Formal Methods

IN

Software Engineering

Computer Science 412/512 Lecture Notes

Spring 2024

Andrew Miner (with modifications by Gianfranco Ciardo)

Last updated: March 17, 2024

Table of Contents

Chapter 1

Introduction

1.1 Model taxonomy

We assume a "model" is a collection of "state variables" which are modified by "actions" (or "events"). We can characterize the following types of models:

Deterministic:

Actions are deterministic; no choices; nothing "random".

Examples:

- Execution of (single–threaded) code, with no inputs.
- Turing machines with fixed input.

Typical questions:

- Can a certain (type of) state be reached?
- Can a certain action, or set of actions, or sequence of actions, occur?
- E.g., "Will the Turing machine halt?"

Non-deterministic:

There may be arbitrary choices between actions. Can be used to model an unknown "environment" or user interactions.

Examples:

- Execution of multi–threaded code, with no inputs (nonderminism arises from the arbitrary choice of thread interleavings).
- Execution of code with user interaction (nonderminism arises from the arbitrary choice of user keystrokes).
- Games of strategy, e.g., chess or checkers (nonderminism arises from the arbitrary choice for each player's move)
- Puzzles, e.g., Rubik's cube (nonderminism arises from the arbitrary choice for which face to turn)

Typical questions:

- Is there a sequence of choices that will cause a certain (type of) state to be reached?
  E.g., "Will two threads modify a file simultaneously (for some thread interleaving)?"
  E.g., "Is there some user input that causes the program to crash?"
- Will a certain (type of) state be reached, or certain actions occur, *regardless of choices*?
  E.g., "Will two threads modify a file simultaneously (for any thread interleaving)?"
  E.g., "Will the program halt regardless of user input?"
  E.g., "If a thread wants to enter critical section, can it eventually do so?"
- Can a certain state be reached, regardless of choices *for some subset of actions*?
  E.g., "Can I win regardless of the other player's choice of moves?"

Stochastic (and deterministic):

Choices between actions are resolved by "random choice" or "random time" to execute actions.

Examples:

- Random inputs to deterministic algorithms.
- Randomized algorithms.
- Games of chance with no strategy (roulette, craps).
- Failure/repair models: failure times are "random", repair times are "random".

Typical questions:

- Probability of "eventually" reaching a certain state.
  E.g., "What is the probability to win craps?"
- Probability of reaching a certain state by a given time.
  E.g., "What is the probability of a deadlock before 100 hours?"
- Expected time to reach a state or execute an action.
  E.g., "What is the expected time of system failure?"
- Other performance queries.
  E.g., "What is the average number of requests processed per second?"
  E.g., "What fraction of the time is the system being repaired?"

Stochastic and nondeterministic:

The model contains both "random choices" and "nondeterministic choices".

Examples:

- Games of chance with strategy (Backgammon, Parcheesi, Monopoly).

Typical questions:

- Probability or performance, based on a given strategy.
  E.g., "If both players make optimal choices, what is the probability to win?"
- Find a strategy to optimize a performance measure.
  E.g., "What strategy will give me the best chance to win?"

We will spend most of our time discussing "nondeterministic" and "stochastic" models, but not mixed. Note that "deterministic" is a special case of both: deterministic is equivalent to "nondeterministic where there is always at most one choice" and to "stochastic where random choices are not really random".

Also, we will focus on models with

- Discrete states (essentially, this means that all state variables can be modeled using natural numbers or integers).
- Actions occur at instantaneous times (rather than continuously); note that this does not mean that actions can only occur at discrete times. Continuous actions (e.g., rocket flight, aerodynamics, weather) typically require differential equations.

1.2 Model construction and analysis

One way to study a system is to build a model and analyze it "from scratch" each time, e.g., following the algorithm given by Leemis and Park from the ComS 455/555 Simulation class:

1. **Determine the goals and objectives of the analysis**. Can be queries about a specific system (e.g., I want to know the mean time until failure, or whether a deadlock is possible). Can be design questions (e.g., how many redundant components are needed to keep the expected time until failure above 10,000 hours).
2. **Build a conceptual model**. Typically an informal diagram. Model must be detailed enough to meet objectives in first step. But too much detail leads to unnecessary complexity. Need to determine the important state variables.
3. **Build a specification model**. Fill in all the details of the conceptual model (usually this requires much more information). E.g., if machine failure times are "random", what is their distribution? E.g., if threads use a locking protocol to prevent simultaneous writes, what are the details of the protocol?
4. **Build a computational model**. We can write a computer program to analyze the specification model. In ComS 455/555, this was a simulation, based on empirical probability measurements and generating random variates. In this class, we will use "model checking" and/or numerical analysis of the underlying stochastic process.
5. **Verify**. Make sure the computational model matches the specification model. In other words, debug your program.
6. **Validate**. Is the computational/specification model an accurate representation of reality? (Compare the results against the real system, if it exists.)

The above approach may require several iterations.

The most expensive steps are implementation and debugging. For this class, we will instead develop a general–purpose implementation that must be debugged only once. Thus, the program must be able to read a specification model. As such, we need:

A specification "language" for models. These may differ slightly based on the type of model (i.e., stochastic or nondeterministic). These are typically called "modeling formalisms".

A specification "language" for queries. We need to express the quantities of interest (e.g., "Is a deadlock possible?", "What is the probability of a deadlock before time 10,000?") in a formal way.

So, we want a tool that takes a model and queries as input, and gives the answers as output.

Chapter 2

Review of propositional logic

2.1 Syntax

Propositional logic consists of

- propositional constants

  `tt` (true) also written as *true* or $\top$ or 1
  `ff` (false) also written as *false* or $\bot$ or 0

- propositional variables

- operators

  $\neg$: negation
  $\wedge$: conjunction (and)
  $\vee$: disjunction (or)
  $\rightarrow$: conditional (implies)
  $\leftrightarrow$: biconditional (iff, if and only if) also written as $\equiv$ for "equivalence"
  $\otimes$: xor or exclusive-or

- complex propositional formulas of the form

  `f` ::= constant | variable | $\neg$`f` | `f` $\wedge$ `f` | `f` $\vee$ `f` | `f` $\rightarrow$ `f` | `f` $\leftrightarrow$ `f` | `f` $\otimes$ `f`

2.2 Semantics

Operators can be defined by specifying, for all possible operand values, the value of the result of the operator on those operand values. *Truth tables* are a natural way to do this. Truth tables may also be used to prove properties about complex formulas (this is a form of "proof by exhaustive enumeration").

2.2.1 Negation

The negation operator inverts the truth value of its operand:

a	$\neg a$
`ff`	`tt`
`tt`	`ff`

The negation operator has the highest precedence.

2.2.2 Conjunction

The conjunction of two formulas is true if and only if both of the formulas evaluate to true:

a	b	$a \wedge b$
ff	ff	ff
ff	tt	ff
tt	ff	ff
tt	tt	tt

Note that the conjunction operator commutes.

2.2.3 Disjunction

The disjunction of two formulas is true if and only if at least one of the formulas evaluates to true:

a	b	$a \vee b$
ff	ff	ff
ff	tt	tt
tt	ff	tt
tt	tt	tt

Note that the disjunction operator commutes.

2.2.4 Conditional

The conditional, or implication, operator is used to express statements of the form, "if A then B". The operator is defined as:

a	b	$a \rightarrow b$
ff	ff	tt
ff	tt	tt
tt	ff	ff
tt	tt	tt

Note that $a \rightarrow b$ evaluates to *true* when a evaluates to *false*. Also, note that this operator **does not** commute.

2.2.5 Biconditional

The biconditional, or biimplication, operator is used to express equivalence, or "if and only if". The operator is defined as:

a	b	$a \leftrightarrow b$
ff	ff	tt
ff	tt	ff
tt	ff	ff
tt	tt	tt

Note that the biconditional operator commutes.

2.2.6 Exclusive-or

The exclusive-or, or xor, operator is used to express mutual exclusion. The operator is defined as:

a	b	$a \leftrightarrow b$
ff	ff	ff
ff	tt	tt
tt	ff	tt
tt	tt	ff

Note that the exclusive-or operator commutes.

2.3 Manipulating formulas

It is possible to express a formula in several different ways. To check if two formulas are equivalent, one sure method is to generate the truth tables for the two formulas: the formulas are equivalent if and only if the truth tables are identical. In practice, this can be done only for formulas with a few variables, because the truth table for a formula with n variables has 2^n rows.

Property 2.1

$$a \to b \quad \equiv \quad (a \wedge b) \vee \neg a \quad \equiv \quad \neg a \vee b$$

Proof: build the truth table

a	b	$a \to b$	$(a \wedge b) \vee \neg a$	$\neg a \vee b$
ff	ff	tt	tt	tt
ff	tt	tt	tt	tt
tt	ff	ff	ff	ff
tt	tt	tt	tt	tt

Property 2.2

$$\neg(a \to b) \quad \equiv \quad a \wedge \neg b$$

Property 2.3

$$a \leftrightarrow b \quad \equiv \quad (a \wedge b) \vee (\neg a \wedge \neg b)$$

Proof: build the truth table

a	b	$a \leftrightarrow b$	$(a \wedge b) \vee (\neg a \wedge \neg b)$
ff	ff	tt	tt
ff	tt	ff	ff
tt	ff	ff	ff
tt	tt	tt	tt

Formulas can be manipulated just like algebraic expressions, where operator $\wedge$ acts like multiplication, and operator $\vee$ acts like addition. However, there are several extra rules that may be

used to simplify or manipulate logic formulas; these are listed below.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l}
\neg\neg x & \equiv & x & & & & \\
x \wedge \texttt{ff} & \equiv & \texttt{ff} & x \vee \texttt{ff} & \equiv & x & \\
x \wedge \texttt{tt} & \equiv & x & x \vee \texttt{tt} & \equiv & \texttt{tt} & \\
x \wedge x & \equiv & x & x \vee x & \equiv & x & \\
x \wedge \neg x & \equiv & \texttt{ff} & x \vee \neg x & \equiv & \texttt{tt} & \\
\neg(a \wedge b) & \equiv & \neg a \vee \neg b & a \wedge b & \equiv & \neg(\neg a \vee \neg b) & \\
\neg(a \vee b) & \equiv & \neg a \wedge \neg b & a \vee b & \equiv & \neg(\neg a \wedge \neg b) & \text{(De Morgan's Laws)} \\
a \vee (b \wedge c) & \equiv & (a \vee b) \wedge (a \vee c) & a \wedge (b \vee c) & \equiv & (a \wedge b) \vee (a \wedge c) & \text{(Distributing/factoring)}
\end{array}$$

All of these are simple to prove using truth tables.

Example 2.1 Prove $a \leftrightarrow b \equiv (a \rightarrow b) \wedge (b \rightarrow a)$.

Solution:

$$\begin{array}{rcl}
(a \rightarrow b) \wedge (b \rightarrow a) & \equiv & \Big((a \wedge b) \vee \neg a\Big) \wedge \Big((b \wedge a) \vee \neg b\Big) \\
& \equiv & (a \wedge b) \wedge (b \wedge a) \quad \vee \quad (a \wedge b) \wedge \neg b \quad \vee \quad \neg a \wedge (b \wedge a) \quad \vee \quad \neg a \wedge \neg b \\
& \equiv & (a \wedge b) \quad \vee \quad \texttt{ff} \quad \vee \quad \texttt{ff} \quad \vee \quad \neg a \wedge \neg b \\
& \equiv & (a \wedge b) \vee (\neg a \wedge \neg b) \\
& \equiv & a \leftrightarrow b \qquad \text{(from Property 2.3)}
\end{array}$$

2.4 Adequate sets of operators

Note that some operators may be expressed in terms of other operators. For example, the biconditional operator can be expressed in terms of conjunction and conditional operators. Thus, the biconditional operator does not add any expressive power to propositional logic. An *adequate set* of operators is a (minimal) set that is powerful enough to express any formula. Often, logics are *defined* in terms of an adequate set.

Example 2.2 Propositional logic may be defined as

$\texttt{f} ::= \texttt{tt} \mid \text{variable} \mid \neg\texttt{f} \mid \texttt{f} \wedge \texttt{f} \mid$

because $\{\neg, \wedge\}$ is an adequate set for propositional logic:

$$\begin{array}{rcl}
\texttt{ff} & \equiv & \neg\texttt{tt} \\
a \vee b & \equiv & \neg(\neg a \wedge \neg b) \\
a \rightarrow b & \equiv & (a \wedge b) \vee \neg a \equiv \neg\Big(\neg(a \wedge b) \wedge a\Big) \\
a \leftrightarrow b & \equiv & (a \wedge b) \vee (\neg a \wedge \neg b) \equiv \neg\Big(\neg(a \wedge b) \wedge \neg(\neg a \wedge \neg b)\Big)
\end{array}$$

2.5 Satisfiability

The *Satisfiability problem*, or SAT, may be expressed as follows.

> Given a formula over propositional variables, determine if it is possible to assign values to the variables so that the entire formula evaluates to true.

This problem is well–known to be NP–complete. All known algorithms that solve this problem have worst–case running times that are at least exponential in the number of propositional variables. However, many SAT solver tools work quite well in practice.

Chapter 3

Introduction to model checking

The idea behind model checking is as follows. Given

- a model of the system, specified via some formalism; and
- properties of the system, specified via some logic;

determine whether the model satisfies the properties. We begin with a basic, "low–level" formalism to describe systems. We will discuss logics starting in the next chapter.

3.1 Kripke structures

A Kripke structure is a formalism that consists of a finite directed graph, where graph vertices represent states of the system and graph edges represent transitions, along with propositions that may or may not hold depending on the current state of the system. Formally, we have the following.

Definition 3.1 A *Kripke structure* is a tuple $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ where

- $\mathcal{S}$ is a nonempty finite set of *states*;
- $\emptyset \subset \mathcal{S}_0 \subseteq \mathcal{S}$ is the set of *initial* states;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation*, such that each state has at least one outgoing edge: $\forall s \in \mathcal{S}, \exists (s, s') \in \mathcal{R}$ for some $s' \in \mathcal{S}$;
- $\mathcal{A}$ is a finite set of *atomic propositions*.
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{A}}$ is a *labeling function*, where $\mathcal{L}(s)$ is the subset of atomic propositions that hold in state s.

Example 3.1 As an example, consider the following simple CD player. There are three buttons:

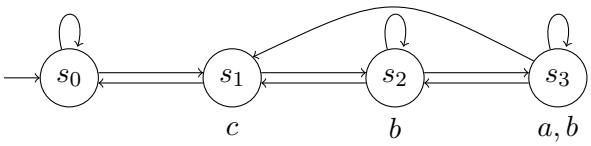
1. open / close
2. stop
3. play

We can model this using a Kripke structure where:

- $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$ with the following meaning
  - s_0: the tray is closed, without a CD
  - s_1: the tray is open
  - s_2: the tray is closed, with a CD, stopped
  - s_3: the tray is closed, with a CD, playing
- $\mathcal{S}_0 = \{s_0\}$
- $\mathcal{R} = \{(s_0, s_0), (s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_2), (s_2, s_3), (s_3, s_1), (s_3, s_2), (s_3, s_3)\}$
- $\mathcal{A} = \{a, b, c\}$ with the following meaning
  - a : playing
  - b : tray is closed with a CD
  - c : tray is open
- $\mathcal{L}$:
  - $\mathcal{L}(s_0) = \{\}$
  - $\mathcal{L}(s_1) = \{c\}$
  - $\mathcal{L}(s_2) = \{b\}$
  - $\mathcal{L}(s_3) = \{a, b\}$

Normally, Kripke structures are *drawn*. We can do this by drawing the graph, indicating the start states, and indicating $\mathcal{L}(s)$ beneath each state s. Alternatively, when state names are unimportant, $\mathcal{L}(s)$ can be drawn inside the graph vertex for state s. We will usually specify Kripke structures by drawing them in either of these ways.

Example 3.2 We can draw the Kripke structure for the CD player from Example 3.1:



Definition 3.2 A *path* in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ is an infinite sequence of states $\pi = (p_0, p_1, p_2, ...) \in \mathcal{S}^\omega$ such that

$$\forall i \geq 0, \qquad (p_i, p_{i+1}) \in \mathcal{R}.$$

Definition 3.3 Given a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$, we let $Paths_M(s)$ denote the set of paths through M that start in state s:

$$Paths_M(s) = \{\pi | \pi = (p_0, p_1, ...) \in \mathcal{S}^\omega, p_0 = s, \pi \text{ is a path in } M\}.$$

We will drop subscript M and simply write $Paths(s)$ if M is clear from context.

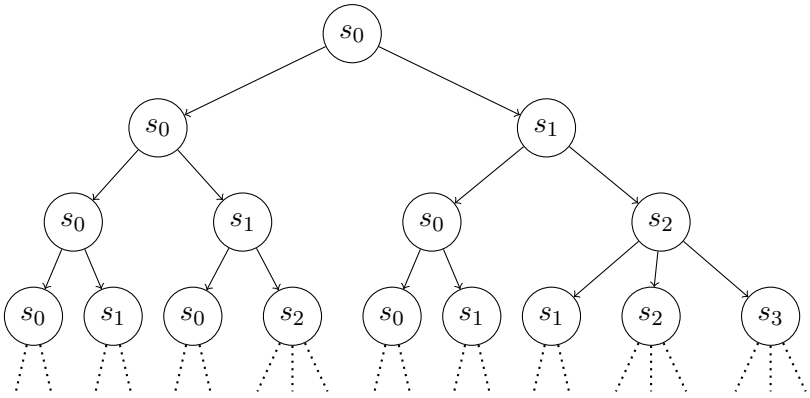
Example 3.3 In the Kripke structure for the CD player from Example 3.1, $\pi = (s_0, s_1, s_1, s_2, ...)$ is *not* a path in the Kripke structure, because $(s_1, s_1) \notin \mathcal{R}$. $\pi = (s_3, s_3, s_2, s_1, s_2, s_2, s_1, s_0, s_1, s_0, s_0, ...)$ is a path in the Kripke structure[1].

[1]Technically it is the prefix of a path, because I did not write out infinitely many states.

3.2 Unfolding a Kripke structure

We can *unfold* a Kripke structure into an infinite tree, called a *computation tree*, as follows. Each node n in the tree is labeled with a Kripke structure state, denoted $L(n)$, and for clarity we will draw $L(n)$ inside node n. In the tree, node n is a child of node m if and only if $(L(m), L(n)) \in \mathcal{R}$, i.e., there's an edge from state $L(m)$ to $L(n)$ in the Kripke structure. The root node of the tree is labeled with the initial state. If there is more than one initial state, we can either draw an imaginary node (with an edge to ech of the initial states) to initialize the Kripke structure, or have a (finite) forest of (infinite) trees, one for each possible initial state. Every path through the tree corresponds to a possible path in the Kripke structure.

Example 3.4 We can draw (part of) the computation tree for the CD player Kripke structure:



Note that for any state s_i, any subtree with root node s_i will be identical, since the subtree corresponds to $Paths(s_i)$.

3.3 Image operations

Two fundamental computations for Kripke structures are *pre-image* and *post-image*.

Pre-image: Given a set of states $\mathcal{X} \subseteq \mathcal{S}$, representing possible states of the system, "now", the pre-image of $\mathcal{X}$ is the set of possible states for the system, "previously" (one step prior to the current state). Formally:

$$PreImage(\mathcal{X}, \mathcal{R}) \;=\; \{s : \exists s' \in \mathcal{X}, (s, s') \in \mathcal{R}\}.$$

Post-image: Given a set of states $\mathcal{X} \subseteq \mathcal{S}$, representing possible states of the system, "now", the post-image of $\mathcal{X}$ is the set of possible states for the system, after the next transition. Formally:

$$PostImage(\mathcal{X}, \mathcal{R}) \;=\; \{s' : \exists s \in \mathcal{X}, (s, s') \in \mathcal{R}\}.$$

Any data structure we use to represent the Kripke structure and sets of states will need to support these two computations efficiently.

Example 3.5 For the CD player from Example 3.1, consider $\mathcal{X} = \{s_0, s_1\}$. The pre-image of $\mathcal{X}$ is $\{s_0, s_1, s_2, s_3\}$ and the post-image of $\mathcal{X}$ is $\{s_0, s_1, s_2\}$.

3.4 Internal representation

For practical models, the Kripke structure can be very large (millions of states is common). A typical representation uses

- an efficient graph data structure to represent $\mathcal{R}$, and
- an efficient data structure to represent subsets of $\mathcal{S}$,

where subsets of $\mathcal{S}$ are used to encode the set of sates satisfying certain properties, such as satisfying an atomic proposition accordingto the labeling function $\mathcal{L}$.

3.4.1 Graph representation

For now we will consider only the "classical" graph data structures.

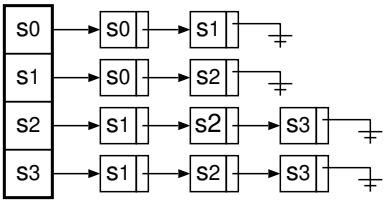
Adjacency matrix:

- Large Kripke structures are usually quite sparse, so a matrix representation is normally not used.
- The adjacency matrix *is* useful to describe operations. More on this, later.

Adjacency list:

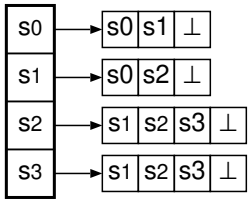
- We can use a straightforward linked list for the outgoing edges from each state, while building the Kripke structure.
- Once the Kripke structure is complete, it often makes sense to convert to a more compact (less dynamic) representation.

Example 3.6 For the CD player from Example 3.1, using a linked list to store the outgoing edges for each state gives the following data structure.



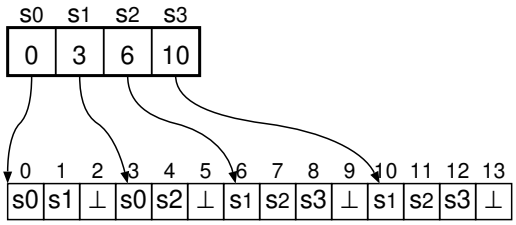
The storage requirement is $|\mathcal{S}| + |\mathcal{R}|$ pointers, and $|\mathcal{R}|$ integers (state indexes).

Example 3.7 The data structure in Example 3.6 can be compacted into a less dynamic structure by replacing each linked list with a null–terminated array, as follows.



