trivially from the definition of $(a\ ?\ n_2 : n_3)$. Note however that it does not follow that an arbitrary $\texttt{cond}(n_4, n_5, n_6)$ call necessarily returns $(n_4\ ?\ n_5 : n_6)$, since in general we have no guarantee that n_4 is an atom, or that it belongs above n_5 and n_6. For example, $\texttt{cond}((a\ ?\ \mathcal{T} : b), b, \mathcal{F})$ returns b, not the invalid node "$((a\ ?\ \mathcal{T} : b)\ ?\ b : \mathcal{F})$". ■

Formula	Satisfied by tree T?	Explanation
$EX\neg a$	Yes	Any path through s_3 satisfies $X\neg a$.
AXa	No	Not all paths satisfy Xa, since paths through s_3 don't. (Equivalent to $\neg EX\neg a$.)
AXb	Yes	All paths satisfy Xb, since all pass through s_2, s_3 or s_4.
$EX(EX(a \wedge b))$	Yes	There is at least one path along which $(a \wedge b)$ holds two steps from now: (s_1, s_3, s_7).
$AX(EXa)$	No	It is not true that, from every possible next state, there exists a next state in which a is true: there is no such next state from s_4. (Equivalent to $\neg EX(AX\neg a)$.)
$AG(a \vee b)$	No	It is not true that $(a \vee b)$ is everywhere true: it is false in s_8. (Equivalent to $\neg EF(\neg a \wedge \neg b)$.)
$AF(a \wedge b)$	Yes	Along every branch, $(a \wedge b)$ is eventually true: in s_2, s_4, or s_7.
$EX(AGa)$	No	There is no next state from which a is always true along all paths, because every next state can reach a state in which a is false: s_2 can reach s_6, s_4 can reach $s_8/s_9/s_{10}$, and s_3 itself does not satisfy a. (Equivalent to $\neg AX(EF\neg a)$.)
$A((a \vee b)\mathcal{U}(a \wedge b))$	Yes	Along every path, $(a \vee b)$ holds until $(a \wedge b)$ holds. The s_1s_2 path succeeds at s_2, s_1s_4 succeeds at s_4, and $s_1s_3s_7$ survives down to s_7, where it is satisfied. (Equivalent to the disgusting $\neg E((\neg a \vee \neg b)\mathcal{U}(\neg a \wedge \neg b))$.)

Figure A.4: Some CTL formulas evaluated on the computation tree T from Figure A.2.

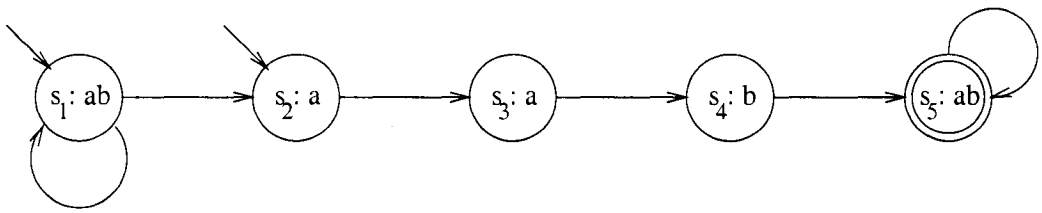

Figure A.5: A state-labeled NFA, $\mathcal{A}'$.

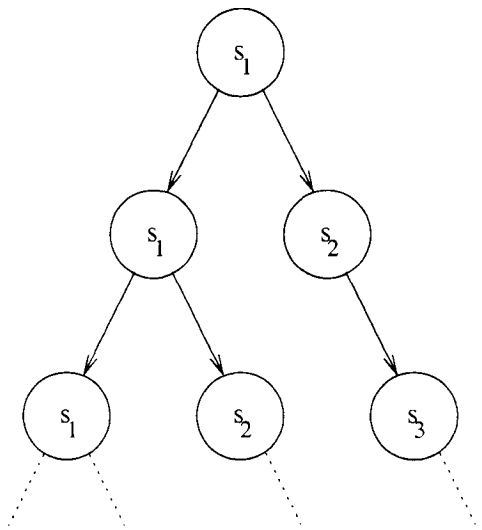

Figure A.6: The first few levels of the infinite computation tree generated by the NFA $\mathcal{A}'$ from Figure A.5.

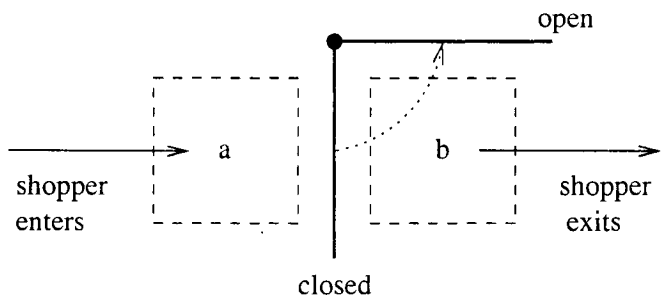

Figure A.7: An automatic door. The door opens automatically when someone steps on sensor a, but must never swing open (or shut) while someone is standing on b.