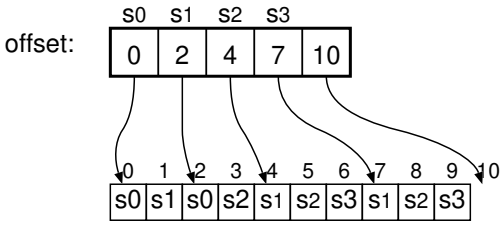
The storage requirement is $|\mathcal{S}|$ pointers, and $|\mathcal{S}| + |\mathcal{R}|$ integers (state indexes). With this data structure, it is more difficult to add new edges.

Example 3.8 The separate arrays from Example 3.7 can be merged into a single array, and the pointers for each state can be replaced by array offsets, giving us the following data structure.



The storage requirement is $|\mathcal{S}|$ array offsets, and $|\mathcal{S}| + |\mathcal{R}|$ integers (state indexes). With this data structure, it is even more difficult to add new edges.

Example 3.9 Looking at the data structure from Example 3.8, we notice that the "$\perp$" sentinels serve only to indicate the end of a list. However, since the next list starts immediately after, it is possible to know the end of a list without using sentinels, as follows.



Since list i ends just before list $i+1$ begins, we have that list i runs from $\mathit{offset}[i]$ to $\mathit{offset}[i+1]-1$. The storage requirement here is $|\mathcal{S}|+1$ array offsets, and $|\mathcal{R}|$ integers (state indexes). This is known as *compressed sparse row format*, widely used in numerical linear algebra implementations.

3.4.2 Subset representation

To store a subset of $\mathcal{S}$, in practice[2] we need a way to store subsets of the integers $\{0, 1, ..., |\mathcal{S}| - 1\}$. There are two straightforward ways to store $\mathcal{X} \subseteq \mathcal{S}$ (we will look at more advanced structures later).

- Use some dictionary structure or list of elements; this requires $\mathcal{O}(|\mathcal{X}|)$ storage. If we simply use a null–terminated array, then $|\mathcal{X}| + 1$ integers are required.

- Use an array $\mathbf{x}$ where

$$\mathbf{x}[i] = \begin{cases} 1 & \text{iff } i \in \mathcal{X} \\ 0 & \text{otherwise} \end{cases}$$

  Using a straightforward implementation (for example, an array of type `bool` or `char`), this requires $|\mathcal{S}|$ bytes. However, with a more sophisticated implementation (using bitwise operators) we can "pack" the array and reduce the requirement to $|\mathcal{S}|$ bits.

[2]If states have meaningful names, we should use a data structure that efficiently maps $\mathcal{S}$ into $\{0, 1, ..., |\mathcal{S}| - 1\}$.

For model checking, the sets $\mathcal{X}$ can be quite large (close or equal to $\mathcal{S}$ in size), in part because set complementation is a common operation. A vector of bits is more efficient when

$$|\mathcal{S}| < |\mathcal{X}| \cdot (\text{number of bits for an integer})$$

which happens frequently.

3.5 Matrix descriptions

Let $\mathbf{E} \in \{0,1\}^{\mathcal{S}\times\mathcal{S}}$ be the adjacency matrix for a Kripke structure, where $\mathbf{E}[i,j] = 1$ iff $(i,j) \in \mathcal{R}$. Let $\mathbf{x}, \mathbf{y} \in \{0,1\}^{\mathcal{S}}$ be vectors of bits representing $\mathcal{X}$ and $\mathcal{Y}$, subsets of $\mathcal{S}$, where $\mathbf{x}[i] = 1$ iff $i \in \mathcal{X}$, and $\mathbf{y}[j] = 1$ iff $j \in \mathcal{Y}$. Then, we have

$$\begin{aligned} \mathcal{Y} \;=\; \mathit{PreImage}(\mathcal{X},\mathcal{R}) &= \{s : \exists s' \in \mathcal{X}, (s,s') \in \mathcal{R}\} \\ \mathcal{Y} &= \{i : \exists j, \mathbf{E}[i,j]\mathbf{x}[j] \neq 0\} \\ \mathbf{y}[i] &= \begin{cases} 1 & \text{if } \sum_j \mathbf{E}[i,j]\mathbf{x}[j] > 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

If we treat multiplication as conjunction and addition as disjunction, and use 0 for `ff` and 1 for `tt`, then we obtain

$$\begin{aligned} \mathbf{y}[i] &= \begin{cases} 1 & \text{if } \sum_j \mathbf{E}[i,j]\mathbf{x}[j] > 0 \\ 0 & \text{otherwise} \end{cases} \\ \mathbf{y}[i] &= \mathbf{E}[i,\bullet] \cdot \mathbf{x} \end{aligned}$$

where $\mathbf{E}[i,\bullet]$ is the vector corresponding to row i of $\mathbf{E}$, and "$\cdot$" denotes vector dot product (using conjunction and disjunction, instead of multiplication and addition). But this gives us

$$\mathbf{y} = \mathbf{E} \cdot \mathbf{x} \tag{3.1}$$

as a concise way to describe the pre-image operation.

Similarly, for post-image, we have

$$\begin{aligned} \mathcal{Y} \;=\; \mathit{PostImage}(\mathcal{X},\mathcal{R}) &= \{s' : \exists s \in \mathcal{X}, (s,s') \in \mathcal{R}\} \\ \mathcal{Y} &= \{j : \exists i, \mathbf{x}[i]\mathbf{E}[i,j] \neq 0\} \\ \mathbf{y}[j] &= \begin{cases} 1 & \text{if } \sum_i \mathbf{x}[i]\mathbf{E}[i,j] > 0 \\ 0 & \text{otherwise} \end{cases} \\ \mathbf{y}[j] &= \mathbf{x} \cdot \mathbf{E}[\bullet,j] \\ \mathbf{y} &= \mathbf{x} \cdot \mathbf{E} \end{aligned} \tag{3.2}$$

where $\mathbf{E}[\bullet,j]$ is the vector corresponding to column j of $\mathbf{E}$.

Example 3.10 For the CD player from Example 3.1, compute the pre-image of $\mathcal{X} = \{s_0, s_1\}$.

Set $\mathcal{X}$ corresponds to vector $\mathbf{x} = [1,1,0,0]$. Using the matrix–vector multiplication operation,

we have

$$\begin{aligned}\mathbf{y} &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \\ &= 1 \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + 1 \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} + 0 \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + 0 \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}\end{aligned}$$

Therefore, the pre-image of $\mathcal{X}$ is $\{s_0, s_1, s_2, s_3\}$.

Example 3.11 For the CD player from Example 3.1, compute the post-image of $\mathcal{X} = \{s_0, s_1\}$.

Using the vector–matrix multiplication operation, we have

$$\begin{aligned}\mathbf{y} &= [1,1,0,0] \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \\ &= 1 \cdot [1,1,0,0] + 1 \cdot [1,0,1,0] + 0 \cdot [0,1,1,1] + 0 \cdot [0,1,1,1] \\ &= [1,1,1,0]\end{aligned}$$

Therefore, the post-image of $\mathcal{X}$ is $\{s_0, s_1, s_2\}$.

Chapter 4

Computation Tree Logic

Computation Tree (Temporal) Logic (CTL) is also known as Branching Time Temporal Logic. It is a popular logic for expressing properties, partly because it is powerful enough to be useful, but weak enough that its model checking algorithms are simple to implement. We will discuss CTL as applied to Kripke structures; later we will use different types of models.

4.1 CTL syntax

CTL formulas are *state formulas*, meaning we can define (and determine) whether they hold or not for each state. A state formula ϕ in CTL has the form:

$$\begin{aligned}\phi \quad ::= \quad & \mathtt{tt} \mid \mathtt{ff} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \\ & \mathsf{AX}\,\phi \mid \mathsf{EX}\,\phi \mid \mathsf{AF}\,\phi \mid \mathsf{EF}\,\phi \mid \mathsf{AG}\,\phi \mid \mathsf{EG}\,\phi \mid \mathsf{A}\ (\phi\ \mathsf{U}\ \phi) \mid \mathsf{E}\ (\phi\ \mathsf{U}\ \phi)\end{aligned}$$

where $a \in \mathcal{A}$ is an *atomic proposition*. Operators A and E are *path quantifiers*:

- A: "for all paths",
- E: "for at least one path (there exists a path)".

Operators X, F, G, and U are *temporal operators*:

- X: "in the ne**X**t state",
- F: "in a **F**uture state" (existential),
- G: "in all states (**G**lobally)" (universal),
- U: "**U**ntil".

Note that, according to CTL syntax, the path quantifiers and temporal operators must always appear in pairs. For example, $\mathsf{AX}\,a$ is a valid CTL formula, but $\mathsf{AGX}\,a$ and $\mathsf{AFGX}\,a$ are not valid CTL formulas. $\mathsf{AX}\,a$ means "for all paths, the next state satisfies atomic proposition a".

4.2 CTL semantics

We must give the rules for when states in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ satisfy a CTL state formula ϕ. If a state $s \in \mathcal{S}$ satisfies ϕ, we write

$$M, s \models \phi$$

although often the model M is omitted (if there is only one model, for instance). We write

$$M, s \not\models \phi$$

if state s does not satisfy ϕ. We give the rules for each type of formula in the syntax:

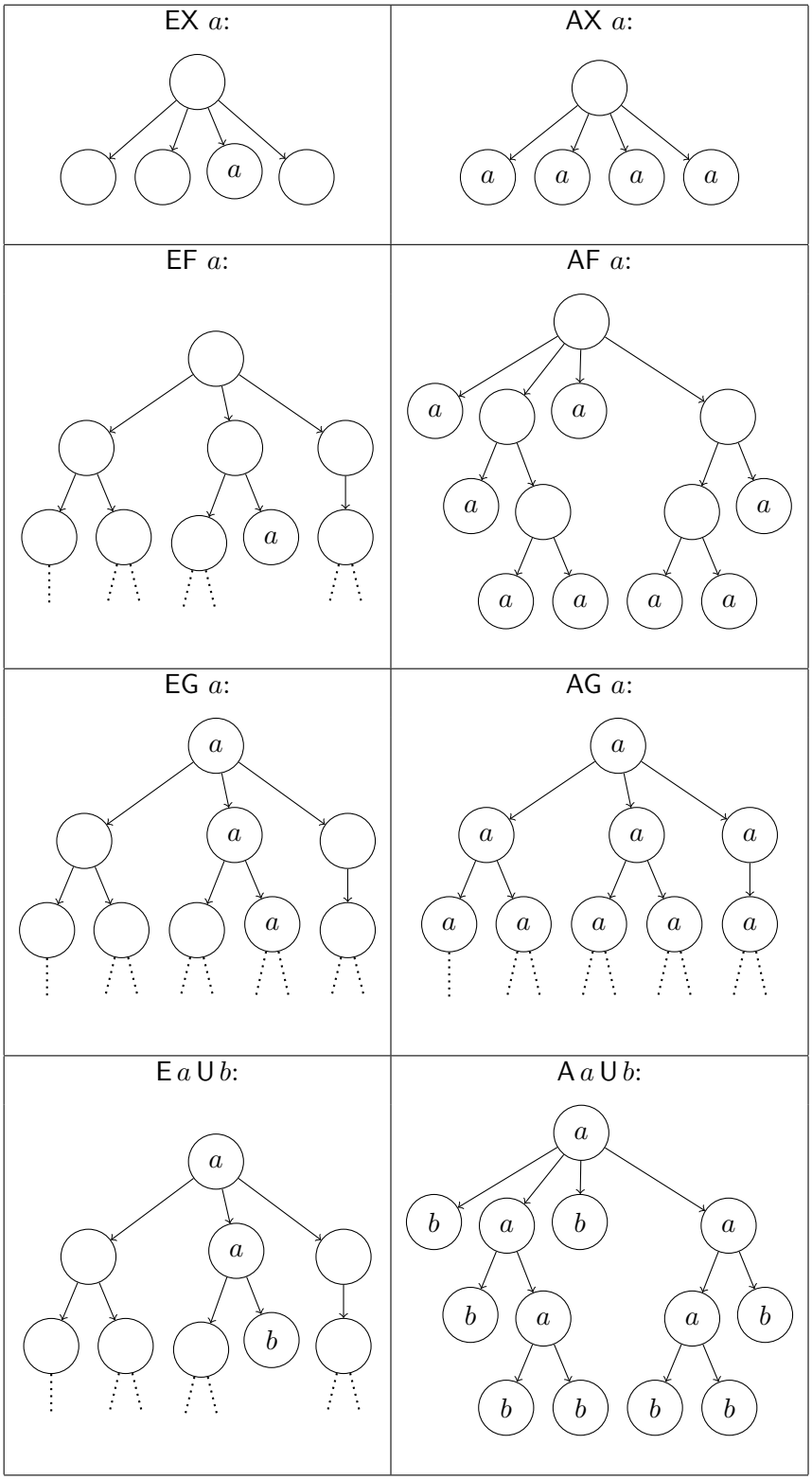
1. $M, s \models \texttt{tt}$, for all $s \in \mathcal{S}$
2. $M, s \not\models \texttt{ff}$, for all $s \in \mathcal{S}$
3. $M, s \models a$, if and only if $a \in \mathcal{L}(s)$
4. $M, s \models \neg\phi$, if and only if $M, s \not\models \phi$
5. $M, s \models \phi_1 \wedge \phi_2$, if and only if $M, s \models \phi_1$ and $M, s \models \phi_2$
6. $M, s \models \phi_1 \vee \phi_2$, if and only if $M, s \models \phi_1$ or $M, s \models \phi_2$
7. $M, s \models \mathsf{AX}\,\phi$, if and only if, $\forall (p_0, p_1, ...) \in Paths_M(s)$, $M, p_1 \models \phi$.
8. $M, s \models \mathsf{EX}\,\phi$, if and only if, $\exists (p_0, p_1, ...) \in Paths_M(s)$ s.t. $M, p_1 \models \phi$.
9. $M, s \models \mathsf{AF}\,\phi$, if and only if, $\forall (p_0, p_1, p_2, ...) \in Paths_M(s)$, there exists an $i \geq 0$ s.t. $M, p_i \models \phi$.
10. $M, s \models \mathsf{EF}\,\phi$, if and only if, $\exists (p_0, p_1, p_2, ...) \in Paths_M(s)$ and an $i \geq 0$ s.t. $M, p_i \models \phi$.
11. $M, s \models \mathsf{AG}\,\phi$, if and only if, $\forall (p_0, p_1, p_2, ...) \in Paths_M(s)$, we have $M, p_i \models \phi$ for all $i \geq 0$.
12. $M, s \models \mathsf{EG}\,\phi$, if and only if, $\exists (p_0, p_1, p_2, ...) \in Paths_M(s)$ s.t. $M, p_i \models \phi$ for all $i \geq 0$.
13. $M, s \models \mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2$, if and only if, $\forall (p_0, p_1, p_2, ...) \in Paths_M(s)$, there exists an $i \geq 0$ s.t.
    (a) $M, p_i \models \phi_2$, and
    (b) $M, p_j \models \phi_1$, for all $0 \leq j < i$
14. $M, s \models \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$, if and only if, $\exists (p_0, p_1, p_2, ...) \in Paths_M(s)$ and an $i \geq 0$ s.t.
    (a) $M, p_i \models \phi_2$, and
    (b) $M, p_j \models \phi_1$, for all $0 \leq j < i$

Finally, we say that the *model* satisfies a formula if *all* of its starting states satisfy the formula:

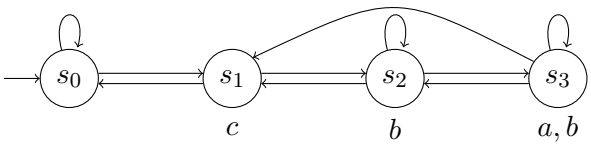
$$M \models \phi \quad \Leftrightarrow \quad \forall s \in \mathcal{S}_0,\ M, s \models \phi$$

where $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ is the Kripke structure.

Example 4.1 What might a "satisfying" computation tree look like for each CTL formula?



Example 4.2 Recall the CD player Kripke structure from Example 3.1:



For this model, does state s_2 satisfy:

1. $\mathsf{EX}\, b$?
2. $\mathsf{AX}\, b$?
3. $\mathsf{EF}\, (\neg c \wedge \neg b)$?
4. $\mathsf{AF}\, c$?
5. $\mathsf{E}\, a\, \mathsf{U}\, b$?
6. $\mathsf{E}\, b\, \mathsf{U}\, c$?
7. $\mathsf{A}\, b\, \mathsf{U}\, c$?

Solutions

1. $\mathsf{EX}\, b$: Yes, because there exists a path of the form $(s_2, s_3, ...)$ and $s_3 \models b$.
2. $\mathsf{AX}\, b$: No, because there is some path that starts with s_2, where b does not hold in the next state. Specifically, a path of the form $(s_2, s_1, ...)$ has $s_1 \not\models b$.
3. $\mathsf{EF}\, (\neg c \wedge \neg b)$: Yes, because using path $(p_0 = s_2, p_1 = s_1, p_2 = s_0, ...)$ and $i = 2$, we have $p_i \models (\neg c \wedge \neg b)$.
4. $\mathsf{AF}\, c$: No. Consider the path $(p_0 = s_2, p_1 = s_2, p_2 = s_2, ...)$. For this path, there does not exist a j s.t. that $p_j \models c$.
5. $\mathsf{E}\, a\, \mathsf{U}\, b$: Yes. Using path $(p_0 = s_2, ...)$ and $i = 0$, we have
   - $p_0 \models b$, and
   - $p_j \models a$, for all $0 \leq j < 0$.
6. $\mathsf{E}\, b\, \mathsf{U}\, c$: Yes. Using path $(p_0 = s_2, p_1 = s_1, ...)$ and $i = 1$, we have
   - $p_1 \models c$, and
   - $p_j \models b$, for all $0 \leq j < 1$.
7. $\mathsf{A}\, b\, \mathsf{U}\, c$: this looks promising, because any path that starts in s_2 and ends up in s_1 will satisfy b until s_1 is reached. However, consider the path $(p_0 = s_2, p_1 = s_2, p_2 = s_2, ...)$. There is no such i where
   - $p_i \models c$, and
   - $p_j \models b$, for all $0 \leq j < i$.

   The second condition holds for any i, but not the first.

4.3 Translating English to CTL

English is not a formal language, so it does not make sense to write an algorithm to convert from English to CTL. However, there are common English patterns that can be mapped to CTL. Here are several useful examples.

1. "The system never reaches a deadlocked state"

   $$\mathsf{AG}\ \neg \mathit{deadlocked}$$

   Technically, the above property is "the system always remains in a non-deadlocked state".

2. "Is it possible to reach a state where condition C holds?"

   $$\mathsf{EF}\ C$$

   Putting the first two together, we might have...

3. "Is it possible to reach a state from which the system never deadlocks?"

   $$\mathsf{EF}\ \mathsf{AG}\ \neg \mathit{deadlocked}$$

   Putting the first two together in the opposite order, we might have...

4. "It is always possible to reach a deadlocked state."

   $$\mathsf{AG}\ \mathsf{EF}\ \mathit{deadlocked}$$

5. "The system will eventually reach a deadlocked state, and remain deadlocked."

   $$\mathsf{AF}\ \mathsf{AG}\ \mathit{deadlocked}$$

6. "Condition C holds infinitely often."

   $$\mathsf{AG}\ \mathsf{AF}\ C$$

7. "Whenever we reach a state where condition B holds, we eventually reach a state where condition C holds."

   $$\mathsf{AG}\ (B \rightarrow \mathsf{AF}\ C)$$

   Note that we effectively rewrote the original statement as "For every state, if condition B holds, then we will eventually reach a state where condition C holds."

8. "Once condition B holds, it holds until condition C holds (and C must eventually hold)."

   $$\mathsf{AG}\ (B \rightarrow \mathsf{A}\, B\, \mathsf{U}\, C)$$

9. "Whenever condition B holds, condition C holds after 2 or more steps."

   $$\mathsf{AG}\ (B \rightarrow \mathsf{AX}\ \mathsf{AX}\ \mathsf{AF}\ C)$$

   Note that this property does not address whether C holds or not in the first two steps.

4.3.1 Safety properties

A *safety* property is one that specifies that some undesired behavior never happens. For example, in a system of traffic lights, we would like to be sure that lights are never simultaneously green for northbound and westbound traffic. Safety properties can be easily expressed in CTL.

Example 4.3 The property

$$\neg \mathsf{EF}(\mathit{north\_green} \ \wedge \ \mathit{west\_green})$$

says that, it is not possible to eventually reach a state where the northbound and westbound lights are both green.

A fundamental characteristic of any safety property ϕ is that, if it does not hold for a state s, we can show that this is the case using a finite path π starting at s.

4.3.2 Liveness properties

A *liveness* property is one that specifies that some desired behavior eventually happens. For example, in a system of traffic lights, we would like to be sure that the northbound traffic will eventually get a green light. Furthermore, we might like this to always be the case. Liveness properties of various strengths can be expressed in CTL.

Example 4.4 The property

$$\mathsf{AG}\ \mathsf{AF}\ \mathit{north\_green}$$

says that, from every state, we will always eventually reach a state where the northbound light is green.

Example 4.5 The property

$$\mathsf{AG}\ \mathsf{EF}\ \mathit{north\_green}$$

says that, from every state, it is always *possible* to reach a state where the northbound light is green.

A fundamental characteristic of any liveness property ϕ is that a finite path π starting at state s cannot be used to show that ϕ does not hold in s.

4.4 Equivalences

As with propositional logic, some properties may be expressed in several different ways in CTL. Given a Kripke structure M and a property to express, sometimes the structure of M itself introduces equivalences. In this section, we discuss formulas that are equivalent in CTL, *for any Kripke structure*. We can manipulate CTL formulas in the same way that we did propositional logic formulas. To prove that formula ϕ_1 is equivalent to formula ϕ_2, we must show that, for any Kripke structure M and any state s, we have $M, s \models \phi_1$ if and only if $M, s \models \phi_2$. Conversely, to show that formulas ϕ_1 and ϕ_2 are *not* equivalent, it is sufficient to find a Kripke structure M and a state s such that $M, s \models \phi_1$ and $M, s \not\models \phi_2$, or vice versa.

Property 4.1

$$\neg \mathsf{EX}\, \phi \quad \equiv \quad \mathsf{AX}\, \neg \phi$$

Proof:

$$\begin{aligned} M, s \models \neg \mathsf{EX}\, \phi \quad &\Leftrightarrow \quad M, s \not\models \mathsf{EX}\, \phi \\ &\Leftrightarrow \quad \nexists\, \pi = (p_0, p_1, ...) \in Paths(s) \ \text{ s.t. } \ M, p_1 \models \phi \\ &\Leftrightarrow \quad \forall \pi = (p_0, p_1, ...) \in Paths(s), \ M, p_1 \not\models \phi \\ &\Leftrightarrow \quad \forall \pi = (p_0, p_1, ...) \in Paths(s), \ M, p_1 \models \neg \phi \\ &\Leftrightarrow \quad M, s \models \mathsf{AX}\, \neg \phi \end{aligned}$$

Property 4.2

$$\neg \mathsf{EF}\, \phi \quad \equiv \quad \mathsf{AG}\, \neg \phi$$

Proof:

$$\begin{aligned} M, s \models \neg \mathsf{EF}\, \phi \quad &\Leftrightarrow \quad M, s \not\models \mathsf{EF}\, \phi \\ &\Leftrightarrow \quad \nexists\, i, \pi = (p_0, p_1, ...) \in Paths(s) \ \text{ s.t. } \ M, p_i \models \phi \\ &\Leftrightarrow \quad \forall \pi = (p_0, p_1, ...) \in Paths(s), \forall i, \quad M, p_i \not\models \phi \\ &\Leftrightarrow \quad \forall \pi = (p_0, p_1, ...) \in Paths(s), \forall i, \quad M, p_i \models \neg \phi \\ &\Leftrightarrow \quad M, s \models \mathsf{AG}\, \neg \phi \end{aligned}$$

Property 4.3

$$\neg \mathsf{EG}\, \phi \quad \equiv \quad \mathsf{AF}\, \neg \phi$$

Proof:

$$\begin{aligned} M, s \models \neg \mathsf{EG}\, \phi \quad &\Leftrightarrow \quad M, s \not\models \mathsf{EG}\, \phi \\ &\Leftrightarrow \quad \nexists\, \pi = (p_0, p_1, ...) \in Paths(s) \ \text{ s.t. } \ \forall i, \ M, p_i \models \phi \\ &\Leftrightarrow \quad \forall \pi = (p_0, p_1, ...) \in Paths(s), \ \exists i, \ M, p_i \not\models \phi \\ &\Leftrightarrow \quad \forall \pi = (p_0, p_1, ...) \in Paths(s), \ \exists i, \ M, p_i \models \neg \phi \\ &\Leftrightarrow \quad \mathsf{AF}\, \neg \phi \end{aligned}$$

Property 4.4

$$\mathsf{E}\, \mathtt{tt}\, \mathsf{U}\, \phi \quad \equiv \quad \mathsf{EF}\, \phi$$

Property 4.5

$$\mathsf{A}\, \mathtt{tt}\, \mathsf{U}\, \phi \quad \equiv \quad \mathsf{AF}\, \phi$$

Property 4.6

$$\neg \mathsf{A}\, \phi_1\, \mathsf{U}\, \phi_2 \quad \equiv \quad \mathsf{E}(\, \neg \phi_2\, \mathsf{U}\, (\neg \phi_1 \wedge \neg \phi_2)\,) \quad \vee \quad \mathsf{EG}\, \neg \phi_2$$

Property 4.7

$$\phi \vee \mathsf{EX}\, \mathsf{EF}\, \phi \quad \equiv \quad \mathsf{EF}\, \phi$$

Proof:

$$\begin{aligned}
M, s \models \phi \vee \mathsf{EX}\,\mathsf{EF}\,\phi \quad &\Leftrightarrow \quad M, s \models \phi \quad \vee \quad M, s \models \mathsf{EX}\,\mathsf{EF}\,\phi \\
&\Leftrightarrow \quad M, s \models \phi \vee \exists \pi = (p_0, p_1, ...) \in Paths(s),\ M, p_1 \models \mathsf{EF}\,\phi \\
&\Leftrightarrow \quad M, s \models \phi \vee \exists \pi = (p_0, p_1, ...) \in Paths(s), \\
&\qquad \exists i, \pi' = (p'_0, p'_1, p'_2, ...) \in Paths(p_1),\ M, p'_i \models \phi \\
&\Leftrightarrow \quad M, s \models \phi \ \vee \ \exists i, \pi = (p_0, p'_0, p'_1, ...) \in Paths(s),\ M, p'_i \models \phi \\
&\Leftrightarrow \quad \exists i, \pi = (p_0, p_1, ...) \in Paths(s),\ M, p_i \models \phi \\
&\Leftrightarrow \quad M, s \models \mathsf{EF}\,\phi
\end{aligned}$$

Property 4.8

$$\phi \vee \mathsf{AX}\,\mathsf{AF}\,\phi \quad \equiv \quad \mathsf{AF}\,\phi$$

Property 4.9

$$\phi \wedge \mathsf{EX}\,\mathsf{EG}\,\phi \quad \equiv \quad \mathsf{EG}\,\phi$$

Property 4.10

$$\phi \wedge \mathsf{AX}\,\mathsf{AG}\,\phi \quad \equiv \quad \mathsf{AG}\,\phi$$

Property 4.11

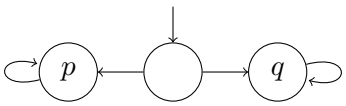
$$\phi_2 \vee (\phi_1 \wedge \mathsf{EX}\,\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2) \quad \equiv \quad \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$$

Property 4.12

$$\phi_2 \vee (\phi_1 \wedge \mathsf{AX}\,\mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2) \quad \equiv \quad \mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2$$

Example 4.6 Is the formula $\mathsf{EX}(p \wedge q)$ equivalent to $(\mathsf{EX}\,p) \wedge (\mathsf{EX}\,q)$?

Solution: It is easy to show that $\mathsf{EX}(p \wedge q) \ \rightarrow \ (\mathsf{EX}\,p) \wedge (\mathsf{EX}\,q)$. However, the reverse is not true. Consider the following Kripke structure:



Clearly, both $\mathsf{EX}\,p$ and $\mathsf{EX}\,q$ hold in the initial state, but $\mathsf{EX}(p \wedge q)$ does not.

4.4.1 Adequate sets of operators for CTL

From the above properties, and from the adequate sets for propositional logic, we see that the following operations are adequate to express all CTL formulas:

1. $\neg$
2. $\wedge$
3. AX or EX
4. EG or AF
5. EU (meaning, $\mathsf{E}\phi_1\mathsf{U}\phi_2$)

4.5 Algorithms for CTL operators

To automatically determine the set of states that satisfy an arbitrary CTL state formula ϕ, we need algorithms for all of the CTL operators. These are sometimes called *labeling* algorithms. For example, if we know which states are labeled with ϕ_1, and which states are labeled with ϕ_2, then we need an algorithm to label states satisfying $\phi = \phi_1 \wedge \phi_2$. We will discuss algorithms only for some operators (namely, the adequate ones: $\neg, \wedge, \mathsf{AX}, \mathsf{EX}, \mathsf{AF}, \mathsf{EG}, \mathsf{EU}$).

4.5.1 Labeling for $\neg$

There is a trivial algorithm for $\neg$: given the labeling for ϕ_1, we can determine the labeling for $\phi = \neg\phi_1$ as follows:

> For each state s, label s with ϕ if and only if s is not labeled with ϕ_1.

Complexity is $\mathcal{O}(|\mathcal{S}|)$.

4.5.2 Labeling for $\wedge$

There is also a simple algorithm for labeling $\phi = \phi_1 \wedge \phi_2$:

> For each state s, label s with ϕ if and only if s is labeled with both ϕ_1 and ϕ_2.

Complexity is $\mathcal{O}(|\mathcal{S}|)$.

4.5.3 Labeling for AX

We can label $\phi = \mathsf{AX}\,\phi_1$ using the following algorithm:

> For each state s, label s with ϕ if and only if every successor of s (i.e., all states s' with $(s, s') \in \mathcal{R}$) is labeled with ϕ_1.

Complexity is $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$.

4.5.4 Labeling for EX

We can label $\phi = \mathsf{EX}\,\phi_1$ using an algorithm similar to the one for AX:

> For each state s, label s with ϕ if and only if some successor of s is labeled with ϕ_1.

Using preimage

Note that this algorithm is exactly the pre-image operation. Thus, if bit vector $\mathbf{x}$ has entries where $\mathbf{x}[i]$ is one if and only if state i is labeled with ϕ_1, then the bit vector $\mathbf{y}$ given by

$$\mathbf{y} = \mathbf{E} \cdot \mathbf{x},$$

where $\mathbf{E}$ is the adjacency matrix corresponding to $\mathcal{R}$, has entries one for states labeled with ϕ. Complexity is $\mathcal{O}(|\mathcal{S}|+|\mathcal{R}|)$; to obtain this complexity using $\mathbf{E}\cdot\mathbf{x}$, a suitable matrix–vector multiplication algorithm must be used (one that exploits the fact that $\mathbf{E}$ is sparse).

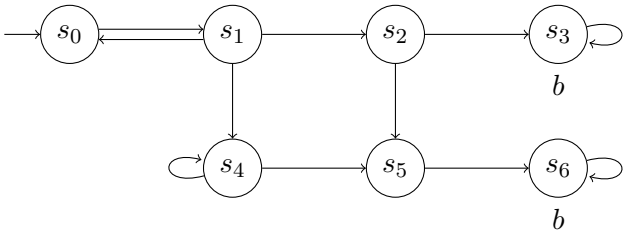
4.5.5 Labeling for AF

We can label $\phi = \mathsf{AF}\,\phi_1$ using an algorithm based on Property 4.8:

1. Any state labeled with ϕ_1 is also labeled with ϕ.
2. If all successors of state s are labeled with ϕ, then label s with ϕ
3. Repeat step (2) until no more changes are possible.

Note that the algorithm is guaranteed to eventually terminate, since at most $|\mathcal{S}|$ states can be labeled. This algorithm, if implemented cleverly, has a complexity of $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$.

Example 4.7 Compute the labeling for $\phi = \mathsf{AF}\,b$ for the following Kripke structure:



Solution: Using the algorithm, in step (1) we label s_3 and s_6 with ϕ. In step (2), we can label s_5 with ϕ. There was a change, so we repeat. In step (2), we can label s_2 with ϕ. There was a change, so we repeat. But no more states have *all* their successors labeled with ϕ, so the algorithm terminates. We therefore have that the states $\{s_2, s_3, s_5, s_6\}$ satisfy $\mathsf{AF}\,b$.
Verify: States s_3 and s_6 satisfy b, so they trivially satisfy $\mathsf{AF}\,b$. State s_5 has only one possible path, and state s_2 has only two possible paths, both of which eventually reach a state satisfying b. But what about the other states? State s_4 has a self loop, thus there exists a path, namely $(s_4, s_4, s_4, ...)$ that never reaches a state satisfying b. Similarly, states s_0 and s_1 can loop forever and never reach a state satisfying b.

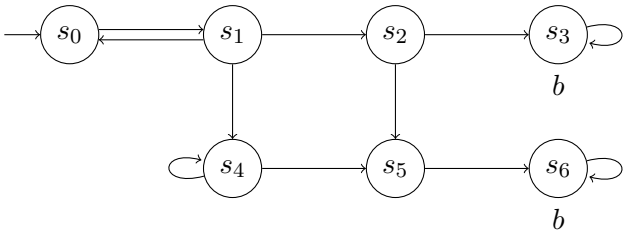
4.5.6 Labeling for EG — first approach: iterative algorithm

We can label $\phi = \mathsf{EG}\,\phi_1$ using an algorithm based on Property 4.9:

1. Any state labeled with ϕ_1 is labeled with ϕ.
2. If no successors of state s are labeled with ϕ, then remove the ϕ label from state s.
3. Repeat step (2) until no more changes are possible.

Note that the algorithm will eventually terminate, since in step 2 we are only removing labels from states, and there are only finitely many states. Also, note that this is essentially the opposite of the algorithm for AF.

Example 4.8 Compute the labeling for $\phi = \mathsf{EG}\,\neg b$ for the following Kripke structure:



Solution: Using the algorithm, in step (1) we label s_0, s_1, s_2, s_4, and s_5 with ϕ. In step (2), we remove label ϕ from s_5, since none of its successors are labeled with ϕ. There was a change, so we repeat. In step (2), we remove label ϕ from s_2. There was a change, so we repeat. But the remaining states all have at least one successor labeled with ϕ, so the algorithm terminates. We therefore have that the states $\{s_0, s_1, s_4\}$ satisfy $\mathsf{EG}\,\neg b$.
Verify: It is easy to see that, starting from states s_0, s_1, and s_4, it is possible to remain in states where b is not satisfied. This is not possible from states s_2, s_3, s_5, and s_6.

Using preimage

We can rewrite the above algorithm in terms of the pre-image operation, as follows. Assume we have a bitvector $\mathbf{x}$ that encodes the states satisfying ϕ_1. Then, we can build the bitvector $\mathbf{y}$ that encodes the states satisfying $\phi = \mathsf{EG}\,\phi_1$ using the iteration

$$\begin{aligned} \mathbf{y}_0 &= \mathbf{x} \\ \mathbf{y}_{n+1} &= \mathbf{y}_n \wedge (\mathbf{E} \cdot \mathbf{y}_n) \end{aligned}$$

and stopping when $\mathbf{y}_{n+1} = \mathbf{y}_n$, at which point we can return $\mathbf{y} = \mathbf{y}_n$. It can be shown that the iteration

$$\begin{aligned} \mathbf{y}_0 &= \mathbf{x} \\ \mathbf{y}_{n+1} &= \mathbf{x} \wedge (\mathbf{E} \cdot \mathbf{y}_n) \end{aligned}$$

produces exactly the same sequence of bitvectors. (Hint: $\mathbf{y}_n$ is the set of all starting states from which it is possible to find a path where the first n states on the path satisfy ϕ_1.)

4.5.7 Labeling for EG — second approach: strongly connected components

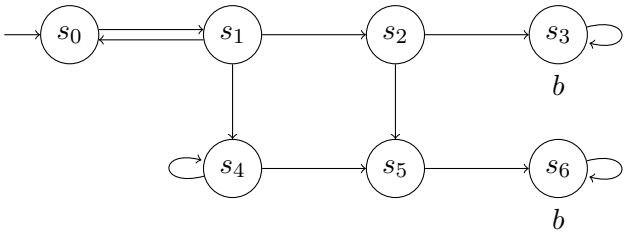
Another algorithm for labeling $\phi = \mathsf{EG}\,\phi_1$ is based on the observation that the only way an infinitely–long path satisfying ϕ_1 can occur is if there is a cycle of states satisfying ϕ_1. The presence of such cycles is determined using strongly–connected components (SCCs). Recall that a SCC is a set of states such that, for any pair of states i and j in the SCC, there is a path from state i to state j.

1. Create a graph M' by removing all states that do not satisfy ϕ_1, and any edges associated with those states. (Note: this might not be a Kripke structure, because it is possible for a state to have no outgoing edges.)
2. Determine the SCCs for M'.
3. For each SCC,
   - if the SCC contains more than one state, then label all states in the SCC with ϕ
   - if the SCC contains only one state, and the state has an edge to itself[1], then label it with ϕ.
4. Any state that can reach (in M') a state labeled with ϕ should also be labeled with ϕ.

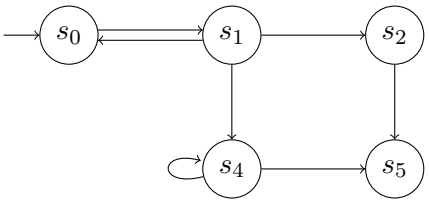
There is an algorithm to determine all SCCs for a graph, with complexity $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$. Thus, this algorithm has complexity $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$.

[1]This check is necessary because, the SCC algorithm will put every state into some SCC.

Example 4.9 Compute the labeling for $\phi = \mathsf{EG}\,\neg b$ for the following Kripke structure:



Solution: Using the algorithm, in step (1) we build a new graph containing only the states satisfying $\neg b$; this gives us:



This graph has the following SCCs: $\{s_0, s_1\}, \{s_2\}, \{s_4\}, \{s_5\}$. Following step (3) of the algorithm, we label $\{s_0, s_1\}$ and $\{s_4\}$ with ϕ. No new states are labeled in step (4); therefore, the states $\{s_0, s_1, s_4\}$ satisfy $\mathsf{EG}\,\neg b$.
Verify: This is the same set as the previous example.

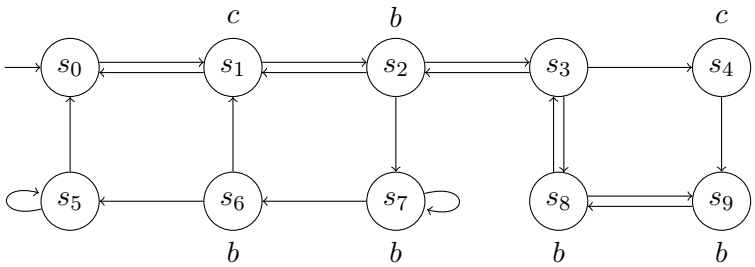
4.5.8 Labeling for EU

We can label $\phi = \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$ based on Property 4.11:

1. Any state labeled with ϕ_2 is labeled with ϕ.
2. If state s is labeled with ϕ_1, and some successor of s is labeled with ϕ, then label s with ϕ.
3. Repeat step (2) until no more changes are possible.

Since $\mathcal{S}$ is finite and we never remove labels from states, the algorithm will eventually terminate. This algorithm, if implemented cleverly (using a single graph traversal), has complexity $\mathcal{O}(|\mathcal{S}|+|\mathcal{R}|)$.

Example 4.10 Compute the labeling for $\phi = \mathsf{E}\,b\,\mathsf{U}\,c$ for the following Kripke structure:



Solution: In step (1) of the algorithm, we label states s_1 and s_4 with ϕ, because they satisfy c. Then, during the first iteration of step (2), we can label states s_2 and s_6 with ϕ, because they satisfy b and have a successor labeled with ϕ. In the next iteration, we can label state s_7 with ϕ. No more changes are possible after that; therefore, the states $\{s_1, s_2, s_4, s_6, s_7\}$ satisfy $\mathsf{E}\,b\,\mathsf{U}\,c$.
Verify: From each of (and only) $\{s_1, s_2, s_4, s_6, s_7\}$, there is a path that

- leads to (or starts at) a state satisfying c, and
- for every state on the path before c is satisfied, b is satisfied.

Using preimage

We can rewrite the above algorithm in terms of the pre-image operation. Suppose the bitvector $\mathbf{b}$ encodes states satisfying b, and $\mathbf{c}$ encodes the states satisfying c. Then, we can build the bitvector $\mathbf{y}$ encoding the states satisfying $\phi = \mathsf{E}\, b\, \mathsf{U}\, c$ using the iteration

$$\begin{aligned} \mathbf{y}_0 &= \mathbf{c} \\ \mathbf{y}_{n+1} &= \mathbf{y}_n \vee ((\mathbf{E} \cdot \mathbf{y}_n) \wedge \mathbf{b}) \end{aligned}$$

and stopping when $\mathbf{y}_{n+1} = \mathbf{y}_n$, at which point we can return $\mathbf{y} = \mathbf{y}_n$.

4.6 Counterexamples and witnesses

Suppose we have a Kripke structure, and we want to verify the property $\mathsf{AG}(\neg \mathit{deadlocked})$, but it turns out the answer is "no". Now what?

For debugging of the model or the underlying system, it would be nice to know "why not". In the case of this property, because we have

$$\neg \mathsf{AG}(\neg \mathit{deadlocked}) \;\equiv\; \mathsf{EF}\ \mathit{deadlocked}$$

we can see that the property does not hold because there exists a path that eventually reaches a *deadlocked* state. Such a path is a called a *counterexample* to the property $\mathsf{AG}(\neg \mathit{deadlocked})$. In general, whenever an "A" property does not hold, then the corresponding "E" property holds and can be used to give a counterexample. Summarizing earlier properties, we have:

$$\begin{array}{lcl} s \not\models \mathsf{AX}\, \phi & \rightarrow & s \models \mathsf{EX}\, \neg\phi \\ s \not\models \mathsf{AF}\, \phi & \rightarrow & s \models \mathsf{EG}\, \neg\phi \\ s \not\models \mathsf{AG}\, \phi & \rightarrow & s \models \mathsf{EF}\, \neg\phi \\ s \not\models \mathsf{A}\, \phi_1\, \mathsf{U}\, \phi_2 & \rightarrow & s \models \mathsf{EG}\, \neg\phi_2 \;\vee\; s \models \mathsf{E}\, (\phi_1 \wedge \neg\phi_2)\, \mathsf{U}\, (\neg\phi_1 \wedge \neg\phi_2) \end{array}$$

What about "E" properties? If $\mathsf{EG}\, \phi$ does not hold, can we give a counterexample? This would require us to show that $\mathsf{AF}\, \neg\phi$ holds, and in general we cannot give a single path to show this. For "E" properties, an example path that satisfies the formula is called a *witness*. Thus, a witness for $\mathsf{EG}\, \phi$ is an example path where ϕ holds in every state, and is also a counterexample for $\mathsf{AF}\, \neg\phi$. We therefore need witness generation algorithms for EX, EG, and EU (the one for EF is a special case).

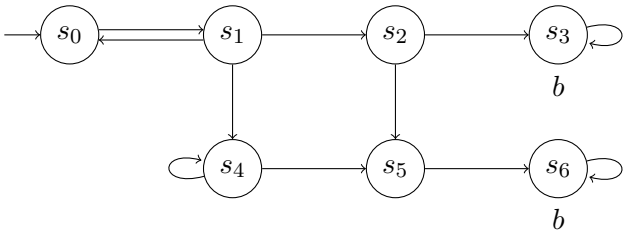
4.6.1 Witnesses for EX

Given a state $s \models \mathsf{EX}\, \phi$, how do we generate a witness for $\mathsf{EX}\, \phi$ that starts in state s? This is trivial: check all successors of s for an $s' \models \phi$; the witness is a path $(s, s', ...)$, where of course we only display the "interesting" portion of the path.