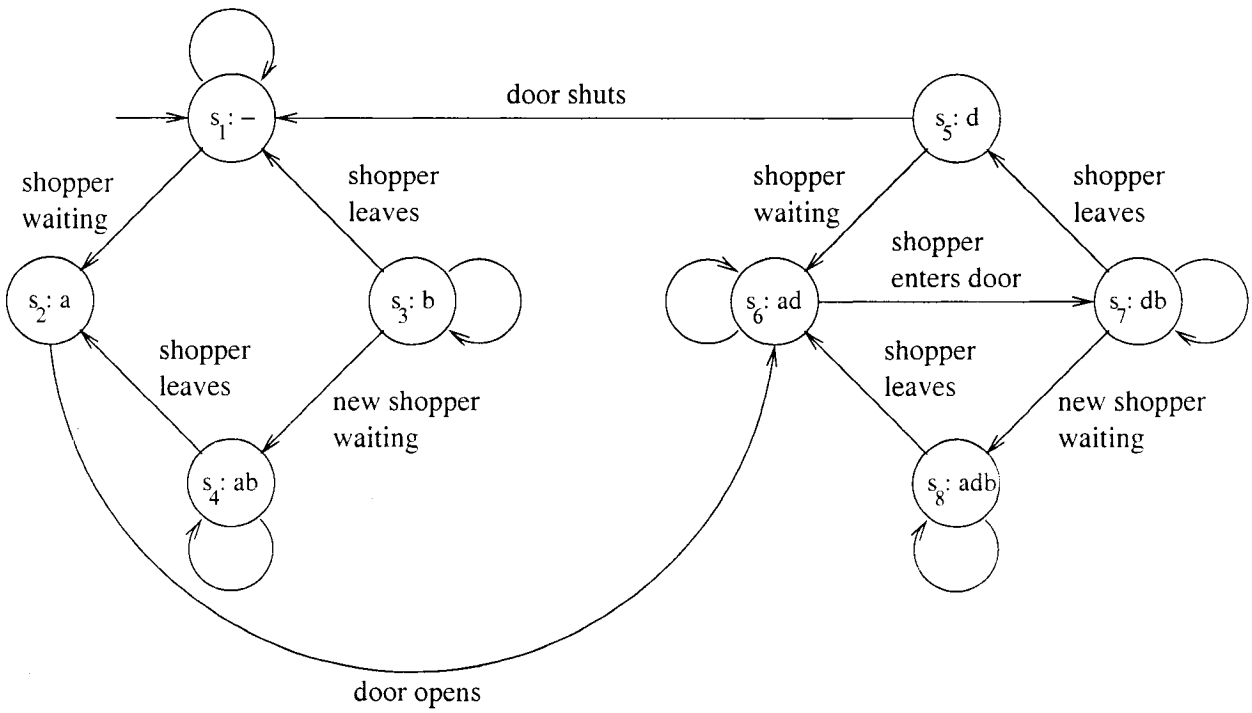

Figure A.8: An NFA $\mathcal{D}$ modeling the automatic door system.

Formula	Satisfied in states	Explanation
a	s_2, s_4, s_6, s_8	First we need the truth values for the ψ (a) and ω ($d \wedge b$) in the $E(\psi \mathcal{U} \omega$ until formula.
$d \wedge b$	s_7, s_8	These are easily obtained from Figure A.8.
$E(a\mathcal{U}(d \wedge b))$	s_7, s_8	To compute the truth values of the until, we begin by labeling $\mathcal{T}$ only those states which immediately satisfy ω ($d \wedge b$).
$E(a\mathcal{U}(d \wedge b))$	s_6, s_7, s_8	Next, we label true any as-yet unlabeled states which: (1) satisfy ψ (a), and (2) have an outgoing edge to a state already labeled true. Of the unlabeled states here, only s_6 satisfies a and has an edge to one of the labeled states (s_7).
$E(a\mathcal{U}(d \wedge b))$	s_2, s_6, s_7, s_8	Repeating the last step. s_2 satisfies a and has an edge to the newly labeled state, s_6. (s_5 has an edge to s_6, but doesn't satisfy a.)
$E(a\mathcal{U}(d \wedge b))$	s_2, s_4, s_6, s_7, s_8	s_4 satisfies a and has an edge to s_2.
$E(a\mathcal{U}(d \wedge b))$	s_2, s_4, s_6, s_7, s_8	None of the remaining states both satisfy a and have an edge to a labeled state, so the algorithm has terminated. All remaining unlabeled states are labeled false. Only those states now labeled true (s_2, s_4, s_6, s_7, and s_8) satisfy $E(a\mathcal{U}(d \wedge b))$.

Figure A.9: An example of model checking an until formula: checking $E(a\mathcal{U}(d \wedge b))$ ("There exists a path along which a is true until a state is reached where d and b are both true") in $\mathcal{D}$ from Figure A.8.

#	Recursive structure	Formula
1	$\mathcal{T}$	$\mathcal{T}$
2	b	b
3	d	d
4	$\neg 3$	$\neg d$
5	$2 \wedge 4$	$b \wedge \neg d$
6	$2 \wedge 3$	$b \wedge d$
7	$EX6$	$EX(b \wedge d)$
8	$5 \wedge 7$	$(b \wedge \neg d) \wedge EX(b \wedge d)$
9	$\neg 8$	$\neg((b \wedge \neg d) \wedge EX(b \wedge d))$
10	$EX5$	$EX(b \wedge \neg d)$
11	$6 \wedge 10$	$(b \wedge d) \wedge EX(b \wedge \neg d)$
12	$\neg 11$	$\neg((b \wedge d) \wedge EX(b \wedge \neg d))$
13	$9 \wedge 12$	$\neg((b \wedge \neg d) \wedge EX(b \wedge d)) \wedge \neg((b \wedge d) \wedge EX(b \wedge \neg d))$
14	$\neg 13$	$\neg(\neg((b \wedge \neg d) \wedge EX(b \wedge d)) \wedge \neg((b \wedge d) \wedge EX(b \wedge \neg d)))$
15	$E(1\mathcal{U}14)$	$E(T\mathcal{U}\neg(\neg((b \wedge \neg d) \wedge EX(b \wedge d)) \wedge \neg((b \wedge d) \wedge EX(b \wedge \neg d))))$
16	$\neg 15$	$\neg E(T\mathcal{U}\neg(\neg((b \wedge \neg d) \wedge EX(b \wedge d)) \wedge \neg((b \wedge d) \wedge EX(b \wedge \neg d))))$

Figure A.10: The order in which the subformulas are model checked, while model checking $\mathcal{D}$ (from Figure A.8) for safety. Aside from booleans and atoms, each row is assembled from previous rows; for example, the final row 16, representing the safety property, is the negation of row 15, which is built from rows 1 and 14.