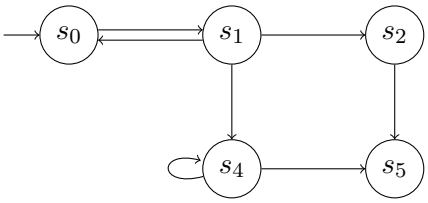
4.6.2 Witnesses for EG

Given a state $s \models \mathsf{EG}\,\phi$, how do we generate a witness for $\mathsf{EG}\,\phi$ that starts in state s? Note that a witness will contain a cycle of states that satisfy ϕ; it suffices to terminate the displayed path with such a cycle (preferably a minimal cycle), rather than displaying an infinitely–long path. The SCC–based algorithm for labeling EG can inspire a witness generation algorithm. Consider the graph M' obtained by removing all states that do not satisfy ϕ, and their incoming and outgoing edges. If s has a self loop in this graph, then trivially $(s, s, ...)$ is a witness. If s belongs to a SCC with at least two states, then from any successor s' of s, there must exist a path in the graph from s' to s. Find and display such a path (ideally, find a *shortest* path). Otherwise, find a path from s to some state s' that either contains a self loop or belongs to a SCC with at least two states. Note that $s' \models \mathsf{EG}\,\phi$. Display the path from s to s', and then display a witness (a cycle) for $\mathsf{EG}\,\phi$ that starts in state s'; the combination of the path from s to s' and the cycle on s' is called a *lasso*.

Example 4.11 For the following Kripke structure, give a witness for $\mathsf{EG}\,\neg b$ starting in state s_0.



Solution: From an earlier example, removing the states that do not satisfy $\neg b$ gives us the graph:



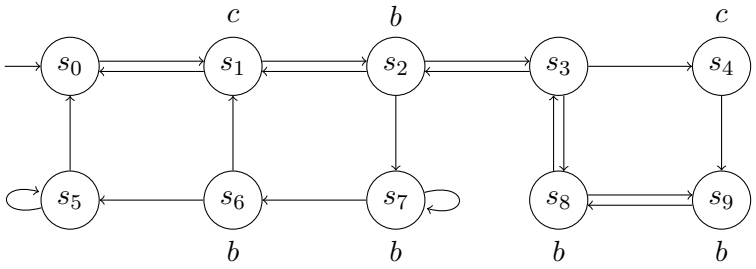
Note that s_0 belongs to the SCC $\{s_0, s_1\}$. Therefore, a witness is s_0, followed by a path in the graph from s_1 (a successor of s_0) to s_0, which turns out to be trivial because there is an edge from s_1 to s_0. This gives us the witness

$$(s_0, s_1, s_0, ...)$$

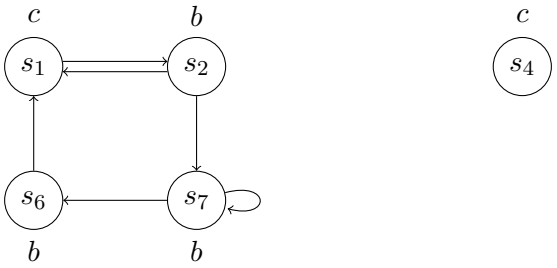
4.6.3 Witnesses for EU

Given a state $s \models \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$, a witness for $\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$ starting from s can be generated by finding a path from s to a state satisfying ϕ_2, along only states satisfying ϕ_1. One way to do this is to find a shortest path from s to a state satisfying ϕ_2 in the graph obtained from the Kripke structure by removing all states (and their incoming and outgoing edges) that do not satisfy $\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$.

Example 4.12 For the following Kripke structure, give a witness for $\mathsf{E}\,b\,\mathsf{U}\,c$ starting in state s_2.



Solution: From an earlier example, we know the set of states satisfying $\mathsf{E}\, b\, \mathsf{U}\, c$ is exactly $\{s_1, s_2, s_4, s_6, s_7\}$; restricting the Kripke structure to these states gives us the graph:



Finding a shortest path from s_2 to a state satisfying c (states s_1 and s_4) will generate the path s_2, s_1; that gives us the witness

$$(s_2, s_1, ...)$$

4.6.4 Nested formulas

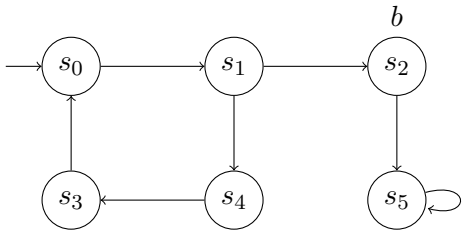
Automatically generating a "complete" witness or counterexample for a nested formula turns out to be impossible in general.

For instance, suppose we generate a witness for a formula $\mathsf{EG}\, \phi$ using the method discussed above. That gives us a lasso where each state satisfies ϕ. But if ϕ is a complex formula, it might not be obvious that each state on the lasso satisfies ϕ. We should then generate witnesses for ϕ, starting from each state on the lasso for $\mathsf{EG}\, \phi$. Unfortunately, this is not always possible: if ϕ contains an "A" formula, then we cannot generate a witness for it.

In general, witnesses can be generated for *ECTL* formulas, that is, CTL formulas containing only (non-negated) E path quantifiers (i.e., negations can only be applied to atomic propositions). Similarly, counterexamples can be generated for *ACTL* formulas (formulas containing only A path quantifiers). This can be done recursively, and the witness (or counterexample) will have a tree-like structure[2]: draw the witness of the outer-most formula as a path, and then for each node in the path, draw a witness for the inner formula. Continue this recursively.

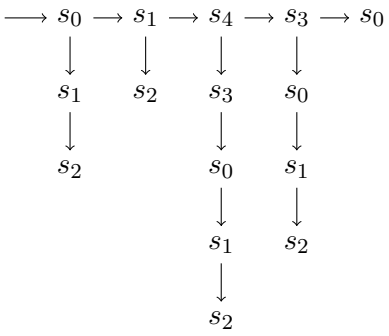
Example 4.13 Consider the Kripke structure:

[2]See "Tree-like counterexamples in model checking", by E. Clarke, S. Jha, Y. Lu, and H. Veith, in Proceedings of LICS (Logic in Computer Science), 2002, pages 19–29.



Generate a witness for the ECTL formula $\mathsf{EG}\,\mathsf{EF}\,b$.

First, note that all states except s_5 satisfy $\mathsf{EF}\,b$. Then, we just need an infinite path starting from s_0 that does not include s_5. We can draw $(s_0, s_1, s_4, s_3, s_0, ...)$, a witness for $\mathsf{EG}\,\mathsf{EF}\,b$ horizontally, and then vertically down from each state, draw a witness for $\mathsf{EF}\,b$ from that state:



Note that we do not need to repeat the witness for $\mathsf{EF}\,b$ for the second occurrence of s_0.

4.7 Fixpoints

In this section, we will discuss some of the theory behind the algorithms for the CTL operators. We consider functions f of the form $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, i.e., functions that take a set of states and return a set of states.

Definition 4.13 A function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ is called *monotonic* if

$$\mathcal{X} \subseteq \mathcal{Y} \quad \to \quad f(\mathcal{X}) \subseteq f(\mathcal{Y}), \qquad \forall \mathcal{X}, \mathcal{Y} \subseteq \mathcal{S}$$

Example 4.14 For $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$, is the function $f(\mathcal{X}) = \mathcal{X} \cup \{s_0\}$ monotonic?
Solution: This example is small enough to enumerate, but let's try a proper proof instead. For any sets $\mathcal{X}, \mathcal{Y}$ with $\mathcal{X} \subseteq \mathcal{Y}$, we must show that $f(\mathcal{X}) \subseteq f(\mathcal{Y})$:

$$f(\mathcal{X}) \;=\; \mathcal{X} \cup \{s_0\} \;\subseteq\; \mathcal{Y} \cup \{s_0\} \;=\; f(\mathcal{Y})$$

Example 4.15 For $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$, the function

$$f(\mathcal{X}) \;=\; \begin{cases} \{s_0\} & \text{If } s_1 \in \mathcal{X} \\ \{s_0, s_1\} & \text{Otherwise} \end{cases}$$

is *not* monotonic, because $\emptyset \subseteq \{s_1\}$ but $f(\emptyset) = \{s_0, s_1\}$ is not a subset of $f(\{s_1\}) = \{s_0\}$.

Property 4.14 The function $f_{\mathcal{R}}(\mathcal{X}) = \mathit{PreImage}(\mathcal{X}, \mathcal{R})$ is monotonic.

Proof: Suppose $\mathcal{X} \subseteq \mathcal{Y}$, and consider $s \in f_{\mathcal{R}}(\mathcal{X})$. Then there must exist some $s' \in \mathcal{X}$ s.t. $(s, s') \in \mathcal{R}$. But $\mathcal{X} \subseteq \mathcal{Y}$, so we must have $s' \in \mathcal{Y}$ also. This implies $s \in f_{\mathcal{R}}(\mathcal{Y})$.

Definition 4.15 For any function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, $\mathcal{X}$ is a *fixpoint* of f if $f(\mathcal{X}) = \mathcal{X}$.

Example 4.16 For the monotonic function $f(\mathcal{X}) = \mathcal{X} \cup \{s_0\}$, there are several fixpoints, including

$$\begin{aligned} f(\{s_0, s_1\}) &= \{s_0, s_1\} \\ f(\mathcal{S}) &= \mathcal{S} \end{aligned}$$

Definition 4.16 For any function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, define f^n, for $n \geq 0$, inductively as

$$\begin{aligned} f^0(\mathcal{X}) &= \mathcal{X} \\ f^{n+1}(\mathcal{X}) &= f(f^n(\mathcal{X})) \end{aligned}$$

Property 4.17 (Tarski–Knaster Theorem) If f is monotonic over $2^{\mathcal{S}}$, where $|\mathcal{S}| = n$, then

1. $f^n(\emptyset)$ is the least fixpoint of f
2. $f^n(\mathcal{S})$ is the greatest fixpoint of f

Proof: Since $\emptyset \subseteq \mathcal{X}$ for any $\mathcal{X}$, we have

$$\begin{aligned} \emptyset &\subseteq f(\emptyset) \\ f(\emptyset) &\subseteq f(f(\emptyset)) \qquad \text{because } f \text{ is monotone} \\ &\vdots \\ f^i(\emptyset) &\subseteq f^{i+1}(\emptyset) \qquad \forall i \end{aligned}$$

Because $\mathcal{S}$ is finite, there exists a $j \geq 0$ such that $f^j(\emptyset) = f^{j+1}(\emptyset)$, and we have

$$f^0(\emptyset) \subset f^1(\emptyset) \subset f^2(\emptyset) \subset \cdots \subset f^j(\emptyset) = f^{j+1}(\emptyset) = f^{j+2}(\emptyset) = \cdots$$

Furthermore, we know $j \leq n$. Therefore, $f(f^n(\emptyset)) = f^n(\emptyset)$, and $f^n(\emptyset)$ is a fixpoint. Now we must show that it is the *least* fixpoint. Consider any fixpoint $\mathcal{X}$. Then, we have

$$\begin{aligned} \emptyset &\subseteq \mathcal{X} \\ f(\emptyset) &\subseteq f(\mathcal{X}) = \mathcal{X} \\ &\vdots \\ f^n(\emptyset) &\subseteq \mathcal{X} \end{aligned}$$

and the proof of part (1) is complete. The proof for part (2) is similar.

Example 4.17 For the monotonic function $f(\mathcal{X}) = \mathcal{X} \cup \{s_0\}$, we have

$$\begin{aligned} f(\emptyset) &= \{s_0\} \\ f(\{s_0\}) &= \{s_0\} \end{aligned}$$

thus the least fixpoint is $\{s_0\}$. Similarly, since $f(\mathcal{S}) = \mathcal{S}$, the greatest fixpoint is $\mathcal{S}$.

4.7.1 Sets satisfying CTL state formulas

Definition 4.18 For any CTL state formula ϕ, let

$$[\![\phi]\!]_M \qquad \text{denote the set of states in } M \text{ satisfying } \phi$$

where the Kripke stucture M may be dropped if it is clear from context.

Using this notation, we can express the labeling algorithms in terms of the sets of states satisfying the formulas. For example, for Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$, we have:

- $[\![\mathtt{tt}]\!]_M \;=\; \mathcal{S}$
- $[\![\mathtt{ff}]\!]_M \;=\; \emptyset$
- $[\![\neg\phi]\!]_M \;=\; \mathcal{S} \setminus [\![\phi]\!]_M$
- $[\![\phi_1 \wedge \phi_2]\!]_M \;=\; [\![\phi_1]\!]_M \cap [\![\phi_2]\!]_M$
- $[\![\phi_1 \vee \phi_2]\!]_M \;=\; [\![\phi_1]\!]_M \cup [\![\phi_2]\!]_M$
- $[\![\mathsf{EX}\,\phi]\!]_M \;=\; \mathit{PreImage}([\![\phi]\!]_M, \mathcal{R})$

Now, define function $A_\mathcal{R} : 2^\mathcal{S} \to 2^\mathcal{S}$ as

$$A_\mathcal{R}(\mathcal{X}) \;=\; \{s : \forall (s, s') \in \mathcal{R}, s' \in \mathcal{X}\}$$

and note that

- $[\![\mathsf{AX}\,\phi]\!]_M \;=\; A_\mathcal{R}([\![\phi]\!]_M)$.

Property 4.19 For any relation $\mathcal{R}$, function $A_\mathcal{R}$ is monotonic.

Proof: Suppose $\mathcal{X} \subseteq \mathcal{Y}$, and consider $s \in A_\mathcal{R}(\mathcal{X})$. Then, we must have $\forall (s, s') \in \mathcal{R}$, $s' \in \mathcal{X}$. Since $\mathcal{X} \subseteq \mathcal{Y}$, we also have $\forall (s, s') \in \mathcal{R}$, $s' \in \mathcal{Y}$. Therefore, we must have $s \in A_\mathcal{R}(\mathcal{Y})$.

4.7.2 Fixpoints and AF

Property 4.20 For any Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ and any state formula ϕ, define $F_\phi(\mathcal{X}) = [\![\phi]\!] \cup A_\mathcal{R}(\mathcal{X})$. Then, $[\![\mathsf{AF}\,\phi]\!]$ is the least fixpoint of F_ϕ.

Proof: First, note that when $\mathcal{X} \subseteq \mathcal{Y}$, we have $[\![\phi]\!] \cup A_\mathcal{R}(\mathcal{X}) \subseteq [\![\phi]\!] \cup A_\mathcal{R}(\mathcal{Y})$ because $A_\mathcal{R}$ is monotonic. Therefore, F_ϕ is monotonic.

Now, using Property 4.8, we have

$$\begin{aligned}
[\![\mathsf{AF}\,\phi]\!] &= [\![\phi \vee \mathsf{AX}\,\mathsf{AF}\,\phi]\!] \\
&= [\![\phi]\!] \cup [\![\mathsf{AX}\,\mathsf{AF}\,\phi]\!] \\
&= [\![\phi]\!] \cup A_\mathcal{R}([\![\mathsf{AF}\,\phi]\!]) \\
&= F_\phi([\![\mathsf{AF}\,\phi]\!])
\end{aligned}$$

thus $[\![\mathsf{AF}\,\phi]\!]$ is a fixpoint of F_ϕ.

Finally, we show that it is the *least* fixpoint. Let $\mathcal{X}$ be some fixpoint, and we will prove by

contradiction that $[\![\mathsf{AF}\,\phi]\!] \subseteq \mathcal{X}$. Consider some $s \in [\![\mathsf{AF}\,\phi]\!] = [\![\phi]\!] \cup A_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!])$, and suppose $s \notin \mathcal{X}$. Because $\mathcal{X} = F_\phi(\mathcal{X}) = [\![\phi]\!] \cup A_{\mathcal{R}}(\mathcal{X})$, we have

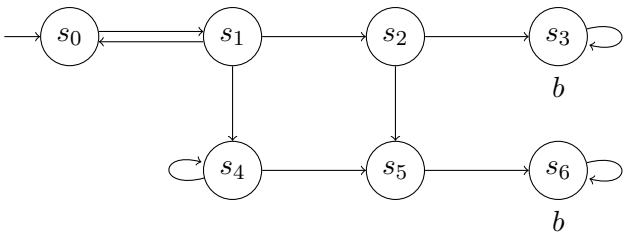
$$s \in [\![\phi]\!] \cup A_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) \qquad \wedge \qquad s \notin [\![\phi]\!] \cup A_{\mathcal{R}}(\mathcal{X})$$

which implies

$$s \in A_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) \qquad \wedge \qquad s \notin [\![\phi]\!] \qquad \wedge \qquad s \notin A_{\mathcal{R}}(\mathcal{X}).$$

Now, $s \notin A_{\mathcal{R}}(\mathcal{X})$ implies that, for some edge $(s, s') \in \mathcal{R}$, $s' \notin \mathcal{X}$. But we again obtain that $s' \notin [\![\phi]\!]$ and there is some edge $(s', s'') \in \mathcal{R}$ with $s'' \notin \mathcal{X}$. Repeating this argument forever, we can obtain an infinite path $(s, s', s'', ...)$ where none of the states satisfy ϕ. But this is impossible if $s \models \mathsf{AF}\,\phi$. We have a contradiction, and our assumption $s \notin \mathcal{X}$ must be false, thus $s \in \mathcal{X}$.

Example 4.18 Show that $\mathcal{X} = \{s_2, s_3, s_4, s_5, s_6\}$ is a fixpoint for F_b for the following Kripke structure:



(recall that $[\![\mathsf{AF}\,b]\!] = \{s_2, s_3, s_5, s_6\}$).

Solution:

$$\begin{aligned} F_b(\{s_2, s_3, s_4, s_5, s_6\}) &= [\![b]\!] \cup A_{\mathcal{R}}(\{s_2, s_3, s_4, s_5, s_6\}) \\ &= \{s_3, s_6\} \cup \{s_2, s_3, s_4, s_5, s_6\} \\ &= \{s_2, s_3, s_4, s_5, s_6\} \end{aligned}$$

Example 4.19 What is the greatest fixpoint of F_ϕ?

Solution: $F_\phi(\mathcal{S}) = [\![\phi]\!] \cup A_{\mathcal{R}}(\mathcal{S}) = [\![\phi]\!] \cup \mathcal{S} = \mathcal{S}$. Therefore, $\mathcal{S}$ is the greatest fixpoint of F_ϕ.

Property 4.21

$$\mathsf{AF}\,\mathsf{AF}\,\phi \quad \equiv \quad \mathsf{AF}\,\phi$$

Proof: From Property 4.20, $[\![\mathsf{AF}\,\phi]\!]$ is the least fixpoint of $F_\phi(\mathcal{X}) = [\![\phi]\!] \cup A_{\mathcal{R}}(\mathcal{X})$, which implies $[\![\phi]\!] \cup A_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) = [\![\mathsf{AF}\,\phi]\!]$ thus $A_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) \subseteq [\![\mathsf{AF}\,\phi]\!]$. Now, consider the least fixpoint of $F_{\mathsf{AF}\,\phi}(\mathcal{X}) = [\![\mathsf{AF}\,\phi]\!] \cup A_{\mathcal{R}}(\mathcal{X})$:

$$\begin{aligned} F_{\mathsf{AF}\,\phi}(\emptyset) &= [\![\mathsf{AF}\,\phi]\!] \cup A_{\mathcal{R}}(\emptyset) = [\![\mathsf{AF}\,\phi]\!] \\ F^2_{\mathsf{AF}\,\phi}(\emptyset) &= [\![\mathsf{AF}\,\phi]\!] \cup A_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) = [\![\mathsf{AF}\,\phi]\!] \end{aligned}$$

It follows that the least fixpoint of $F_{\mathsf{AF}\,\phi}$ is $[\![\mathsf{AF}\,\phi]\!]$. But Property 4.20 says $[\![\mathsf{AF}\,\mathsf{AF}\,\phi]\!]$ is the least fixpoint of $F_{\mathsf{AF}\,\phi}$. Therefore, $[\![\mathsf{AF}\,\mathsf{AF}\,\phi]\!] = [\![\mathsf{AF}\,\phi]\!]$.

4.7.3 Fixpoints and EG

Property 4.22 For any Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ and any state formula ϕ, define $G_\phi(\mathcal{X}) = [\![\phi]\!] \cap \mathit{PreImage}(\mathcal{X}, \mathcal{R})$. Then, $[\![\mathsf{EG}\,\phi]\!]$ is the greatest fixpoint of G_ϕ.

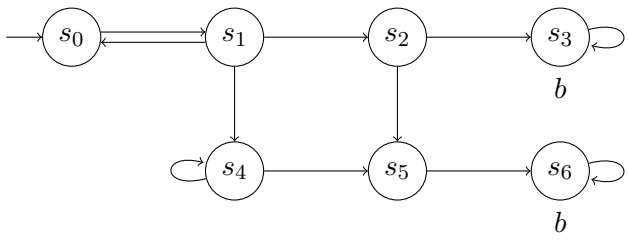
Proof: First, note that G_ϕ is monotonic since *PreImage* is monotonic. Using Property 4.9, we have

$$\begin{aligned} [\![\mathsf{EG}\,\phi]\!] &= [\![\phi \wedge \mathsf{EX}\,\mathsf{EG}\,\phi]\!] \\ &= [\![\phi]\!] \cap [\![\mathsf{EX}\,\mathsf{EG}\,\phi]\!] \\ &= [\![\phi]\!] \cap \mathit{PreImage}([\![\mathsf{EG}\,\phi]\!], \mathcal{R}) \\ &= G_\phi([\![\mathsf{EG}\,\phi]\!]) \end{aligned}$$

thus $[\![\mathsf{EG}\,\phi]\!]$ is a fixpoint of G_ϕ.

Finally, we must show that it is the *greatest* fixpoint. Let $\mathcal{X}$ be some fixpoint of G_ϕ. Since $\mathcal{X} = G_\phi(\mathcal{X})$, it follows that $\mathcal{X} \subseteq [\![\phi]\!]$. Furthermore, we have $\mathcal{X} \subseteq \mathit{PreImage}(\mathcal{X}, \mathcal{R})$, which says that, from any state $s \in \mathcal{X}$, there is an edge $(s, s') \in \mathcal{R}$ with $s' \in \mathcal{X}$. Therefore, for each state $s \in \mathcal{X}$, there exists a path that remains forever in the states in $\mathcal{X}$, which all satisfy ϕ. Thus, all states in $\mathcal{X}$ satisfy $\mathsf{EG}\,\phi$, thus $\mathcal{X} \subseteq [\![\mathsf{EG}\,\phi]\!]$.

Example 4.20 Show that $\mathcal{X} = \{s_0, s_1\}$ is a fixpoint for $G_{\neg b}$ for the following Kripke structure:



(recall that $[\![\mathsf{EG}\,\neg b]\!] = \{s_0, s_1, s_4\}$).

Solution:

$$\begin{aligned} G_{\neg b}(\{s_0, s_1\}) &= [\![\neg b]\!] \cap \mathit{PreImage}(\{s_0, s_1\}) \\ &= \{s_0, s_1, s_2, s_4, s_5\} \cap \{s_0, s_1\} \\ &= \{s_0, s_1\} \end{aligned}$$

Example 4.21 What is the least fixpoint of G_ϕ?

Solution: $G_\phi(\emptyset) = [\![\phi]\!] \cap \mathit{PreImage}(\emptyset, \mathcal{R}) = [\![\phi]\!] \cap \emptyset = \emptyset$. Therefore, $\emptyset$ is the least fixpoint of G_ϕ.

Property 4.23

$$\mathsf{EG}\,\mathsf{EG}\,\phi \equiv \mathsf{EG}\,\phi$$

4.7.4 Fixpoints and EU

Property 4.24 For any Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{A}, \mathcal{L})$ and any state formulas ϕ_1 and ϕ_2, define $U_{\phi_1\phi_2}(\mathcal{X}) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathit{PreImage}(\mathcal{X}, \mathcal{R}))$. Then, $[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$ is the least fixpoint of $U_{\phi_1\phi_2}$.

Proof: First, note that $U_{\phi_1\phi_2}$ is monotonic. From Property 4.11, we have

$$\begin{aligned}
[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!] &= [\![\phi_2 \vee (\phi_1 \wedge \mathsf{EX}\,\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2)]\!] \\
&= [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap [\![\mathsf{EX}\,\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]) \\
&= [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathit{PreImage}([\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!], \mathcal{R}) \\
&= U_{\phi_1\phi_2}([\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!])
\end{aligned}$$

thus $[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$ is a fixpoint of $U_{\phi_1\phi_2}$.

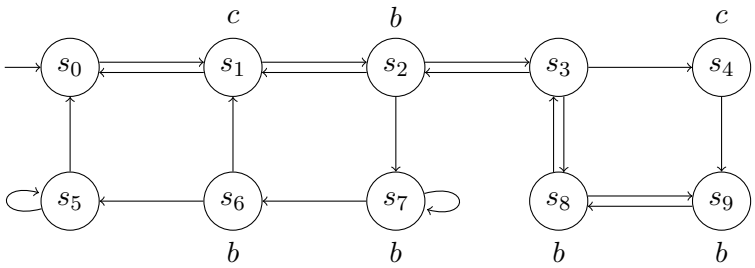
Finally, we must show that it is the *least* fixpoint by showing that $[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!] \subseteq \mathcal{X}$, for any $\mathcal{X}$ that is a fixpoint for $U_{\phi_1\phi_2}$. Consider some $s \in [\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$. By definition, this says there exists a $j \geq 0$ and a path $(s_0, s_1, s_2, ...) \in \mathit{Paths}(s)$ such that

- $s_j \models \phi_2$, and
- $\forall i < j, s_i \models \phi_1$.

We show that $s \in \mathcal{X}$ by induction on j. In the base case, $j = 0$ and $s \models \phi_2$. But by definition of $U_{\phi_1\phi_2}$ and the fact that $\mathcal{X} = U_{\phi_1\phi_2}(\mathcal{X})$, we know $[\![\phi_2]\!] \subseteq \mathcal{X}$ and trivially $s \in \mathcal{X}$.

Now, assume it holds for $j \leq k$ and prove it holds for $j = k+1$. Considering the path $(s'_0 = s_1, s'_1 = s_2, ..., s'_k = s_{k+1})$, we have $s_1 \in [\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$, and by the inductive hypothesis (since this path uses $j = k$), we have $s_1 \in \mathcal{X}$. But then $s_0 = s \in \mathcal{X}$, because $s_0 \in [\![\phi_1]\!]$ and $s_0 \in \mathit{PreImage}(\mathcal{X}, \mathcal{R})$.

Example 4.22 Show that $\mathcal{X} = \{s_1, s_2, s_4, s_6, s_7, s_8, s_9\}$ is a fixpoint for U_{bc} for the following Kripke structure:



(recall that $[\![\mathsf{E}\,b\,\mathsf{U}\,c]\!] = \{s_1, s_2, s_4, s_6, s_7\}$).

Solution:

$$\begin{aligned}
U_{bc}(\mathcal{X}) &= [\![c]\!] \cup ([\![b]\!] \cap \mathit{PreImage}(\{s_1, s_2, s_4, s_6, s_7, s_8, s_9\}, \mathcal{R})) \\
&= \{s_1, s_4\} \cup (\{s_2, s_6, s_7, s_8, s_9\} \cap \{s_0, s_1, s_2, s_3, s_4, s_6, s_7, s_8, s_9\}) \\
&= \{s_1, s_2, s_4, s_6, s_7, s_8, s_9\}
\end{aligned}$$

Property 4.25

$$\mathsf{E}\,\phi_1\,\mathsf{U}\,(\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2) \equiv \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$$

Property 4.26

$$\begin{aligned}
\mathsf{EF}\,\mathsf{EF}\,\phi &\equiv \mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,(\mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,\phi) \\
&\equiv \mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,\phi \\
&\equiv \mathsf{EF}\,\phi
\end{aligned}$$

Property 4.27

$$\begin{aligned} \mathsf{AG}\,\mathsf{AG}\,\phi &\equiv \mathsf{AG}\,\neg\mathsf{EF}\,\neg\phi \\ &\equiv \neg\mathsf{EF}\,\mathsf{EF}\,\neg\phi \\ &\equiv \neg\mathsf{EF}\,\neg\phi \\ &\equiv \mathsf{AG}\,\phi \end{aligned}$$

4.8 Fairness

Suppose we have a system of three interacting processes, where each process

1. computes,
2. waits for the semaphore,
3. updates a shared data structure,
4. releases the semaphore,

and repeats those steps forever. Suppose we model the choice of "who gets the semaphore" when multiple processes are waiting as a non-deterministic choice. Now, suppose we would like to check that a process can always eventually enter the critical section, something like

$$\mathsf{EG}(\text{“process 1 waits”} \rightarrow \mathsf{AF}\ \text{“processs 1 updates”}).$$

It is likely that this property *will not* hold, because there will be a computation path where the semaphore alternates between processes two and three, and process one will wait forever. The problem here is that we would like to have an additional constraint, namely, that the semaphore operates "fairly". In CTL, the issue of fairness can be addressed using *fairness constraints*, which are sometimes[3] simply sets of states.

Definition 4.28 A *fair path for constraint* $\mathcal{C}$ is a path $(p_0, p_1, p_2, ...)$ such that some states in $\mathcal{C}$ are visited infinitely often. Formally, there is an infinite set $\{i_1, i_2, ...\}$ where, for all k, $p_{i_k} \in \mathcal{C}$.

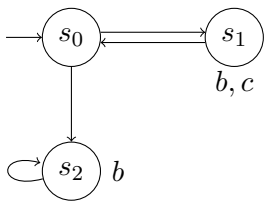
We then treat the issue of fairness by quantifying over the *fair paths*, i.e.,

- $\mathsf{A}_{\mathcal{C}}$: for all fair paths for constraint $\mathcal{C}$.
- $\mathsf{E}_{\mathcal{C}}$: for some fair path for constraint $\mathcal{C}$.

For example, we have:

$M, s \models \mathsf{A}_{\mathcal{C}}\mathsf{X}\,\phi$, if and only if, for all fair paths $(p_0, p_1, ...) \in \mathit{Paths}(s)$ for constraint $\mathcal{C}$, $p_1 \models \phi$.

Example 4.23 Consider the following Kripke structure:



[3]There are other ways to specify fairness constraints that are also useful. We will see a different one, later.

For the constraint $\mathcal{C} = [\![\neg b]\!] = \{s_0\}$, we have

$$M, s_0 \models \mathsf{A}_{\mathcal{C}}\mathsf{X}\,(b \wedge c)$$

because the only fair path for constaint $\mathcal{C}$ is $(s_0, s_1, s_0, s_1, ...)$.

We only need the operator $\mathsf{E}_{\mathcal{C}}\mathsf{G}$, because all other "fair" operators can be expressed in terms of their unfair conterparts, and $\mathsf{E}_{\mathcal{C}}\mathsf{G}$. Specifically, we will use $\mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}$ for "there exists a fair path for constraint $\mathcal{C}$".

1. $\mathsf{E}_{\mathcal{C}}\mathsf{X}\,\phi \;\equiv\; \mathsf{EX}(\phi \wedge \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt})$
2. $\mathsf{E}_{\mathcal{C}}\,\phi_1\,\mathsf{U}\,\phi_2 \;\equiv\; \mathsf{E}(\phi_1\,\mathsf{U}\,(\phi_2 \wedge \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}))$

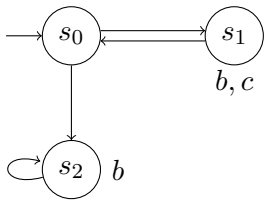
Since EX, EU, and EG are an adequate set of temporal operators, we are done.

4.8.1 Labeling algorithm for $\mathsf{E}_{\mathcal{C}}\mathsf{G}$

Recall that one algorithm for labeling $\phi = \mathsf{EG}\,\phi_1$ was to use strongly connected components (SCCs), based on the observation that, for ϕ_1 to occur in every state on an infinitely–long path, we need to have cycles of states satisfying ϕ_1. We only need a small modification if we want an infinitely–long *fair* path: we need to have SCCs where at least one state in the SCC is in set $\mathcal{C}$. We have the following algorithm.

1. Create a new graph M' by removing all states that do not satisfy ϕ_1, and any edges associated with those states.
2. Determine the SCCs for M'.
3. For each SCC **that contains at least one state in** $\mathcal{C}$,
   - if the SCC contains more than one state, label all its states with ϕ
   - if the SCC contains only one state, and the state has an edge to itself, label it with ϕ.
4. Any state that can reach (in M') a state labeled with ϕ should also be labeled with ϕ.

Example 4.24 Given the Kripke structure below and constraint $\mathcal{C} = [\![\neg b]\!] = \{s_0\}$, which states satisfy $\mathsf{A}_{\mathcal{C}}\mathsf{X}(b \wedge c)$?



Solution: We have the equivalence

$$\neg\mathsf{A}_{\mathcal{C}}\mathsf{X}b \;\equiv\; \mathsf{E}_{\mathcal{C}}\mathsf{X}\neg b \;\equiv\; \mathsf{EX}(\neg b \wedge \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt})$$

which says that, $\mathsf{A}_{\mathcal{C}}\mathsf{X}\,b$ does not hold if there is a fair path with a next state satisfying $\neg b$. Rewriting our formula, we obtain

$$\mathsf{A}_{\mathcal{C}}\mathsf{X}(b \wedge c) \;\equiv\; \neg\mathsf{EX}(\neg(b \wedge c) \;\wedge\; \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt})$$

Using the labeling algorithm for $\mathsf{E}_{\mathcal{C}}\mathsf{G}$, we determine SCCs and check for states in $\mathcal{C}$:

- SCC $\{s_0, s_1\}$ has one state, s_0, in $\mathcal{C}$
- SCC $\{s_2\}$ has no states in $\mathcal{C}$

Therefore, $[\![\mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}]\!] = \{s_0, s_1\}$. Also, we have $[\![\neg(b \wedge c)]\!] = \{s_0, s_2\}$. That gives us

$$[\![\neg(b \wedge c) \ \wedge \ \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}]\!] \ = \ \{s_0, s_2\} \cap \{s_0, s_1\} \ = \ \{s_0\}$$

We next determine EX of that formula, using

$$\mathit{PreImage}(\{s_0\}, \mathcal{R}) \ = \ \{s_1\}$$

Finally, taking the complement, we have

$$[\![\mathsf{A}_{\mathcal{C}}\mathsf{X}(b \wedge c)]\!] \ = \ \{s_0, s_2\}$$

Verify: We already determined $s_0 \in [\![\mathsf{A}_{\mathcal{C}}\mathsf{X}(b \wedge c)]\!]$ in the previous example. But what about s_2? This is a correct solution because *there are no fair paths from* s_2. Thus, the statement, "for all fair paths, the next state satisfies $b \wedge c$" is trivially true. If this is not as intended, then the issue is how to define $\mathsf{A}_{\mathcal{C}}$ when there are no fair paths. If the intended definition is, "there are some fair paths, and for all of them ...", then the equivalence

$$\neg\mathsf{A}_{\mathcal{C}}\mathsf{X}b \ \equiv \ \mathsf{E}_{\mathcal{C}}\mathsf{X}\neg b$$

does *not* hold.

Chapter 5

High–level Formalisms

We can verify properties of a system via the following steps.

1. Build an appropriate Kripke structure.
2. Express the properties we want in a suitable logic.
3. Use an automatic tool to model check the Kripke structure against the properties.

We have seen how to do steps 2 and 3 above. What about step 1? This can be done in a few ways:

- By hand. That's what we have done so far, in class. However, step 1 can be difficult if the system is large or complex.
- Write a program to generate a Kripke structure for a particular system, and analyze it. This works, but, for each new system to analyze, you need to write a new program, which means more time spent debugging.
- Use another, more compact and convenient model to describe the system. For this to be effective, we must be able to automatically construct a Kripke structure described by the model. That way, we can write, debug, test, etc., *one* program that reads a model as input and constructs its underlying Kripke structure.

  These compact models are called **high–level formalisms**.

5.1 Requirements of a formalism

What does a high–level formalism need to be able to do? For model checking, it must

1. Describe a finite set of states.
2. Provide the initial state(s) of the system.
3. Provide *formal* rules (thus the name "formalism") for moving between states: if we are currently in state s, what state(s) can be reached from s in one step?
4. Be easier to use than describing a Kripke structure by hand. This is simply a practical requirement. This is the "high–level" part of "high–level formalisms".

Note that the first three requirements will allow us to construct a Kripke structure. We then write queries in terms of the high–level formalism. In particular, the atomic propositions will be expressed in terms of features in the high–level model.

There are many high–level formalisms. Why? If you think of a high–level formalism as a high–level language, then it is like the several high–level programming languages:

- certain tasks are easier using certain languages;
- people have preferences;
- they are easy to invent for a particular application.

For us, the choice of formalism is mostly irrelevant. (Think of writing a compiler; this task is basically the same for a C compiler as for a Pascal compiler.)

5.2 Petri nets

Petri nets are one of many high–level formalisms. We will study these because

- They are graphical (easier to learn? easier to understand a model expressed in them?).
- They are fairly powerful and expressive.
- They have a rich underlying theory[1]
- They are fairly well–known .
- Analysis is relatively easy (more features usually implies more difficult analysis).

5.2.1 Informal introduction

A Petri net is a directed graph with two types of nodes:

Places, drawn as circles, which contain a non-negative integer number of **tokens**.

Transitions, drawn as rectangles or bars, corresponding to actions that cause tokens to "move".

Arcs connect places to transitions and transitions to places, never places to places or transitions to transitions (this is a special type of graph called *bipartite*).

So, let's see how Petri nets are a high-level formalism by checking the list of requirements:

1. How does a Petri net describe a discrete set of states?

   Each place can contain a non-negative integer number of tokens. The tokens within a place are indistinguishable. The state of the Petri net is completely described by its *marking*, which determines, for each place, how many tokens it contains. So, if $\mathcal{P}$ is the set of places, a marking $\mathbf{m}$ is a function $\mathbf{m} : \mathcal{P} \to \mathbb{N}$. Or, equivalently, a marking $\mathbf{m}$ is a vector of naturals, $\mathbf{m} \in \mathbb{N}^{\mathcal{P}}$, where $\mathbf{m}[p]$ is the number of tokens in place p. Hence, the set of states described by a Petri net is at most the set $\mathbb{N}^{\mathcal{P}}$, possibly infinite but countable (thus always discrete).

[1] T. Murata. "Petri Nets: Properties, Analysis, and Applications", in *Proceedings of the IEEE*, 77 (4), April 1989, pages 541–580

2. How to specify the initial state of a Petri net?

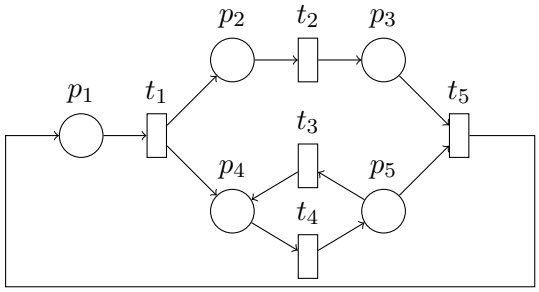
   This is done by specifying the *initial marking* of the Petri net. This means that there is a unique initial state, in contrast to the definition of a generic Kripke structure, which allows for a set of initial states. This is a minor restriction simply due to tradition; in principle Petri nets could have been defined as having multiple initial markings (but they were not).

3. What are the rules for changing states in a Petri net?

   The marking of the net is changed by transitions. A transition is said to be *enabled* if all of its input places (the places with an arc to this transition) have at least one token. An enabled transition may *fire* by removing a token from each input place and adding a token to each output place. Thus, the total number of tokens is not necessarily "conserved".

Example 5.1 In the Petri net below, suppose the current marking is

$$\begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \left[\begin{array}{ccccc} 3 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$



$\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$

$\mathcal{T} = \{t_1, t_2, t_3, t_4, t_5\}$

Which transitions are enabled in this marking?

t_1**: enabled** because its input places (p_1) contain tokens.

t_2**: disabled** because its input places (p_2) do not contain tokens.

t_3**: disabled** because its input places (p_5) do not contain tokens.

t_4**: disabled** because its input places (p_4) do not contain tokens.

t_5**: disabled** because (at least one of) its input places (p_3 and p_5) do not contain tokens.

If t_1 fires, what is the new marking?

- Remove 1 token from p_1
- Add 1 token to p_2
- Add 1 token to p_4

This results in a new marking,

$$\begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \left[\begin{array}{ccccc} 2 & 1 & 0 & 1 & 0 \end{array}\right], \end{array}$$

where transitions $\{t_1, t_2, t_4\}$ are enabled. Any of them may fire. If t_1 fires again, the new marking is

$$\begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \left[\begin{array}{ccccc} 1 & 2 & 0 & 2 & 0 \end{array}\right] \end{array}$$

and now transitions $\{t_1, t_2, t_4\}$ are enabled. From here, if t_4 fires, the new marking is

$$\begin{array}{c} \begin{array}{cccccc} & p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \left[\begin{array}{cccccc} & 1 & 2 & 0 & 1 & 1 \end{array}\right] \end{array}$$

and now transitions $\{t_1, t_2, t_3, t_4\}$ are enabled. From here, if t_2 fires, the new marking is

$$\begin{array}{c} \begin{array}{cccccc} & p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \left[\begin{array}{cccccc} & 1 & 1 & 1 & 1 & 1 \end{array}\right] \end{array}$$

and now all transitions are enabled. In this marking, we see an example of *conflict*:

- if t_3 fires, then t_5 will become disabled;
- if t_5 fires, then t_3 will become disabled.

This example illustrates how Petri nets can model the following behaviors:

conflict: firing of one transition may disable another (e.g., t_3 and t_5).

choice: one place (e.g., p_5) can be input to multiple transitions.

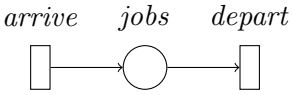
concurrency: firing of one transition does not affect firing of the other (e.g, t_2 and t_4).

synchronization: one transition (e.g., t_5) requires tokens in multiple input places to be enabled.

sequentialization: if the initial marking is $[1, 0, 0, 0, 0]$ instead of $[3, 0, 0, 0, 0]$, t_3 can only become enabled after t_4 fires.

mutual exclusion: if the initial marking is $[1, 0, 0, 0, 0]$ instead of $[3, 0, 0, 0, 0]$, t_3 and t_4 can never be enabled in the same marking; in fact, all pairs of transitions are now mutually exclusive, except for (t_3, t_5), (t_2, t_3), and (t_2, t_4).

Example 5.2 The Petri net below models a single–server service station with an unbounded queue. The number of tokens in place *jobs* represents the number of customers in the station.



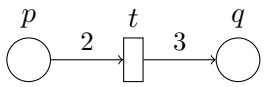
- Transition *arrive* has no input places; therefore it is always enabled, and it "creates" tokens. This is called a *source* transition.
- Transition *depart* has no output places; it "consumes" tokens. This is called a *sink* transition.

It is easy to see that the number of tokens in place *jobs* can grow arbitrarily large.

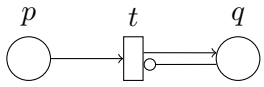
5.2.2 Petri net extensions

Petri nets as described so far are *ordinary* Petri nets. Several extensions have been proposed (several are commonly used):

Arc weights: these specify a number of tokens to be added or removed. Below, t is enabled iff p has at least 2 tokens; firing t removes 2 tokens from p and adds 3 tokens to q.



Inhibitor arcs: these allow the presence of tokens in places to disable a transition. Below, t is disabled if place q contains at least one token.



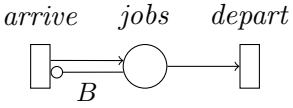
If the inhibitor arc had weight w, then t would be disabled if q contains at least w tokens.

Transition guards: these are boolean functions of the marking that must be true for a transition to be enabled.

Marking–dependent arc weights: arc weights can be functions of the current marking, instead of constants. These are also known as "self-modifying nets".

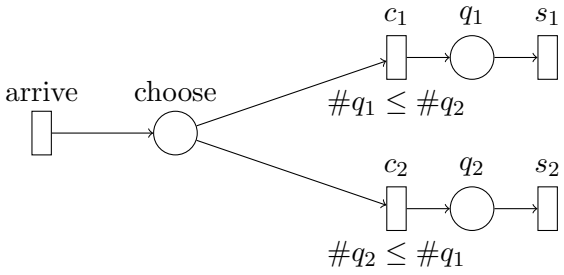
Color: Tokens have colors (the formalisms is called "Colored Petri Nets"). Colors can store data (i.e., distinguish tokens). There can be multiple "dimensions" of colors. This makes things *significantly* more complex, so we will not consider this extension.