#	Recursive structure	Truth value in s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
1	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
2	b	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$
3	d	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
4	$\neg 3$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
5	$2 \wedge 4$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
6	$2 \wedge 3$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$
7	$EX6$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
8	$5 \wedge 7$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
9	$\neg 8$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
10	$EX5$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
11	$6 \wedge 10$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
12	$\neg 11$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
13	$9 \wedge 12$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
14	$\neg 13$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
15	$E(1\mathcal{U}14)$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
16	$\neg 15$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$

Figure A.11: Model checking the safety property in $\mathcal{D}$. The truth values of subformulas are computed first, leading to the eventual answer in the last row: the system satisfies safety (no matter which state you start in).

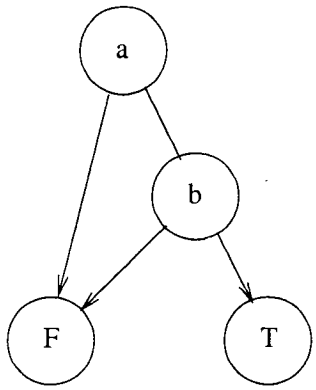

Figure A.12: The BDD representation of the formula $a \wedge b$ (and of all equivalent formulas, such as $\neg(\neg a \vee \neg b)$).

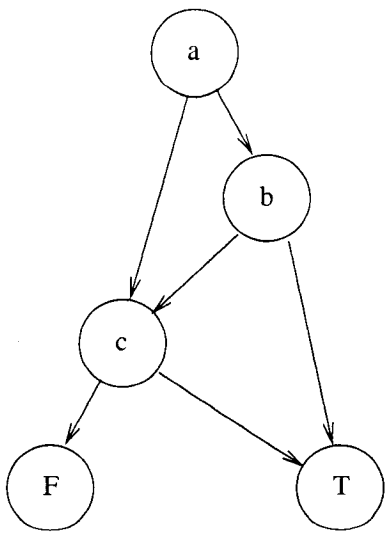

Figure A.13: The BDD representation of the formula $c \vee \neg(a \rightarrow \neg b)$.

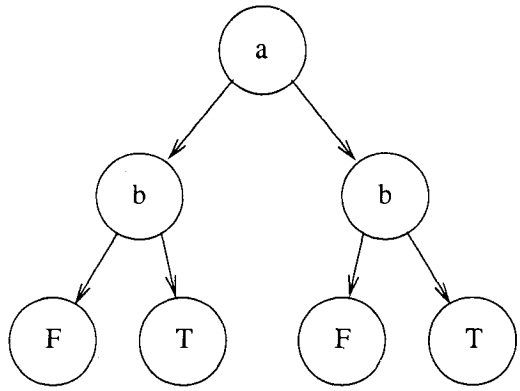

Figure A.14: The unreduced form of the $a \wedge b$ BDD from Figure A.12.

$$
\begin{aligned}
\texttt{not}(\texttt{not}(n_1)) &= n_1 \\
\texttt{and}(n_1, n_2) &= \texttt{and}(n_2, n_1) \\
\texttt{or}(n_1, n_2) &= \texttt{or}(n_2, n_1) \\
\texttt{not}(\texttt{and}(n_1, n_2)) &= \texttt{or}(\texttt{not}(n_1), \texttt{not}(n_2)) \\
\texttt{not}(\texttt{or}(n_1, n_2)) &= \texttt{and}(\texttt{not}(n_1), \texttt{not}(n_2)) \\
\texttt{and}(\texttt{or}(n_1, n_2), n_3) &= \texttt{or}(\texttt{and}(n_1, n_3), \texttt{and}(n_2, n_3)) \\
\texttt{or}(\texttt{and}(n_1, n_2), n_3) &= \texttt{and}(\texttt{or}(n_1, n_3), \texttt{or}(n_2, n_3)) \\
\texttt{cond}(\texttt{not}(n_1), n_2, n_3) &= \texttt{cond}(n_1, n_3, n_2) \\
\texttt{not}(\texttt{cond}(n_1, n_2, n_3)) &= \texttt{cond}(n_1, \texttt{not}(n_2), \texttt{not}(n_3)) \\
\texttt{and}(\texttt{cond}(n_1, n_2, n_3), n_4) &= \texttt{cond}(n_1, \texttt{and}(n_2, n_4), \texttt{and}(n_3, n_4)) \\
\texttt{and}(\texttt{cond}(n_1, n_2, n_3), \texttt{cond}(n_1, n_4, n_5)) &= \texttt{cond}(n_1, \texttt{and}(n_2, n_4), \texttt{and}(n_3, n_5)) \\
\texttt{or}(\texttt{cond}(n_1, n_2, n_3), n_4) &= \texttt{cond}(n_1, \texttt{or}(n_2, n_4), \texttt{or}(n_3, n_4)) \\
\texttt{or}(\texttt{cond}(n_1, n_2, n_3), \texttt{cond}(n_1, n_4, n_5)) &= \texttt{cond}(n_1, \texttt{or}(n_2, n_4), \texttt{or}(n_3, n_5))
\end{aligned}
$$

Figure A.15: Some useful distributivity and commutativity identities relating BDD operations.

Appendix B

MCMC excerpts

This chapter collects some excerpts from MCMC, the Java implementation described in Chapter 6. Included are:

- the README file describing how to get started with MCMC (pages 118-120)
- a commented sample input file (pages 121-123)
- a transcript showing MCMC's output on the sample input file (pages 124-128)
- a code excerpt from `Checker.java`: the core methods `computeProbability()`, `measure()`, `step()`, and `findSatisfyingTraceFrom()` (pages 129-135)

B.1 README

```
This is the readme file for MCMC, a Markov chain model checker.
Last updated March 30th, 2003.



WHAT IT IS:
-----------

    MCMC is a Java program for analyzing the properties of labeled Markov
chains.  You enter a description of a Markov chain and an LTL formula, and it
computes the probability that the formula is true in each node of the Markov
chain.
    The algorithm used is described in my thesis:
www.cs.mcgill.ca/~jacob/thesis.



INSTALLING:
-----------

    These instructions are for running MCMC on a Unix/Linux machine, but it
will also run on Windows or pretty much any platform supporting Java.
    Copy the file mcmc-0.9.5.tar.gz, available at
www.cs.mcgill.ca/~jacob/mcmc, into a directory and untar it:

gunzip mcmc-0.9.5.tar.gz
tar xvf mcmc-0.9.5.tar



COMPILING:
----------

    You'll need Java version 1.2 or higher to compile MCMC:

cd mcmc-0.9.5
make
```

```
    This will also generate the javadoc APIs in a docs subdirectory.


RUNNING:
--------

    From the mcmc-0.9.5 dir:

java MCMC

    Or, for example, to pass in the included sample input file:

java MCMC < sample-input

    The output should be self-explanatory.  It includes:

    - the probability of each input formula being satisfied from each state
in the Markov chain
    - for each formula and state, a satisfying trace and a contradicting
trace (when they exist)
    - diagnostic information: execution time, # LTL-BDDs created, # calls to
measure()/step(), average # variable substitutions/# variable terms
substituted per substitute() call

    A "satisfying trace" for a formula phi is a path determining phi as true:
a finite path, every plausible infinite continuation of which (in the given
Markov chain) satisfies phi.  A "contradicting trace" is a path determining
phi as false.
    That should be enough to get you started.  Use sample-input as a template
and read the examples provided on the command line.  Questions or ambiguities
may be resolved by the comments in classes like mcmc.checker.Checker or
mcmc.checker.Shell.


BUGS:
-----

    This is a proof-of-concept implementation and will break down quickly on
```

large input. The simple Markov chain representation is an obvious target for improvement.

Arithmetic is floating-point, so probabilities close to 0 or 1 are rounded. Do not use MCMC for your air traffic controlling needs. In particular, the satisfying/contradicting traces computed for each formula may only satisfy it with very high/low probabilities, rather than true 1 or 0.

Atom probabilities must be 1 or 0. Intermediate probabilities were handled in a previous version, but seemed useless and slowed the program down.

The code is lightly tested, but on the whole Knuth applies: "Beware of bugs in the above code; I have only proved it correct, not tried it." All bug reports welcome large or small.

LICENSE:

CHANGE HISTORY:

0.9.5:
- bounded untils
- satisfying/contradicting traces
- more informative output: # BDDs, # substitutions
- minor changes