Example 5.3 The Petri net below models a single–server service node with a bounded queue.



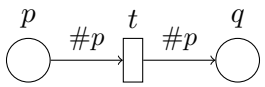
For the given initial marking, the maximum number of tokens in place *jobs* is B, since transition *arrive* is disabled when there are B (or more) tokens in place *jobs*. If an initial marking specifies more than B tokens in place *jobs*, then transition *arrive* will remain disabled until transition *depart* fires enough times that fewer than B tokens appear in place *jobs*.

Example 5.4 The Petri net below models the arrival of customers, where each customer joins either queue 1 or queue 2, based on which one is shorter.



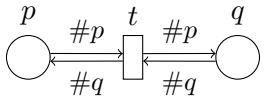
This is done via the guard $\#q_1 \leq \#q_2$ on transition c_1, and the guard $\#q_2 \leq \#q_1$ on transition c_2, where $\#p$ means the number of tokens currently in place p. Therefore, c_1 is enabled only if the number of tokens in q_1 is not more than the number of tokens in q_2, and vice–versa. If the queues have equal length, then both c_1 and c_2 are enabled (the customer chooses one of the two queues, nondeterministically).

Example 5.5 We can remove all tokens from a place, and add them to another place, using marking–dependent arc cardinalities, as shown below.



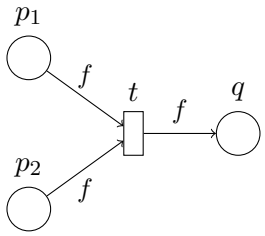
When t fires, all tokens are removed from p and added to q. Important: technically, t is enabled even when p is empty, but its firing in this case does not change the marking. How would you prevent from being enabled if p is empty?

Example 5.6 We can exchange the number of tokens in two places, using marking–dependent arc cardinalities as shown below.



When t fires, all tokens are removed from p and q, and the (previous) number of tokens in p is added to q, and the (previous) number of tokens in q is added to p.

Example 5.7 The Petri net below shows how to synchronize f jobs.



If $f = 1$, then transition t is "ordinary" synchronization of a completed pair of tasks, one in p_1 and one in p_2. If $f = 2$, then transition t synchronizes two completed pairs of tasks at a time. If $f = \min(\#p_1, \#p_2)$, then transition t synchronizes as many completed pairs of tasks as possible.

5.2.3 Expressive power of Petri nets

Suppose we assign a symbol to each transition, and the firing of that transition produces the symbol. Then we can ask "what is the language generated by the Petri net"? Or, equivalently, a word is "accepted" if there exists a firing sequence that generates that word. The expressive power can be measured by answering the question, "what types of languages are accepted by Petri nets"? It turns out:

- Arc weights do not extend the power of ordinary Petri nets, thus we will say "Petri nets" to mean ordinary Petri nets or ordinary Petri nets extended with arc weights, from now on.

- Petri nets with inhibitor arcs are Turing equivalent.
- Petri nets without inhibitor arcs are **not** Turing equivalent. Just like non-deterministic push-down automata (NPDAs, which accept exactly the set of context-free languages), they are strictly more powerful than finite automata. Interestingly enough, there are context-free languages that cannot be accepted by Petri nets, and there are Petri net languages that cannot be accepted by NPDAs.
- Guards or marking-dependent weights are also Turing-equivalent extensions. However, it is possible to (rather severely) restrict marking-dependent weights (what marking-dependent expressions can be used on which arcs) so that the resulting class of Petri nets is strictly more expressive than Petri nets but still not as powerful as Turing Machines.

On the other hand, if the number of tokens in each place is bounded (thus, the total number of markings that can be reached from the initial marking is *finite*), then the Petri net describes a finite state machine (regardless of which extensions are allowed), and the above extensions serve only to simplify the model description.

5.2.4 Formal definition

A Petri net is a tuple $(\mathcal{P}, \mathcal{T}, I, O, H, g, \mathbf{m}_0)$ where

- $\mathcal{P}$ is a finite set of places.
- $\mathcal{T}$ is a finite set of transitions, with $\mathcal{T} \cap \mathcal{P} = \emptyset$
- $I : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$ is a function to describe the marking-dependent input arc cardinalities
- $O : \mathcal{T} \times \mathcal{P} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$ is a function to describe the marking-dependent output arc cardinalities
- $H : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N} \cup \{\infty\}$ is a function to describe the marking-dependent inhibitor arc cardinalities
- $g : \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \{false, true\}$ are the transition guard functions
- $\mathbf{m}_0 \in \mathbb{N}^{\mathcal{P}}$ is the initial marking

Note: the various extensions we discussed can be "removed" in the above definition as follows:

- If we don't want marking-dependent arc cardinalities, ensure that the value of I, O, and H is independent of the third parameter, the marking.
- If we don't want inhibitor arcs, set H to the constant ∞.
- If we don't want guards, set g to the constant $true$.

Definition 5.1 (Enabling rule) Transition t is enabled in marking $\mathbf{m}$ if:

$$g(t, \mathbf{m}) = true \qquad \text{and} \qquad \forall p \in \mathcal{P}, \quad I(p, t, \mathbf{m}) \leq \mathbf{m}[p] < H(p, t, \mathbf{m})$$

Definition 5.2 (Firing rule) If transition t is enabled in marking $\mathbf{m}$, its firing leads to a new marking $\mathbf{n}$, with

$$\forall p \in \mathcal{P}, \qquad \mathbf{n}[p] = \mathbf{m}[p] - I(p, t, \mathbf{m}) + O(t, p, \mathbf{m})$$

We write this as $\mathbf{m} \xrightarrow{t} \mathbf{n}$, and naturally extend this relation to sequences of transitions $\sigma \in \mathcal{T}^*$, writing $\mathbf{m} \xrightarrow{\sigma} \mathbf{n}$, or simply $\mathbf{m} \xrightarrow{*} \mathbf{n}$ if the specific sequence is irrelevant. Note that we always have $\mathbf{m} \xrightarrow{*} \mathbf{m}$, since $\mathbf{m} \xrightarrow{\epsilon} \mathbf{m}$, but not necessarily $\mathbf{m} \xrightarrow{+} \mathbf{m}$, where "+" means a non-empty sequence of transitions.

Chapter 6

Reachability and Coverability

We now consider a fundamental question for all high–level formalisms, but phrase it in terms of Petri nets.

The reachability problem: Given a Petri net with initial marking $\mathbf{m}_0$, is it possible to (eventually) reach a specified marking $\mathbf{m}$?

This problem is known to be decidable for ordinary Petri nets (and undecidable for Turing–equivalent Petri net extensions), but requires exponential time and space in the worst case. This decidability result is somewhat surprising because the set of markings that a Petri net can reach may be infinite. In fact, Petri nets were defined in 1961 but reachability remained an open question for 20 years; it was resolved (positively) in 1981. A related (but simpler) problem is the following.

The coverability problem: Given a Petri net with initial marking $\mathbf{m}_0$ and a marking $\mathbf{m}$ for that Petri net, is it possible to (eventually) reach a marking $\mathbf{m}'$ that *covers* $\mathbf{m}$, i.e., for all places $p \in \mathcal{P}$, $\mathbf{m}'[p] \geq \mathbf{m}[p]$ or, in vector form, $\mathbf{m}' \geq \mathbf{m}$?

6.1 The reachability set and the reachability graph

Given a Petri net $(\mathcal{P}, \mathcal{T}, I, O, H, g, \mathbf{m}_0)$, its *reachability set* is $\mathcal{S} = \{\mathbf{m} \in \mathbb{N}^{\mathcal{P}} : \mathbf{m}_0 \xrightarrow{*} \mathbf{m}\}$, while its *reachability graph* is the (edge-labeled) graph $(\mathcal{S}, \mathcal{E})$, with $\mathcal{E} = \{(\mathbf{m}, \mathbf{n}, t) \in \mathcal{S} \times \mathcal{S} \times \mathcal{T} : \mathbf{m} \xrightarrow{t} \mathbf{n}\}$.

We already know that the reachability set, thus the reachability graph, is not necessarily finite, but we can attempt to build it using Algorithm 6.1. The algorithm terminates iff the reachability set is finite, in which case:

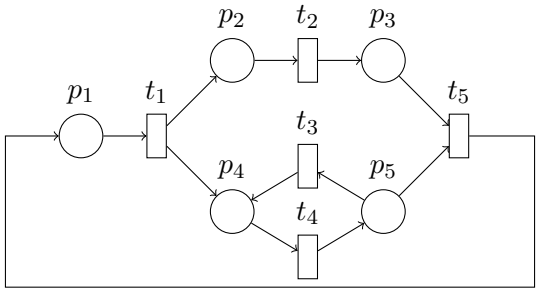
- If markings can be added to and removed from sets $\mathcal{U}$ and $\mathcal{S}$ in constant time (not a realistic assumption), then the algorithm has complexity $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{T}|)$, if we check each transition in each marking.
- If markings are removed from $\mathcal{U}$ in FIFO order, we get "breadth–first" search.
- If markings are removed from $\mathcal{U}$ in LIFO order, we get "depth–first" search.
- The reachability graph contains enough information to answer any reachability question, and can be used to determine a firing sequence from $\mathbf{m}_0$ to any reachable marking $\mathbf{m}$. But what if the reachability set $\mathcal{S}$ is infinite?

Algorithm 6.1 Reachability graph construction

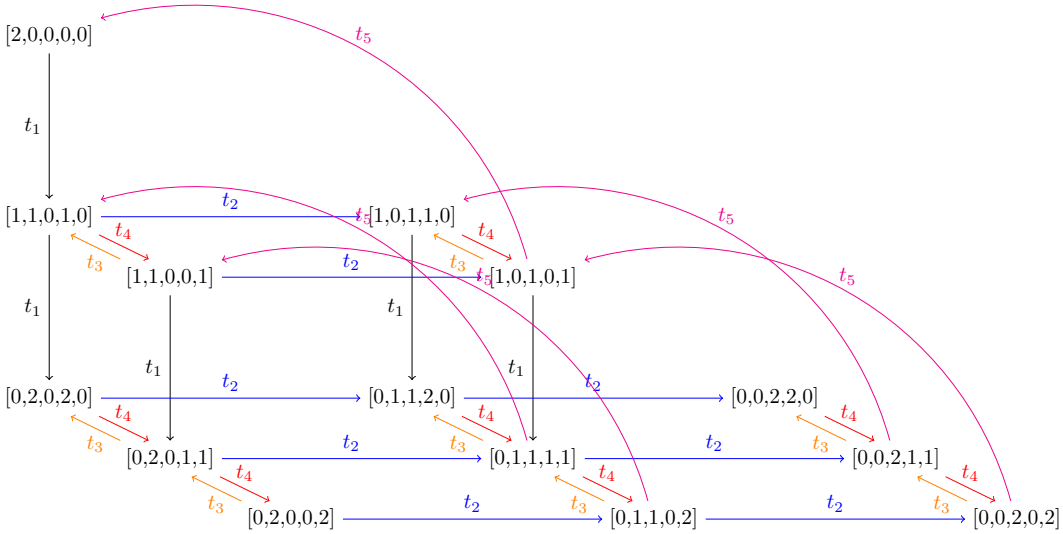
$\mathcal{U} \leftarrow \{\mathbf{m}_0\}$; // set of unexplored markings
$\mathcal{S} \leftarrow \{\mathbf{m}_0\}$; // reachability set so far
$\mathcal{E} \leftarrow \emptyset$; // edges in the reachability graph
while $\mathcal{U} \neq \emptyset$ **do**
  remove a marking $\mathbf{m}$ from $\mathcal{U}$;
  for each transition t enabled in $\mathbf{m}$ **do**
    $\mathbf{m}' \leftarrow$ the marking reached from $\mathbf{m}$ by firing t;
    if $\mathbf{m}' \notin \mathcal{S}$ **then**
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{m}'\}$;
      $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathbf{m}'\}$;
    endif
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{m}, \mathbf{m}', t)\}$;
  endfor
endwhile

Example 6.1 Build the reachability graph for the Petri net below, with initial marking

p_1	p_2	p_3	p_4	p_5
[2	0	0	0	0]



Using the reachability graph construction algorithm and some patience, we obtain the following graph (edges are color coded by transition).

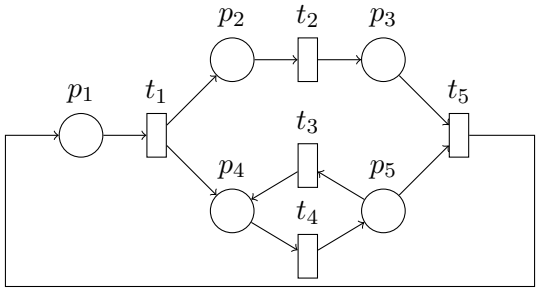


6.1.1 State explosion

High–level models (including Petri nets) often produce an extremely large number of reachable states, even for "small" models. This is known as the "state explosion problem".

Example 6.2 Compute the number of reachable marking for the Petri net below, when the initial marking is

$$\begin{array}{c} \begin{array}{ccccccc} & p_1 & p_2 & p_3 & p_4 & p_5 & \end{array} \\ \left[\begin{array}{ccccc} N & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$



In general, it is unknown how to answer this question without building the reachability graph. However, this Petri net is simple enough that we can answer this question by studying the Petri net, as follows. Note that, if transition t_1 fires n more times than transition t_5, then

- The sum of tokens in places p_2 and p_3 is exactly n. There are $n+1$ ways that this can occur.
- Similarly, the sum of tokens in places p_4 and p_5 is exactly n.
- The above two sums are completely independent.

Therefore, the number of reachable markings, given that transition t_1 has fired n more times than transition t_5, is $(n+1)^2$. For the given initial marking, the number of times t_1 may fire, before t_5 fires, is $0, 1, 2, ..., N$. The number of reachable markings is therefore

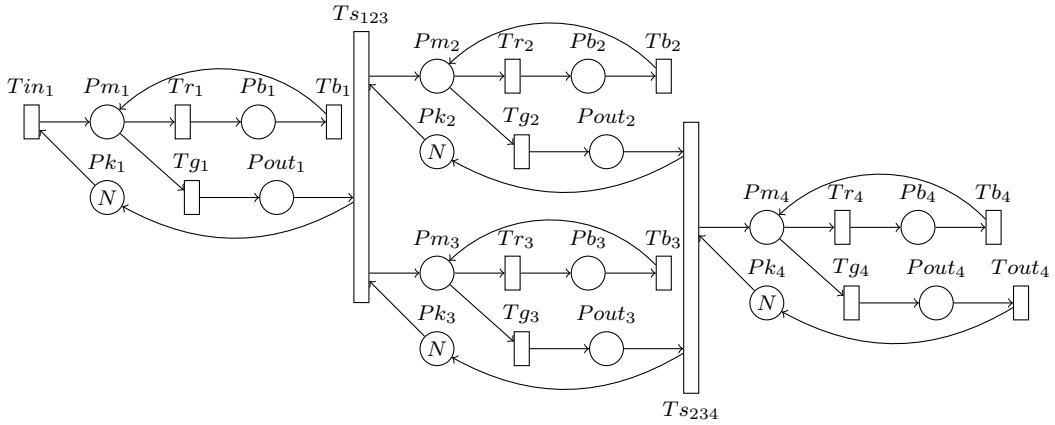
$$\sum_{n=0}^{N}(n+1)^2 \;=\; \sum_{n=0}^{N} n^2 + 2\sum_{n=0}^{N} n + \sum_{n=0}^{N} 1 \;=\; \cdots \;=\; \frac{(2N+3)(N+2)(N+1)}{6}$$

In particular, note that (as a sanity check):

- If $N = 0$, the number of markings is $(3 \cdot 2 \cdot 1)/6 = 1$
- If $N = 2$, the number of markings is $(7 \cdot 4 \cdot 3)/6 = 14$

Thus, the size of the reachability set grows as $\mathcal{O}(N^3)$.

Example 6.3 The Petri net below is a model of a "kanban" manufacturing system, composed from four stations. In each station, a part is processed, either successfullly (and the part can move on to the next stage), or unsuccessfully (which requires additional work to "un-do" the previous processing). Station 1 produces parts used (in parallel) by stations 2 and 3, and stations 2 and 3 produce parts used (together) by station 4.



The initial marking controls the maximum number of raw parts N that can be present at each station. It can be shown that the number of reachable markings is exactly

$$|\mathcal{S}| = \frac{(N+1)^3(N+2)^3(N+3)^3(3N^2+12N+10)}{2160}$$

a quantity that grows as $\mathcal{O}(N^{11})$.

6.2 Coverability

6.2.1 Coverability set

When the reachability set of a Petri net is infinite, we cannot of course generate it in its entirety. It is important to realize that this can only happen when the number of tokens in at least one of the places of the Petri net can grow arbitrarily large. The next best thing we can do, then, is to explore the *coverability set*. To discuss this concept, though, we need to introduced the notion of ω-marking: this is just like an ordinary marking, but it may use "ω" instead of a natural number as the number of tokens in a place, to signify that this number can be arbitrarily large.

Thus, an ω-marking is an element of $(\mathbb{N}\cup\{\omega\})^{\mathcal{P}}$ and, when comparing markings elementwise, we have that $\omega \geq \omega$ and, for any $n \in \mathbb{Z}$, $\omega > n$; also, when firing transitions, we have $\omega + n = \omega - n = \omega$,

Definition 6.1 Given a Petri net $(\mathcal{P}, \mathcal{T}, I, O, H, g, \mathbf{m}_0)$, an ω-marking $\mathbf{m}'$ *covers* an ω-marking $\mathbf{m}$ iff $\mathbf{m}'[p] \geq \mathbf{m}[p]$ for all places $p \in \mathcal{P}$. We write $\mathbf{m}' \geq \mathbf{m}$ if $\mathbf{m}'$ covers $\mathbf{m}$, and $\mathbf{m}' \gneq \mathbf{m}$ if $\mathbf{m}' \geq \mathbf{m}$ but $\mathbf{m}' \neq \mathbf{m}$. Note that an ω-marking always covers itself.

Definition 6.2 Given a Petri net $(\mathcal{P}, \mathcal{T}, I, O, H, g, \mathbf{m}_0)$, a marking $\mathbf{m}$ is said to be *coverable* if there exists a marking $\mathbf{m}' \geq \mathbf{m}$ that is reachable from the initial marking $\mathbf{m}_0$.

Definition 6.3 A *coverability set* of a Petri net is a finite set $\mathcal{C}$ of ω-markings satisfying:
(1) for every reachable marking $\mathbf{m}$, there exists an ω-marking $\mathbf{m}' \in \mathcal{C}$ such that $\mathbf{m} \leq \mathbf{m}'$, and
(2) for every ω-marking $\mathbf{m}' \in \mathcal{C}$, either $\mathbf{m}'$ is a reachable marking in the Petri net (thus it is an ordinary marking, i.e., none of its components are ω), or there exists a strictly increasing infinite sequence of reachable markings $(\mathbf{m}_0, \mathbf{m}_1, ...)$ that converges to $\mathbf{m}'$.
Then, we say that a coverability set $\mathcal{C}$ is *minimal* iff no proper subset of $\mathcal{C}$ is a coverability set.

The second condition is essential: without it, $\{[\omega, ..., \omega]\}$ (the set containing a single ω-marking with all components equal to ω) would always be a (minimal) coverability set for the Petri net.

Theorem 6.4 Every Petri net admits a unique minimal coverability set.

6.2.2 Coverability tree

A *coverability tree* is a tree of ω-markings that covers all markings reachable from the initial marking. A coverability tree can be built using the following algorithm:

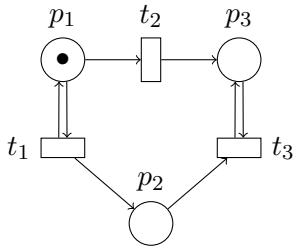
Algorithm 6.2 Coverability Tree Construction

set $\mathbf{m}_0$ as the root of the tree; tag $\mathbf{m}_0$ as "new".
while there exist markings tagged "new"
  select a marking $\mathbf{m}$ tagged as "new" and change its tag to "blank";
  if there exists $\mathbf{m}'$, on the path from $\mathbf{m}_0$ to $\mathbf{m}$, with $\mathbf{m}' = \mathbf{m}$ **then**
    tag $\mathbf{m}$ as "old";
    continue;
  elsif $\mathbf{m}$ enables no transitions **then**
    tag $\mathbf{m}$ as "dead";
    continue;
  endif
  for each transition t enabled in $\mathbf{m}$ **do**
    $\mathbf{m}' \leftarrow$ the marking reached from $\mathbf{m}$ by firing t
    if there exists $\mathbf{m}''$ on the path from $\mathbf{m}_0$ to $\mathbf{m}'$ with $\mathbf{m}' \gneq \mathbf{m}''$
      for each p such that $\mathbf{m}'[p] > \mathbf{m}''[p]$ **do**
        $\mathbf{m}'[p] \leftarrow \omega$;
      endfor
    endif
    add $\mathbf{m}'$ to the tree, by pointing to it from $\mathbf{m}$ with an edge labeled t;
    tag $\mathbf{m}'$ as "new";
  endfor
endwhile

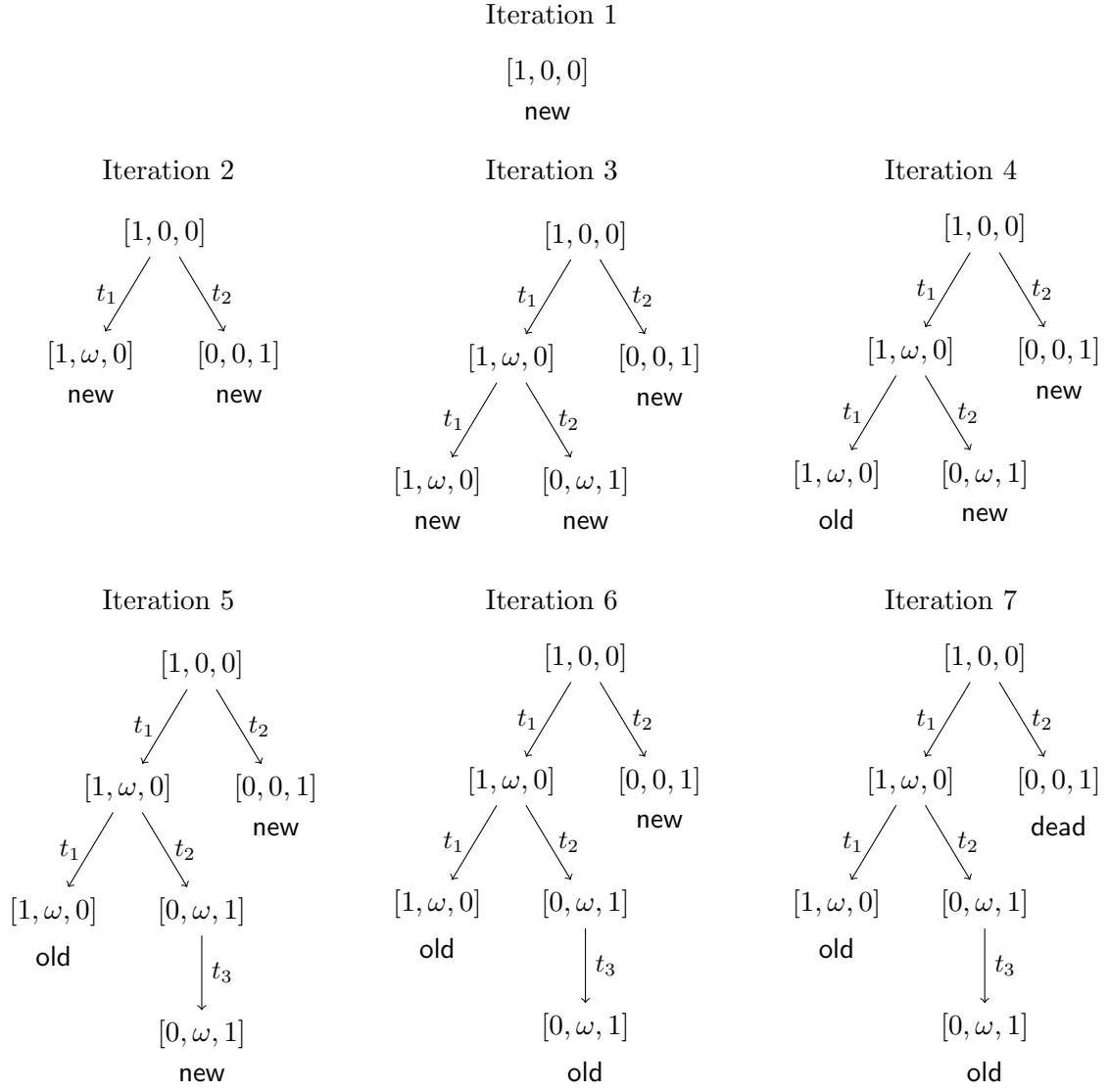
Note that the statement "if there exists $\mathbf{m}''$, on the path from $\mathbf{m}_0$ to $\mathbf{m}$, with $\mathbf{m}' \gneq \mathbf{m}''$" has a subtle implication: if there are multiple such $\mathbf{m}''$, then we may be able to add several ω's to marking $\mathbf{m}'$. Also, adding an ω to $\mathbf{m}'$ could cause *another* $\mathbf{m}''$ to become covered by $\mathbf{m}'$. For example, if $\mathbf{m}' = [1, 1]$ and the path from $\mathbf{m}_0$ to $\mathbf{m}'$ includes markings $[1, 0]$ and $[0, 2]$, then $\mathbf{m}'$ covers $[1, 0]$ but does not cover $[0, 2]$. Because $[1, 0]$ is covered, $\mathbf{m}'$ becomes $[1, \omega]$. But now $\mathbf{m}' = [1, \omega]$ covers $[0, 2]$, and so $\mathbf{m}'$ should become $[\omega, \omega]$.

Obviously, the nodes of a coverability tree form a coverability set, once duplicate nodes are removed, but not necessarily a minimal coverability set, as the following example demonstrates.

Example 6.4 Consider the Petri net below, with initial marking $[1, 0, 0]$. Transition t_1 can fire arbitrarily many times, until t_2 fires; then, t_3 can fire as many times as t_1 fired. Using the above algorithm, we can build a coverability tree, drawn below at the beginning of each iteration of the while loop.



Note that $\{[1,0,0],[1,\omega,0],[0,0,1],[0,\omega,1]\}$ is a coverability set of the Petri net, but not a minimal one. To minimize it, remove from it $[1,0,0]$ (which is covered by $[1,\omega,0]$) and $[0,0,1]$ (which is covered by $[0,\omega,1]$), resulting in the minimal coverability set $\{[1,\omega,0],[0,\omega,1]\}$.



6.2.3 Coverability graph

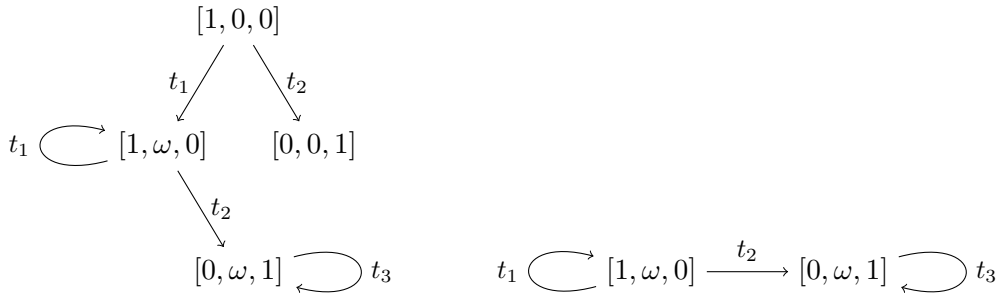
A *coverability graph* is similar to a coverability tree. Nodes in a coverability graph correspond to the unique markings in a coverability tree. In a coverability graph, there is an edge from $\mathbf{m}$ to $\mathbf{m}'$, labeled with transition t, if

- t is enabled in marking $\mathbf{m}$, and
- the firing of t in $\mathbf{m}$ leads to marking $\mathbf{m}''$ with $\mathbf{m}' \geq \mathbf{m}''$.

Note that a coverability graph (or tree) is not necessarily unique for a Petri net, because in general either one could depend on the algorithm used to generate it, or the order in which markings

are discovered. This is because a coverability graph (or tree) may or may not contain markings that are covered by other markings.

Example 6.5 Two coverability graphs for the previous example are shown below. The left one takes all unique markings from the coverability tree built in Example 6.4; the right one eliminates markings that are covered by a marking containing ω.



A coverability graph (or tree or set) does *not* contain enough information to definitively answer the reachability problem:

- We can say for certain that a marking $\mathbf{m}$ *is not reachable* if there is no marking $\mathbf{m}' \geq \mathbf{m}$ in the coverability graph.
- We can say for certain that a marking $\mathbf{m}$ (without any ω entries) *is reachable* if marking $\mathbf{m}$ is in the coverability graph.
- A marking $\mathbf{m}$ (without any ω entries) *might be reachable* if there is a marking $\mathbf{m}' \gtrsim \mathbf{m}$ in the coverability graph.

The inability to say for certain that a marking is reachable is due to the loss of information that occurs due to the symbol ω, and to the fact that a reachable marking $\mathbf{m}$ may be not be in the coverability set if there is a rechable marking $\mathbf{m}'$ s.t. $\mathbf{m}' \gtrsim \mathbf{m}$.

Differences between the coverability graph and the reachability graph:

- The coverability graph is always finite, even for unbounded Petri nets, while the reachability graph will be infinite in this case.
- The coverability graph contains no ω-markings with ω entries iff the reachability graph is finite.
- The coverability graph contains an ω-marking with ω entries iff the reachability graph is infinite.

6.3 Model checking with Petri nets

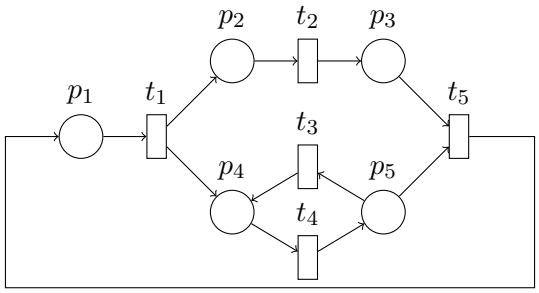
If the choice of which Petri net transition to fire is non-deterministic, and assuming the reachability graph is finite, then a (high–level) Petri net model can be used to describe a Kripke structure: the reachability graph becomes the Kripke structure (strictly speaking, we should remove the edge labels indicating the identity of the PN transition causing that change of marking; in practice, this information can be useful to understand the behavior, and to debug the Petri net). However, some care must be taken since the reachability graph may have markings with no outgoing edges; this can be handled if we either

1. define an atomic proposition holding only in the dead states, and add self-loops on them, or
2. modify the model checking algorithms so that they can work correctly also with finite paths.

Also, we must define the atomic propositions of interest and label the states in the Kripke structure accordingly. Each atomic proposition a will correspond to a function $f_a : \mathbb{N}^{\mathcal{P}} \to \{0, 1\}$, so that the labeling function $\mathcal{L}$ satisfies $a \in \mathcal{L}(\mathbf{m})$ iff $f_a(\mathbf{m}) = 1$. This allows us to describe the atomic propositions and the labeling function in terms of the Petri net itself. Finally, we can write properties in our favorite logic.

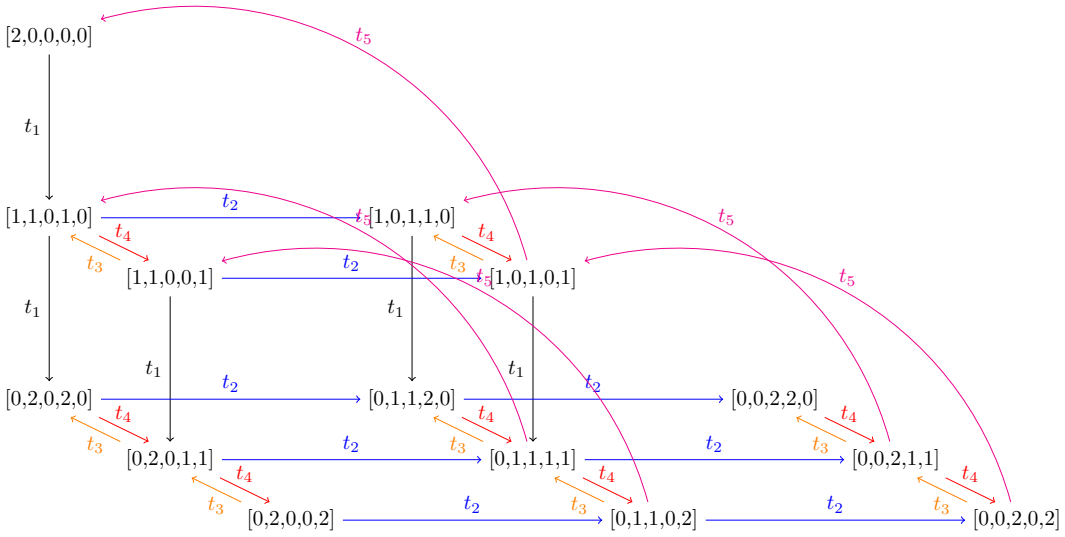
Example 6.6 For the Petri net below, with initial marking

$$\begin{array}{c} p_1 \; p_2 \; p_3 \; p_4 \; p_5 \\ [\; 2 \quad 0 \quad 0 \quad 0 \quad 0 \;] \end{array}$$



is the property AF(place p_1 is empty) satisfied?

First, we build the Kripke structure, which we obtain from the reachability graph:



Then, we build atomic proposition a corresponding to "place p_1 is empty"; this can be done defining the function $f_a(\mathbf{m}) = 1$ if $\mathbf{m}[p_1] = 0$ else $f_a(\mathbf{m}) = 0$. Rewriting the formula, we obtain $\mathsf{AF}\, a \equiv \neg\mathsf{EG}\neg a$. By inspection, we see that the set of states satisfying atomic proposition $\neg a$ is $[\![\neg a]\!] = \{[20000], [11010], [11001], [10110], [10101]\}$. Since every state in the set $[\![\neg a]\!]$ can reach a state in $[\![\neg a]\!]$ in one step, we have that the expression $\mathsf{EG}(\neg a)$ is satisfied by the set of states

$[\![\mathsf{EG}(\neg a)]\!] = [\![\neg a]\!]$. Finally, taking the complement of $[\![\mathsf{EG}(\neg a)]\!]$, we obtain the set of states satisfying AF(place p_1 is empty), which is the set

$$[\![\mathsf{AF}(\text{place } p_1 \text{ is empty})]\!] = \{[02020], [02011], [02002], [01120], [01111], [01102], [00220], [00211], [00202]\}.$$

Since this set does not contain the initial marking, we conclude that the Petri net does *not* satisfy AF(place p_1 is empty).

6.4 Dealing with state explosion

Researchers have developed several methods to deal with the state explosion problem. The methods discussed below are presented only briefly, enough to give the basic idea. References are listed as a starting point for further study.

6.4.1 Partial order reduction

The idea of partial order reduction[1] is to reduce the size of the reachability graph or Kripke structure generated from a high-level model by exploiting commutativity of concurrently firable transitions.

In terms of a Petri net, suppose transitions t_1 and t_2 enjoy these two properties:

- If both t_1 and t_2 are enabled, firing one does not disable the other.
- In any marking $\mathbf{m}$ that enables both t_1 and t_2, firing t_1 and then t_2 reaches the same marking as firing t_2 and then t_1. More formally, $\mathbf{m} \xrightarrow{t_1} \mathbf{m}' \xrightarrow{t_2} \mathbf{n}$ and $\mathbf{m} \xrightarrow{t_2} \mathbf{m}'' \xrightarrow{t_1} \mathbf{n}$ for some intermediate markings $\mathbf{m}'$ and $\mathbf{m}''$.

We can extend this to sets of more than two transitions.

Depending on the property of interest to verify (which is usually restricted, for example it cannot contain X operators), it is not necessary to examine *all* possible execution orders. There are different techniques for this.

For example, "Model checking using representatives" defines an equivalence relation $\sim$ between infinite sequences such that, for the formula Φ of interest,

$$\sigma \sim \sigma' \quad \rightarrow \quad (\text{both } \sigma \text{ and } \sigma' \text{ satisfy } \Phi) \vee (\text{neither } \sigma \text{ nor } \sigma' \text{ satisfy } \Phi)$$

If there are several equivalent sequences (according to $\sim$), we only need to explore one *representative* sequence. In other words, we choose a sequence σ_r to explore, and do not need to explore any other σ with $\sigma \sim \sigma_r$. This allows us to eliminate states from the Kripke structure:

> a state not belonging to any representative sequence does not need to be generated.

This approach:

- Is still *explicit* (it explores one state at a time).
- Can sometimes reduce the number of states "enough" to be useful. For example, a reduction from 10^{12} to 10^8 can make a problem tractable, but a reduction from 10^{60} to 10^{40} will not help an explicit generation algorithm.

[1] See for example

- Peled, Doron. "All from one, one for all: model checking using representatives". Proceedings of CAV, 1993.
- Valmari, Antti. "Stubborn sets for reduced state space generation". Advances in Petri Nets 1990.

6.4.2 Symmetry reductions

6.4.3 Abstraction / refinement

6.4.4 On-the-fly model checking

6.4.5 Bounded model checking

6.4.6 Clever data structures

See next chapter.

Chapter 7

Decision diagrams

As discussed earlier, a high–level model can produce an extremely large reachability graph, easily containing millions or billions of states, or even much more. One way to combat this problem is to employ techniques that can handle large numbers of states. *Decision diagrams* are one such technique. We will start with the special case of *binary* decision diagrams.

7.1 Boolean functions

We consider functions over L boolean variables $\{x_L, ..., x_1\}$, i.e., $f : \mathbb{B}^L \rightarrow \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$. Let $f_{x_i=v}$ denote the function that results from forcing x_i to have value $= v$ and simplifying. Note that the resulting function will not depend on variable x_i, but for simplicity we will still consider it a function over L variables. Formally,

$$f_{x_i=v}(x_L, ..., x_1) \;=\; f(x_L, ..., x_{i+1}, v, x_{i-1}, ..., 1).$$

Given the two functions $f_{x_i=0}$ and $f_{x_i=1}$, we can reconstruct the original function f using *Boole's expansion theorem*, also known as *Shannon's expansion*:

$$f(x_L, ..., x_1) \;=\; (x_i \wedge f_{x_i=1}(x_L, ..., x_1)) \;\vee\; (\neg x_1 \wedge f_{x_i=0}(x_L, ..., x_1)).$$

Functions $f_{x_i=1}$ and $f_{x_i=0}$ are sometimes called the positive and negative *Shannon's cofactors*, or simply the *cofactors*, of f with respect to x_i.

Example 7.1 Suppose we have a function $f(x_3, x_2, x_1) = x_3 \wedge x_2 \;\vee\; \neg x_1$. Then

$$\begin{aligned} f_{x_2=0}(x_3, x_2, x_1) &= f(x_3, 0, x_1) \\ &= x_3 \wedge 0 \;\vee\; \neg x_1 \\ &= \neg x_1 \end{aligned}$$

and

$$\begin{aligned} f_{x_2=1}(x_3, x_2, x_1) &= f(x_3, 1, x_1) \\ &= x_3 \wedge 1 \;\vee\; \neg x_1 \\ &= x_3 \;\vee\; \neg x_1 \end{aligned}$$

Finally,

$$\begin{aligned} f(x_3, x_2, x_1) &= (x_2 \wedge f_{x_2=1}(x_3, x_2, x_1)) \vee (\neg x_2 \wedge f_{x_2=0}(x_3, x_2, x_1)) \\ &= (x_2 \wedge (x_3 \ \vee \ \neg x_1)) \vee (\neg x_2 \wedge \neg x_1) \\ &= x_3 \wedge x_2 \ \vee \ x_2 \wedge \neg x_1 \ \vee \ \neg x_2 \wedge \neg x_1 \\ &= x_3 \wedge x_2 \ \vee \ (x_2 \vee \neg x_2) \wedge \neg x_1 \\ &= x_3 \wedge x_2 \ \vee \ \neg x_1 \end{aligned}$$

Definition 7.1 Two functions $f, g : \mathbb{B}^L \ \rightarrow \ \mathbb{B}$ are equivalent, written $f \equiv g$, if

$$f(v_L, ..., v_1) = g(v_L, ..., v_1), \quad \forall (v_L, ..., v_1) \in \mathbb{B}^L.$$

Property 7.2 If f and g are boolean functions over the same set of variables $x_L, ..., x_1$, then for any variable x_i and value $b \in \mathbb{B}$,

$$f_{x_i=b} \not\equiv g_{x_i=b} \quad \rightarrow \quad f \not\equiv g$$

Proof: If $f_{x_i=b} \not\equiv g_{x_i=b}$, then $\exists v_L, ..., v_1 \in \mathbb{B}^L$ such that $f_{x_i=b}(v_L, ..., v_1) \neq g_{x_i=b}(v_L, ..., v_1)$. But then

$$f(v_L, ..., v_{i+1}, b, v_{i-1}, ..., v_1) = f_{x_i=b}(v_L, ..., v_1) \neq g_{x_i=b}(v_L, ..., v_1) = g(v_L, ..., v_{i+1}, b, v_{i-1}, ..., v_1)$$

which implies $f \not\equiv g$.

7.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) over variables $\{x_L, ..., x_1\}$ is a directed acyclic graph (DAG) with two types of nodes:

- **Non-terminal nodes**. Each non-terminal node p:
  - Is labeled with one of the variables x_k in $\{x_L, ..., x_1\}$; we write $p.var = x_k$.
  - Has two outgoing edges, labeled with 0 and 1, pointing to children $p[0]$ and $p[1]$.
- **Terminal nodes**. These are nodes with no outgoing edges. There can only be (up to) two terminal nodes, **0** and **1**.

Finally, each path from a *root* node (with no incoming edges) and a terminal node cannot traverse multiple nodes labeled with the same variable.

Top down encoding

Each BDD node (and the entire graph rooted at that node) encodes *one* boolean function

$$f : \mathbb{B}^L \ \rightarrow \ \mathbb{B}.$$

Equivalently, each node (and the graph beneath it) encodes some expression over the boolean variables. The function f_p encoded by non-terminal node p labeled with x_k can be evaluated recusively as follows, assuming $x_L = i_L, ..., x_1 = i_1$:

$$f_p(i_L, ..., i_1) = \begin{cases} 0 & \text{if } p = \mathbf{0}, \text{ thus } k = 0 \\ 1 & \text{if } p = \mathbf{1}, \text{ thus } k = 0 \\ f_{p[i_k]}(i_L, ..., i_1) & \text{otherwise, thus } k > 0 \end{cases}.$$

We can then say that the terminal nodes **0** and **1** encode constant functions, since $f_{\mathbf{0}}$ is the constant function 0 and $f_{\mathbf{1}}$ is the constant function 1, while, if p is a non-terminal node,

$$\begin{array}{lll} p[0] & \text{encodes} & f_{p[0]} = f_p(x_L, ..., x_{k+1}, 0, x_{k-1}, ..., x_1) \\ p[1] & \text{encodes} & f_{p[1]} = f_p(x_L, ..., x_{k+1}, 1, x_{k-1}, ..., x_1) \end{array}$$

or, in other words, the 0-child of p corresponds to fixing variable x_k to value 0, and the 1-child of p corresponds to fixing variable x_k to value 1.

Bottom up encoding

Another way to think about the function encoded by a node p is as follows. Terminal nodes encode the constant functions:

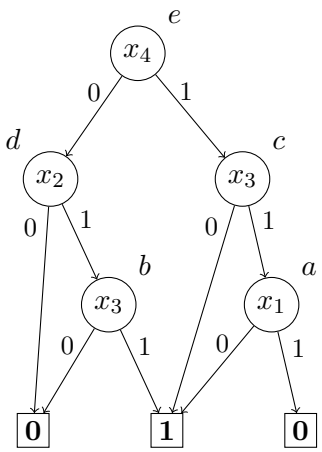
$$\begin{array}{lll} \text{node } \mathbf{0} & \text{encodes} & f_{\mathbf{0}}(x_L, ..., x_1) = 0 \\ \text{node } \mathbf{1} & \text{encodes} & f_{\mathbf{1}}(x_L, ..., x_1) = 1 \end{array}$$

while we can determine the function encoded by a non-terminal node p (and its subgraph) by applying Shannon's expansion. If $p[0]$ encodes $f_{p[0]}$ and $p[1]$ encodes $f_{p[1]}$, then node p encodes

$$f_p(x_L, ..., x_1) \;=\; (x_k \wedge f_{p[1]}(x_L, ..., x_1)) \;\vee\; (\neg x_k \wedge f_{p[0]}(x_L, ..., x_1))$$

where $x_k = p.var$.