B.2 Sample input

```
#      (Comment lines starting with "#" are ignored by the program.)
#      This is a sample input to the MCMC Markov chain model checking program.
# You can pass it in to the program with a command like this:
#
# java MCMC < sample-input
#
# And it will parse the labeled Markov chain M below (the matrix, not the
# diagram), parse the LTL formulas which follow it at the end of the file, and
# output their probabilities of being true in each state of M.  For example, it
# will say that the formula "a&c" has probability 1 in state 5 and 0 in all
# others, because as shown in the diagram, state 5 is the only state where both
# a and c are true.
#      See the README file in this directory for more usage information.
#      The Markov chain is read in the standard matrix syntax, with a row of
# outgoing edge probabilities for each state.  For example, the 0.7 arrow from
# state s5 to state s1 in the diagram is represented by the 0.7 in the first
# column of the fifth row.  It indicates that when the Markov chain is in state
# 5, it has probability 0.7 (70%) of moving to state 1 at the next step.
#      Atom probabilities are in a similar format.  For example, the fact that
# atom c is true (present) only in states 5 and 6 is indicated by the fact that
# the line starting "c:" contains 1's only in the 5th and 6th columns.
#      LTL syntax:
#
#      - "T" stands for LTL true and is true in all states, "F" for false.
#      - "!a" asserts "not a" (a is false in the current state).
#      - "b&c" asserts "b and c" (b and c are both true), "b|c" means "b or c".
#      - "Xb" asserts "next b": b will be true in the next state.
#      - "aUb" asserts "a until b": b will eventually be true, and in every
# state until it's true, a will be true.
#      - "aU[5]b" asserts "a until b within 5 steps": b will be true within 5
# steps, and in every state until it's true, a will be true.  (The bound is
# inclusive, so, for example, "aU[0]b" is equivalent to "b".)
#
#      So, "Xc" is 20% likely to be true in state 3, because from state 3 the
# machine is 80% likely to stay in state 3 (where c is false), and only 20%
# likely to switch to state 6 where c is true.
#      Another example: "TUa" is true with probability 1 in all states, since
# no matter where you start you'll eventually end up in state 3 or state 4, and
```

```
# in both a is true.  "TUc" is harder.  Try to work out "TU!(TU(a&XXc))" and
# you'll see why I wrote the program.

# Markov chain M, as a diagram:
#
#                                                  -
#                                                 / \
#                                                |   | 0.8
#                                                |   v
#       +-----+            +-----+            +-----+
#       | s1  |       0.8  | s2  |       0.2  | s3  |
#       |     |<---------->|     |----------->|     |
#       |  a  | 0.5        |  b  |            | a b |
#       +-----+            +-----+            +-----+
#          | ^               |                 ^ ^
#          | | 0.7           |              0.1 | | 1
#          |  \              |0.3              /  |
#          |   \_________    |    _________/    |
#          |             \   |   /              |
#          |              \  |  /               |
#      0.1 |           0.1 | | |                | 0.2
#          v               v v |                v
#       +-----+            +-----+            +-----+
#       | s4  |            | s5  |            | s6  |
#       |     |            |     |            |     |
#       |  a  |            | a c |            | b c |
#       +-----+            +-----+            +-----+
#        ^   |             ^   |
#      1 |   |           0.2 |   |
#        \_/                \_/
#

# The same Markov chain M in matrix form:

    0    0.8  0    0.1  0.1  0
    0.5  0    0.2  0    0.3  0
    0    0    0.8  0    0    0.2
    0    0    0    1    0    0
    0.7  0    0.1  0    0.2  0
    0    0    1    0    0    0
```

```
a:   1    0    1    1    1    0
b:   0    1    1    0    0    1
c:   0    0    0    0    1    1

# Input formulas:

a
a&c
X(!a)
TU(b|c)
Xc
TUa
TU[2]c
TU[10]c
TUc
TU!(TU(a&XXc))

#       Copyright 2002-2003, Jacob Eliosoff (jacob@cs.mcgill.ca).
```

B.3 Sample transcript

This chapter shows the output when MCMC is run on the sample input file, with the command `java MCMC < sample-input`. For brevity, trace output was suppressed.

The machine used for the test was a dual-processor 870-MHz Pentium 3, running Debian Linux 3.0 and Java 1.3.1. Total execution time was under a second, due to the small (6-state) input Markov chain. (For this small $\mathcal{M}$, execution time for the more complicated formulas in Figure 4.2 is also under a second.)

```
Please enter a Markov chain, as n lines of n outgoing probabilities, followed
by some lines of atom probabilities (0/1), followed by a blank line:

Parsed the Markov chain:

          -1-     -2-     -3-     -4-     -5-     -6-
Edge weights:
  1:      0       0.8     0       0.1     0.1     0
  2:      0.5     0       0.2     0       0.3     0
  3:      0       0       0.8     0       0       0.2
  4:      0       0       0       1       0       0
  5:      0.7     0       0.1     0       0.2     0
  6:      0       0       1       0       0       0
Atoms:
  a:      1       0       1       1       1       0
  b:      0       1       1       0       0       1
  c:      0       0       0       0       1       1


Now enter formulas to model check, one per line.  Examples:

  "a&c"         (a and c)                 a and c are both true.
  "X(!a)"       (next not a)              a is false in the next state.
  "TU(b|c)"     (true until (b or c))     Eventually b is true, or c (or both).
  "aU[5]c"      (a until c within 5)      a is true until, within 5 steps, c is.

"trace" toggles traces, "debug" toggles debug output, "quit"/"exit" quits.

>
>
>
```

```
Trace output off.

>
Parsed the formula: a = a
Calculating...
  Prob in state 1:        1
  Prob in state 2:        0
  Prob in state 3:        1
  Prob in state 4:        1
  Prob in state 5:        1
  Prob in state 6:        0
Took 0.012 seconds.
So far:
  3 BDDs created, 3 passed to measure().
  19 measure() calls (6 nontriv), 12 step() (6).
  6 var subs, avg 0.32 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: a&c = (a?c:F)
Calculating...
  Prob in state 1:        0
  Prob in state 2:        0
  Prob in state 3:        0
  Prob in state 4:        0
  Prob in state 5:        1
  Prob in state 6:        0
Took 0.01 seconds.
So far:
  5 BDDs created, 4 passed to measure().
  38 measure() calls (12 nontriv), 28 step() (16).
  12 var subs, avg 0.32 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: X(!a) = X!a
Calculating...
  Prob in state 1:        0.8
  Prob in state 2:        0
  Prob in state 3:        0.2
  Prob in state 4:        0
  Prob in state 5:        0
```

```
  Prob in state 6:        0
Took 0.0040 seconds.
So far:
  7 BDDs created, 6 passed to measure().
  70 measure() calls (24 nontriv), 58 step() (28).
  24 var subs, avg 0.34 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: TU(b|c) = U(T,(b?T:c))
Calculating...
  Prob in state 1:        0.9
  Prob in state 2:        1
  Prob in state 3:        1
  Prob in state 4:        0
  Prob in state 5:        1
  Prob in state 6:        1
Took 0.0070 seconds.
So far:
  11 BDDs created, 8 passed to measure().
  107 measure() calls (34 nontriv), 91 step() (40).
  34 var subs, avg 0.32 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: Xc = Xc
Calculating...
  Prob in state 1:        0.1
  Prob in state 2:        0.3
  Prob in state 3:        0.2
  Prob in state 4:        0
  Prob in state 5:        0.2
  Prob in state 6:        0
Took 0.0040 seconds.
So far:
  13 BDDs created, 10 passed to measure().
  139 measure() calls (46 nontriv), 117 step() (48).
  46 var subs, avg 0.33 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: TUa = U(T,a)
Calculating...
```

```
  Prob in state 1:        1
  Prob in state 2:        1
  Prob in state 3:        1
  Prob in state 4:        1
  Prob in state 5:        1
  Prob in state 6:        1
Took 0.0020 seconds.
So far:
  15 BDDs created, 11 passed to measure().
  164 measure() calls (52 nontriv), 143 step() (54).
  52 var subs, avg 0.32 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: TU[2]c = (c?T:X(c?T:Xc))
Calculating...
  Prob in state 1:        0.34
  Prob in state 2:        0.39
  Prob in state 3:        0.36
  Prob in state 4:        0
  Prob in state 5:        1
  Prob in state 6:        1
Took 0.0040 seconds.
So far:
  20 BDDs created, 13 passed to measure().
  196 measure() calls (64 nontriv), 175 step() (70).
  64 var subs, avg 0.33 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: TU[10]c = (c?T:X(c?T:X(c?T:X(c?T:X(c?T:X(c?T:X(c?T:X(c?T:X(c
?T:X(c?T:Xc)))))))))
Calculating...
  Prob in state 1:        0.760044
  Prob in state 2:        0.842524
  Prob in state 3:        0.892626
  Prob in state 4:        0
  Prob in state 5:        1
  Prob in state 6:        1
Took 0.023 seconds.
So far:
  52 BDDs created, 21 passed to measure().
```

```
  306 measure() calls (112 nontriv), 335 step() (150).
  112 var subs, avg 0.37 subs (0.0 var terms) per substitute() call.

>
Parsed the formula: TUc = U(T,c)
Calculating...
  Prob in state 1:        0.833333
  Prob in state 2:        0.916667
  Prob in state 3:        1
  Prob in state 4:        0
  Prob in state 5:        1
  Prob in state 6:        1
Took 0.0080 seconds.
So far:
  54 BDDs created, 22 passed to measure().
  345 measure() calls (118 nontriv), 360 step() (156).
  122 var subs, avg 0.35 subs (0.0029 var terms) per substitute() call.

>
Parsed the formula: TU!(TU(a&XXc)) = U(T,!U(T,(a?XXc:F)))
Calculating...
  Prob in state 1:        0.330579
  Prob in state 2:        0.252066
  Prob in state 3:        0
  Prob in state 4:        1
  Prob in state 5:        0.289256
  Prob in state 6:        0
Took 0.053 seconds.
So far:
  74 BDDs created, 31 passed to measure().
  443 measure() calls (154 nontriv), 511 step() (210).
  192 var subs, avg 0.43 subs (0.0474 var terms) per substitute() call.

>
>
>
```

B.4 Code excerpt: Checker.java

This excerpt contains the core methods from `Checker.java`: `computeProbability()`, `measure()`, `step()`, and `findSatisfyingTraceFrom()`.

```
/**
 * Computes the probability that node s in the MarkovChain satisfies LTL
 * formula phi, ie, the probability that an infinite path starting from s
 * satisfies phi.
 * <p>
 * The essential idea is to "step" P: for P to be true in s, what must be
 * true in the node following s?  (See step().)  Based on this unwinding,
 * the algorithm builds equations relating the probabilities of different
 * formulas in different nodes, then solves the resulting system of
 * equations.
 * <p>
 * The checker caches data about the MarkovChain, so it will screw up if
 * the MarkovChain is modified between calls to this method.
 *
 * @param   phi  an LTL-BDD.
 * @param   s    index of a node in the MarkovChain (1 to n, not 0 to n-1).
 * @return  the probability of phi being satisfied by a path starting at
 * the given node.  Results are cached - the second call with the same
 * arguments will be fast.
 * @see     #step(BDD, int)
 */
public double computeProbability(BDD phi, int s)
{
    Expression solution = measure(phi, s);
    if (solution.getVariableCoefficients().isEmpty()) {
        /* Solution expression is a scalar, as desired: */
        return solution.getScalarTerm();
    } else {
        /* Solution expression has Variables - wasn't fully solved.  This
           should never happen: */
        throw new RuntimeException("Solution to computeExpression(" + phi +
            ", " + s + ") isn't scalar: " + solution);
    }
}

/**
```

```
     * @param   phi  an LTL-BDD.
     * @param   s    index of a node in the MarkovChain (1 to n, not 0 to n-1).
     * @return  a trace starting from the given state which determines phi as
     * true, ie, a finite path P in the Markov chain such that every plausible
     * infinite continuation of P satisfies phi.
     * <p>
     * The trace is represented as an array of state ints (1 to n, not 0 to
     * n-1).  Eg, the array [1,3,1] represents the trace (s1,s3,s1).
     * <p>
     * If no such trace exists in the Markov chain for the given phi and s,
     * returns null.  (This implies that the no plausible path from s satisfies
     * phi, which implies that the probability that a path from s satisfies phi
     * is 0.)
     * <p>
     * This method looks for a short trace, but no guarantees are made about
     * the length (ie, it may not be the shortest).
     */
    public int[] findSatisfyingTraceFrom(BDD phi, int s)
    {
        ArrayList trace = new ArrayList();
        double prob, highestProb;
        ArrayList choices = new ArrayList();
        int choiceIndex;

        while (true) {
            trace.add(new Integer(s));
            prob = computeProbability(phi, s);
            Log.debug("Added s" + s + " to trace (prob " + prob + ")...");

            /* Approximate - values very close to 0 or 1 (within
               Precision.PRECISION) are counted as 0/1: */
            if (prob < Precision.PRECISION) {
                return null;
            } else if (prob > (1 - Precision.PRECISION)) {
                int length = trace.size();
                int[] result = new int[length];
                for (int i = 0; i < length; ++i) {
                    result[i] = ((Integer)trace.get(i)).intValue();
                }
                return result;
```

```
        } else {
            /* Find the successor state in which step(phi, s) is most
               likely to be satisfied: */
            phi = step(phi, s);
            highestProb = 0;
            choices.clear();
            for (int i = 1, z = iMarkovChain.getNumberOfNodes(); i <= z;
                 ++i) {
                if (iMarkovChain.getEdgeWeight(s, i) > 0) {
                    prob = computeProbability(phi, i);
                    if (prob > highestProb + Precision.PRECISION) {
                        choices.clear();
                        choices.add(new Integer(i));
                        highestProb = prob;
                    } else if (prob == highestProb) {
                        choices.add(new Integer(i));
                    }
                }
            }

            choiceIndex = (int)(Math.random() * choices.size());
            s = ((Integer)choices.get(choiceIndex)).intValue();
        }
    }
}

/**
 * Computes an Expression representing the probability that the specified
 * node in the MarkovChain satisfies the given LTL formula.
 *
 * @param   phi  an LTL-BDD.
 * @param   s    index of a node in the MarkovChain (1 to n, not 0 to n-1).
 * @return  an Expression representing the probability of phi being true in
 * node s.
 * @see     #computeProbability(BDD, int)
 */
protected Expression measure(BDD phi, int s)
{
    ++iMeasureCount;
    MEASURE_BDDS.add(phi);
```

```
Expression solution;
if (phi instanceof BDDBoolean) {
    /* Trivial case: */
    solution = ((phi == BDDBoolean.TRUE) ? ONE_EXPRESSION :
                                           ZERO_EXPRESSION);
} else {
    /* Check the cache: */
    Integer sInteger = new Integer(s);
    solution = (Expression)MEASURE_CACHE.get(phi, sInteger);

    if (solution == null) {
        ++iNontrivialMeasureCount;

        /* Create and cache a Variable representing the solution: */
        solution = new Expression();
        Variable x = new Variable();
        solution.add(1, x);
        MEASURE_CACHE.put(phi, sInteger, solution);

        /* Solve recursively by stepping: */
        Expression expr = new Expression();
        int nNodes = iMarkovChain.getNumberOfNodes();
        BDD phiPrime;
        double edgeWeight;
        phiPrime = step(phi, s);
        for (int sPrime = 1; sPrime <= nNodes; ++sPrime) {
            edgeWeight = iMarkovChain.getEdgeWeight(s, sPrime);
            if (edgeWeight > 0) {
                expr.add(edgeWeight,
                         measure(phiPrime, sPrime));
            }
        }

        /* Equate the recursive solution with the Variable.  This step
           should (eventually) reduce every solution Expression to a
           scalar, so that computeProbability() works: */
        try {
            solution = expr.solveFor(x);
        } catch (UnsolvableEquationException e) {
```

```
                e.printStackTrace();
                throw new RuntimeException(
                    "Unexpected exception while solving " +
                    "computeExpression(" + phi + ", " + s + "): " + e);
            }
            x.substitute(solution);
        }
    }
    return solution;
}

/**
 * Computes what must be true at the next step for the given formula phi to
 * be true at this step, based on the atom truth values in the current
 * node s.  Examples:
 * <ul>
 *   <li>step(T, s) = T
 *   <li>step(Xb, s) = b
 *   <li>step(c, s) = T if s(c) (that is, if c is true in s), F if !s(c)
 *   <li>step((Xc)&amp;(aUb), s) is c if s(b); F if !s(b) and !s(a); and
 * c&amp;(aUb) if !s(b) and s(a)
 *   <li>step((!a)UF, s) is F
 * </ul>
 * Note that, as in the last of these examples, step(phi, s) always returns
 * F if phi is an unrealizable until (an until of the form PUQ, where the
 * probability of Q being satisfied is 0 in every node reachable from node
 * s).
 *
 * @param   phi  an LTL-BDD.
 * @param   s    index of a node in the MarkovChain (1 to n, not 0 to n-1).
 * @return  the LTL-BDD which must be true at the next node for phi to be
 * true at the current node.
 */
protected BDD step(BDD phi, int s)
{
    ++iStepCount;

    BDD phiPrime;
    if (phi instanceof BDDBoolean) {
        /* Trivial case: */
```

```
        phiPrime = phi;
    } else {
        /* Check the cache: */
        Integer sInteger = new Integer(s);
        phiPrime = (BDD)STEP_CACHE.get(phi, sInteger);

        if (phiPrime == null) {
            ++iNontrivialStepCount;

            BDDCond phiCond = (BDDCond)phi;
            LTLBDDAtom alpha = (LTLBDDAtom)phiCond.getVariable();
            BDD psi = phiCond.getTrueCase();
            BDD omega = phiCond.getFalseCase();

            BDD alphaPrime = null;
            if (alpha instanceof LTLAtom) {
                LTLAtom alphaAtom = (LTLAtom)alpha;
                String atom = alphaAtom.getAtom();
                alphaPrime = (iMarkovChain.isAtomTrue(atom, s) ?
                              BDDBoolean.TRUE :
                              BDDBoolean.FALSE);
            } else if (alpha instanceof LTLNext) {
                LTLNext alphaNext = (LTLNext)alpha;
                BDD tau = alphaNext.getSubformula();
                alphaPrime = tau;
            } else if (alpha instanceof LTLUntil) {
                LTLUntil alphaUntil = (LTLUntil)alpha;
                BDD tau = alphaUntil.getSubformula1();
                BDD upsilon = alphaUntil.getSubformula2();
                BDD tauUntilUpsilon = LTLBDDFactory.until(tau, upsilon);

                /* Check if the until is realizable: */
                if ((psi != BDDBoolean.TRUE) ||
                    (omega != BDDBoolean.FALSE)) {
                    /* Delegate (so each until is only checked once): */
                    alphaPrime = step(tauUntilUpsilon, s);
                } else {
                    /* Check directly: */
                    TreeSet reachableNodes = getReachableNodesFrom(s);
                    Iterator tIter = reachableNodes.iterator();
```

```
                    boolean allProbsZero = true;
                    for (int t; allProbsZero && tIter.hasNext();) {
                        t = ((Integer)tIter.next()).intValue();
                        allProbsZero &=
                            (computeProbability(upsilon, t) == 0);
                    }
                    if (allProbsZero) {
                        /* The until is unrealizable, so substitute F: */
                        alphaPrime = BDDBoolean.FALSE;
                    } else {
                        BDD tauPrime = step(tau, s);
                        BDD upsilonPrime = step(upsilon, s);
                        alphaPrime =
                            upsilonPrime.or(tauPrime.and(tauUntilUpsilon));
                    }
                }
            }

            if (alphaPrime == BDDBoolean.TRUE) {
                /* So, needn't bother computing omegaPrime: */
                BDD psiPrime = step(psi, s);
                phiPrime = psiPrime;
            } else if (alphaPrime == BDDBoolean.FALSE) {
                /* Similarly, needn't bother computing psiPrime: */
                BDD omegaPrime = step(omega, s);
                phiPrime = omegaPrime;
            } else {
                BDD psiPrime = step(psi, s);
                BDD omegaPrime = step(omega, s);
                phiPrime = alphaPrime.cond(psiPrime, omegaPrime);
            }
            STEP_CACHE.put(phi, sInteger, phiPrime);
        }
    }
    return phiPrime;
}
```

Bibliography

[ASSB96] A Aziz, KK Sanwal, V Singhal, RK Brayton. *Verifying Continuous Time Markov Chains.* In Proc of 8th International Conference on Computer-Aided Verification (CAV '96), vol 1102 of Lecture Notes In Computer Science, pp 269-276. Springer Verlag, 1996.

[ASBBS95] A Aziz, V Singhal, F Balarin, RK Brayton, AL Sangiovanni-Vincentelli. *It Usually Works: The Temporal Logic for Stochastic Systems.* In Proc of 7th International Conference on Computer-Aided Verification (CAV '95), vol 939 of Lecture Notes In Computer Science, pp 155-165. Springer Verlag, 1995.

[Bai98] C Baier. *On the Algorithmic Verification of Probabilistic Systems.* Habilitation Thesis. U Mannheim, 1998.

[BCHKR97] C Baier, E Clark, V Hartonas-Garmhausen, M Kwiatkowska, M Ryan. *Symbolic model checking for probabilistic processes.* In Proc of 24th International Colloquium On Automata Languages And Programming, vol 1256 of Lecture Notes In Computer Science, pp 430-440. Springer Verlag, 1997.

[Bry86] RE Bryant. *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, C-35(8), August 1986, pp 677-691. Reprinted in M Yoeli, *Formal Verification of Hardware Design*, pp 253-267. IEEE Press, 1990.

[CGP00] E Clarke, O Grumberg, D Peled. *Model Checking.* MIT Press, 2000.

[CT97] C Courcoubetis, S Tripakis. *Probabilistic model checking: formalisms and algorithms for discrete and real-time systems.* In summer school Verification of Digital and Hybrid Systems, Antalya, Turkey, 1997.

[CY95] C Courcoubetis, M Yannakakis. *The complexity of probabilistic verification.* Journal of the ACM, 42(4), July 1995, pp 857-907. ACM Press, New York, 1995.

[HJ94] H Hansson, B Jonsson. *A logic for reasoning about time and reliability.* Formal Aspects of Computing, 6(5), pp 512-535, 1994.

[Kwi02] M Kwiatkowska. *Tutorial on Probabilistic Model Checking.* Workshop on Mathematical Models and Techniques for Analysing Systems, Montreal, 2002.

[KNP02] M Kwiatkowska, G Norman and D Parker. *Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach.* In Proc of Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02), vol 2280 of Lecture Notes In Computer Science, pp 52-66. Springer Verlag, 2002.

[Sip97] M Sipser. *Introduction to the Theory of Computation.* Brooks/Cole, 1997.

[Var85] MY Vardi. *Automatic verification of probabilistic concurrent finite-state programs.* In Proc of 26th Symposium on Foundations of Computer Science (FOCS '85), pp 327-338, 1985.

[Var99] MY Vardi. *Probabilistic linear-time model checking: an overview of the automata-theoretic approach.* In Proc of 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS '99), vol 1601 of Lecture Notes In Computer Science, pp 265-276. Springer Verlag, 1999.