Example 7.2 Consider the BDD shown below. In the figures, terminal nodes are drawn as rectangles (and may be repeated for clarity), non-terminal nodes are drawn as circles, with its associated variable shown inside. Each edge is numbered to indicate if it is the 0-child or 1-child of a node (alternatively, in the, literature the two edges are drawn with a dashed or solid line, respectively). We show names for each node (e, d, c, b, and a) just to be able to refer to them in our discussion.



x_4	x_3	x_2	x_1	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

On the right is the truth table for the function f encoded by node e and its subgraph (the entire BDD), which can be verified by checking each path through the BDD. For example, starting at node e, choosing $x_4 = 0$ takes us to node d, choosing $x_2 = 1$ takes us to node b, and choosing $x_3 = 1$ takes us to terminal node 1. Thus, in the truth table, we find that, when $x_4 = 0$, $x_3 = 1$, and $x_2 = 1$, function f returns 1, regardless of the value of x_1.

We can determine f_e, the function encoded by node e, bottom up using Shannon's expanson:

$$\begin{array}{lclcl} f_a &=& \neg x_1 \wedge 1 \;\vee\; x_1 \wedge 0 &=& \neg x_1 \\ f_b &=& \neg x_3 \wedge 0 \;\vee\; x_3 \wedge 1 &=& x_3 \\ f_c &=& \neg x_3 \wedge 1 \;\vee\; x_3 \wedge f_a &=& \neg x_3 \vee \neg x_1 \\ f_d &=& \neg x_2 \wedge 0 \;\vee\; x_2 \wedge f_b &=& x_3 \wedge x_2 \\ f_e &=& \neg x_4 \wedge f_d \;\vee\; x_4 \wedge f_c &=& \neg x_4 \wedge x_3 \wedge x_2 \;\vee\; x_4 \wedge (\neg x_3 \vee \neg x_1) \end{array}$$

7.2.1 Ordered BDDs

A BDD is *ordered*, on OBDD, if every path through the BDD visits variables attached to nodes in the same order. We assume that variables are ordered with x_L at the top, down to x_1 at the bottom.

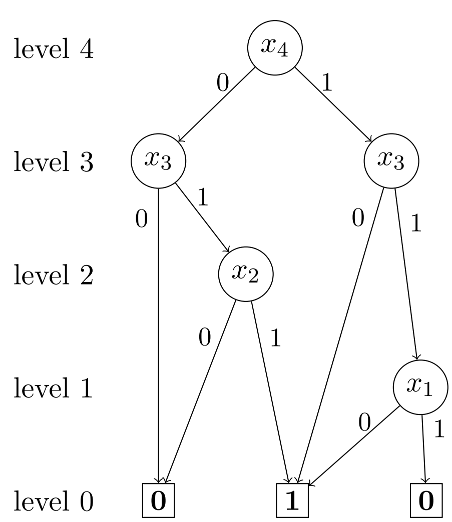
In an OBDD, for any non-terminal node p we say $p.lvl = k$ iff $p.var = x_k$ and that $q.lvl = 0$ for any terminal node q. We also say that the OBDD is an L-level OBDD. Then, an OBDD satisfies the following property.

Property 7.3 For any non-terminal node p,

$$p.lvl > p[0].lvl \qquad \text{and} \qquad p.lvl > p[1].lvl$$

Thus, every node is at a higher level than its children. From now on, we will assume that all decision diagrams are ordered.

Example 7.3 We can reorder the variables of the BDD in Example 7.2 to obtain the BDD shown below, with level information shown. In general, rearranging the variable order *does not* preserve the shape of the BDD.



7.2.2 Duplicate nodes

Definition 7.4 Non-terminal nodes p and q are *duplicates* if

1. $p.var = q.var$, and
2. $p[0] = q[0]$, and
3. $p[1] = q[1]$.

Property 7.5 Duplicate nodes encode the same function.
Proof: If p and q are duplicate non-terminals, then $p.var = q.var = x_k$ for some k, and $p[0] = q[0]$ and $p[1] = q[1]$. But then

$$\begin{aligned} f_p(x_L, ..., x_1) &= \neg x_k \wedge f_{p[0]}(x_L, ..., x_1) \quad \vee \quad x_k \wedge f_{p[1]}(x_L, ..., x_1) \\ &= \neg x_k \wedge f_{q[0]}(x_L, ..., x_1) \quad \vee \quad x_k \wedge f_{q[1]}(x_L, ..., x_1) \\ &= f_q(x_L, ..., x_1) \end{aligned}$$

If nodes p and q are duplicates in a BDD, there is no need to store both nodes. It is sufficient to keep (say) p and discard q, by redirecting any incoming edge to q to p instead.

7.3 Quasi-reduced BDDs

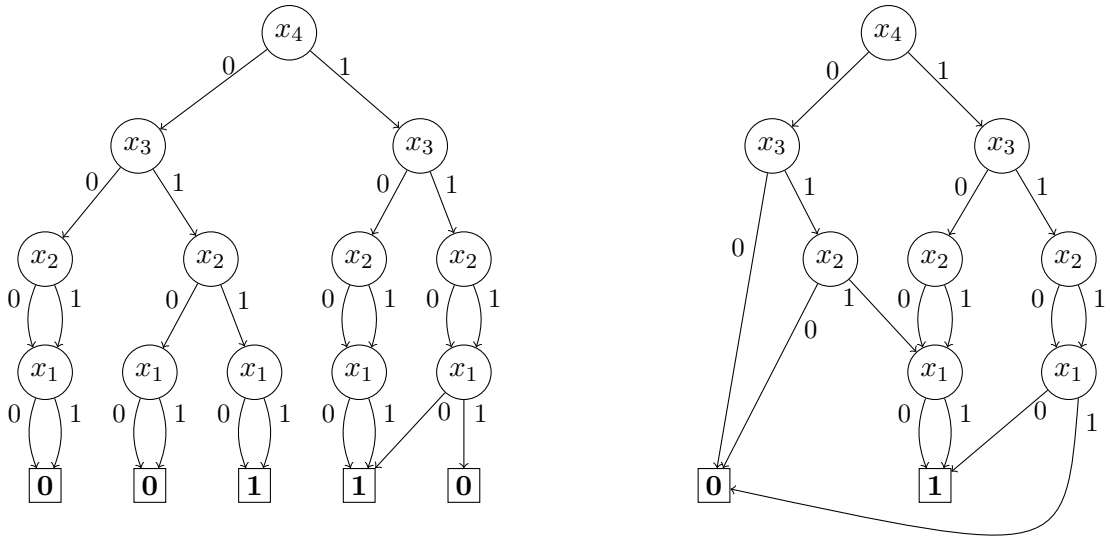
An (ordered) BDD is a *quasi reduced* BDD (QRBDD) if

- It contains no duplicate nodes.
- For every non-terminal node p, each child q satisfies $q.lvl = p.lvl - 1$ or $q.lvl = 0$, i.e., node q must be a node at the level below, or a terminal.
- The children of a non-terminal node cannot both be **0**.
- Root nodes are either **0** or non-terminal nodes at level L.

Note that a QRBDD has no "long edges", except to terminal node 0. Also note that any (ordered) BDD can be transformed into a QRBDD by adding nodes so that the level requirement is satisfied, and then eliminating duplicate nodes.

NOTE: an alternative definition of QRBDDs forbids *all* long edges, even those to terminal **0**, thus requires a chain of L non-terminal nodes also to encode the constant 0 (as the original definition already does for the constant 1).

Example 7.4 The BDD from Example 7.3 can be converted into a QRBDD by first adding nodes along each long edge (left graph) and then eliminating duplicate nodes (right graph).



7.3.1 Representing sets

We can represent any set $\mathcal{S} \subseteq \mathbb{B}^L$ using a BDD p over L variables for the *indicator function* of $\mathcal{S}$.

$$f_p(x_L, ..., x_1) = 1 \quad \text{iff} \quad (x_L, ..., x_1) \in \mathcal{S}.$$

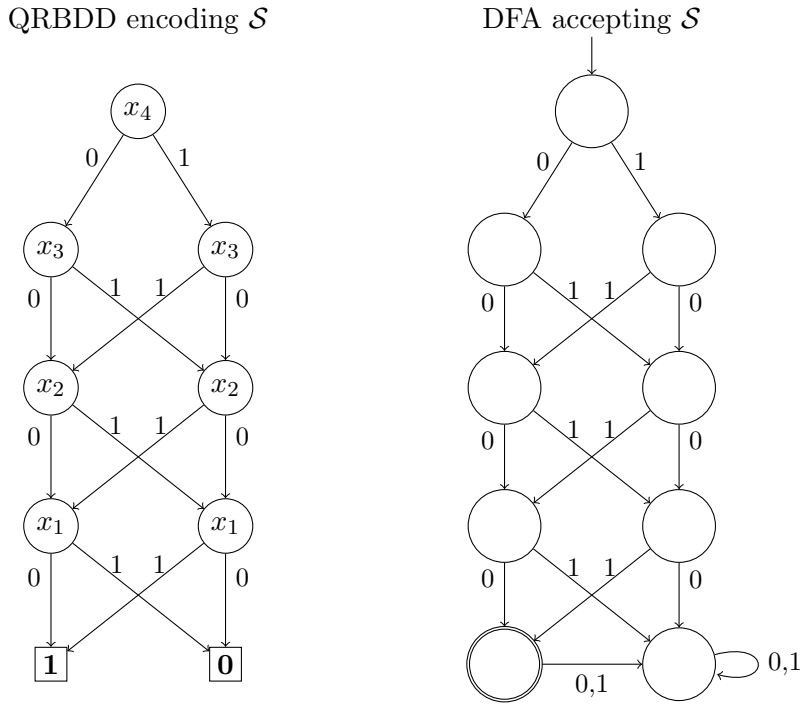
In other words, all and only the paths in the BDD that reach terminal node **1** correspond to variable assignments that cause the function to return 1, thus belong to the set.

Example 7.5 Consider the set

$$\mathcal{S} = \{(x_4, x_3, x_2, x_1) \in \mathbb{B}^4 \mid (x_4 + x_3 + x_2 + x_1) \bmod 2 = 0\}$$

I.e., the set of bit vectors of length exactly 4, with an even number of bits set.

A QRBDD encoding of set $\mathcal{S}$ is shown below, on the left. Except for level 4, each level has exactly two nodes; the left node means "an even number of bits has been seen so far" and the right node means "an odd number of bits has been seen so far". A (minimized) DFA A such that $L(A) = \mathcal{S}$ is shown below, on the right.



As this example illustrates, for the special case of fixed-length strings of bits, QRBDDs are (almost!) isomorphic to minimized DFAs; the only difference is the terminal nodes in the QRBDD.

7.3.2 Canonicity

Property 7.6 Let p and q be nodes in the same QRBDD such that either $p.lvl = q.lvl$ or q is terminal node **0**. Let f_p, f_q denote the functions of the form $\mathbb{B}^L \to \mathbb{B}$ encoded by p and q. If $f_p \equiv f_q$, then $p = q$.

Proof by induction on $p.lvl$**:**

In the base case, we have $p.lvl = 0$, and so both p and q are terminal nodes. Then either

$$p = \mathbf{0} \;\rightarrow\; f_p \equiv 0 \;\rightarrow\; f_q \equiv 0 \;\rightarrow\; q = \mathbf{0}$$

or

$$p = \mathbf{1} \;\rightarrow\; f_p \equiv 1 \;\rightarrow\; f_q \equiv 1 \;\rightarrow\; q = \mathbf{1}$$

In either case, $p = q$.

Now, assume the property holds for $p.lvl \leq n$, and show it holds for $p.lvl = n + 1$. Since $p.lvl = n + 1 > 0$, it follows that p is a non-terminal node. Suppose q is terminal node **0**. Then

$$f_q \equiv 0 \;\rightarrow\; f_p \equiv 0 \;\rightarrow\; f_{p[0]} \equiv 0 \wedge f_{p[1]} \equiv 0$$

Because we have an ordered BDD, $p[0].lvl < p.lvl = n + 1$. By inductive hypothesis, $p[0] = q$. Similarly we obtain that $p[1] = q$. But such a node p is not allowed in a QRBDD, so we have a contradiction. Thus, we must have $p.lvl = q.lvl$.

Since $f_p \equiv f_q$, from Property 7.2 we must have $f_{p[0]} \equiv f_{q[0]}$ and $f_{p[1]} \equiv f_{q[1]}$. By inductive hypothesis, $p[0] = q[0]$ and $p[1] = q[1]$. Since a QRBDD cannot contain duplicate nodes, it follows that $p = q$.

Property 7.6 establishes that QRBDDs are a *canonical form*: given a fixed variable ordering, there cannot be two different QRBDD representations of the same function f (otherwise, it would be possible to create different nodes p and q, both encoding f). This has important practical implications:

- Regardless of the algorithm used to build a QRBDD for a particular function f, we can be sure that the same QRBDD is obtained. In other words, the QRBDD obtained depends only on the encoded function, and not the choice of algorithm. (This applies only to the "final" QRBDD. Different algorithms may need different numbers of "intermediate" QRBDDs of vastly different sizes.)
- If we have two functions, f and g, encoded by nodes in a single QRBDD, checking if $f \equiv g$ is trivial: they are equivalent if and only if they are encoded by the same node.
- We can avoid unnecessary duplicate computations (discussed next): checking if a computation has been done already for a function (or pair of functions) is equivalent to checking if the computation has been done already for a node (or pair of nodes).

7.3.3 Operations on QRBDDs

We now study operations to build QRBDDs. An important process to discuss is *how to reduce the graph*, or in other words, how to we ensure that the graph does not contain duplicates. Normally the following is done:

- The graph is kept reduced (i.e., duplicate free) at "all" times, except while new nodes are under construction.
- Once a new temporary node has been built, we do a "reduction" operation on that node. This means checking that both children are not 0 (which can be considered a duplicate of node 0), and checking for a duplicate node already in the graph.
- If a duplicate is present, then discard the temporary node and use the existing equivalent one.
- If a duplicate is not present, then add the temporary node to the graph, and to the data structure used to detect duplicate nodes.
- Usually, a hash table (called the *unique table* in the literature) is used to detect duplicate nodes, where the hash function uses the variable of the node and the children pointers of the node as search key, and returns the pointer of the found node, if it exists. Terminal nodes **0** and **1** can be handled separately (e.g., as "special" pointer values).

So then how do we build a QRBDD?

1. If we know the shape of the QRBDD, then just build the graph, checking for duplicates as nodes are built. In practice, this works only for simple patterns.
2. Build up trivial functions and then combine them with operators (discussed below).

Note that, unlike most traditional data structures, nodes in a decision diagram are almost never modified "in place". Because any decision diagram node can have several parents, and a single QRBDD graph is used to encode several functions, a given node in the graph may be shared by

more than one "root" function. Thus, modifying one node might affect more than one function. As such, what is typically done instead is to build new decision diagram nodes (instead of modifying them) and create a new "root" function. Old nodes may then become disconnected, and can be removed (various ways to detect this situation exist).

Most operations are done recursively, by simultaneously traversing the operand graphs. We present several useful operations.

AND

Suppose we want to implement an "AND" operation. More formally, suppose we have functions f and g already encoded in a QRBDD, and want to find or build (as necessary) nodes in the QRBDD to encode the "element-wise and" function h, satisfying

$$h(x_L, ..., x_1) \;=\; f(x_L, ..., x_1) \;\wedge\; g(x_L, ..., x_1).$$

If f and g are indicator functions for sets $\mathcal{X}_f$ and $\mathcal{X}_g$, then we are determining $\mathcal{X}_h = \mathcal{X}_f \cap \mathcal{X}_g$.

To see how this can be done recursively, we use Shannon's expanson:

$$\begin{aligned}
h(x_L, ..., x_1) &= f(x_L, ..., x_1) \;\wedge\; g(x_L, ..., x_1) \\
&= ((\neg x_k \wedge f_{x_k=0}) \vee (x_k \wedge f_{x_k=1})) \;\wedge\; ((\neg x_k \wedge g_{x_k=0}) \vee (x_k \wedge g_{x_k=1})) \\
&= (\neg x_k \wedge f_{x_k=0} \wedge \neg x_k \wedge g_{x_k=0}) \;\vee\; (\neg x_k \wedge f_{x_k=0} \wedge x_k \wedge g_{x_k=1}) \;\vee \\
&\quad (x_k \wedge f_{x_k=1} \wedge \neg x_k \wedge g_{x_k=0}) \;\vee\; (x_k \wedge f_{x_k=1} \wedge x_k \wedge g_{x_k=1}) \\
&= (\neg x_k \wedge f_{x_k=0} \wedge \neg x_k \wedge g_{x_k=0}) \;\vee\; (x_k \wedge f_{x_k=1} \wedge x_k \wedge g_{x_k=1}) \\
&= (\neg x_k \wedge (f_{x_k=0} \wedge g_{x_k=0})) \;\vee\; (x_k \wedge (f_{x_k=1} \wedge g_{x_k=1}))
\end{aligned}$$

Therefore, we have

$$\begin{aligned}
h_{x_k=0}(x_L, ..., x_1) &\equiv f_{x_k=0}(x_L, ..., x_1) \wedge g_{x_k=0}(x_L, ..., x_1) \\
h_{x_k=1}(x_L, ..., x_1) &\equiv f_{x_k=1}(x_L, ..., x_1) \wedge g_{x_k=1}(x_L, ..., x_1)
\end{aligned}$$

and we can proceed recursively from level L, corresponding to x_L, downward. This gives us the following algorithm. In the algorithm, reduce(p) checks a temporary node p. If both children are $\mathbf{0}$, it discards p and returns $\mathbf{0}$. If p is a duplicate of an existing node p' (discovered using the unique table), then it discards p and returns p'. Otherwise, it adds p to the unique table and returns p.

Algorithm 7.1 AND operation for QRBDDs

```
qrbdd AND(qrbdd f, qrbdd g) {
    if (f = 0) ∨ (g = 0) return 0;
    if f = g return f;        // includes the case f = g = 1
    h ← new temporary node at same level as f and g;
    h[0] ← AND(f[0], g[0]);
    h[1] ← AND(f[1], g[1]);
    h' ← reduce(h);
    return h';
}
```

What is the worst-case complexity of Algorithm 7.1?

- Each recursive call for nodes at level k generates two recursive calls for nodes at level $k-1$.
- The number of recursive calls doubles at each level.
- If there are L levels, there will be up to 2^L recursive calls at level 0.

This is bad! We can do better when we realize the following:

- Each QRBDD node (except those at level L) can have several incoming edges.
- This means we might generate the same recursive call several times.
- We can improve complexity using *memoization*: store the result of each recursive call in a *compute table* (or *cache*), and if we see the same recursive call again, use the already-computed result in the compute table. Typically, a hash table is used for this, with search key consisting of the operator (unless we use a different cache for each operator) and the pointers of the operands, and returning the pointer to the node storing the result, if previously computed).

This gives us the following improved algorithm.

Algorithm 7.2 AND operation for QRBDDs, using a compute table

```
qrbdd AND(qrbdd f, qrbdd g) {
    if (f = 0) ∨ (g = 0) return 0;
    if f = g return f;
    if ∃h such that ((f, ∧, g), h) ∈ CT return h;
    h ← new temporary node at same level as f and g;
    h[0] ← AND(f[0], g[0]);
    h[1] ← AND(f[1], g[1]);
    h' ← reduce(h);
    Add ((f, ∧, g), h') to CT;
    return h';
}
```

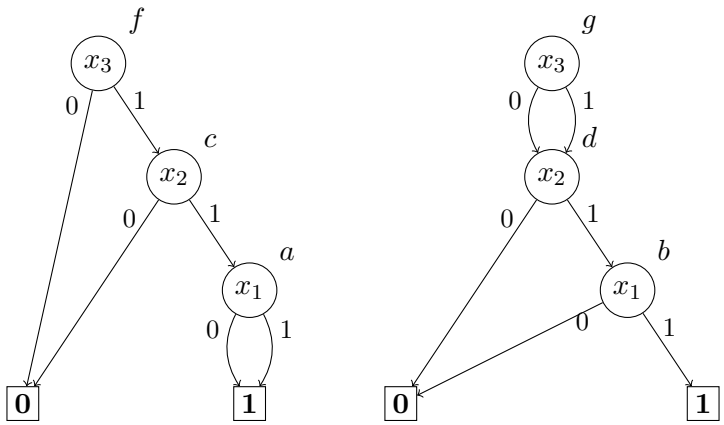
What is the worst-case complexity of Algorithm 7.2? This is based on the observation that work is done only for the *distinct* recursive calls. The number of recursive calls for nodes at level k is bounded by the total number of QRBDD nodes in f at level k, multiplied by the total number of QRBDD nodes in g at level k. If we let $|f|_k$ denote the number of QRBDD nodes at level k reachable from node f and $|f|$ denote the total number of QRBDD nodes reachable from node f, then the total number of distinct recursive calls is bounded by

$$\sum_{L \geq k \geq 1} |f|_k \cdot |g|_k \quad \leq \quad |f| \cdot |g|.$$

Thus, with the compute table, the complexity depends on the sizes of input QRBDDs, which can be much smaller that 2^L.

QUESTION: can you suggest a simple but effective improvement to Algorithm 7.2?

Example 7.6 For the QRBDD below (where nodes are named, and some terminal nodes are duplicated for convenience), build the result of AND(f, g).



AND(f, g):

- h: temporary node at level 3
- $h[0]$: AND($f[0]$, $g[0]$) = AND(0, d) = 0
- $h[1]$: AND($f[1]$, $g[1]$) = AND(c, d)

  AND(c, d):

  - i: temporary node at level 2
  - $i[0]$: AND($c[0]$, $d[0]$) = AND(0, 0) = 0
  - $i[1]$: AND($c[1]$, $d[1]$) = AND(a, b)

    AND(a, b):

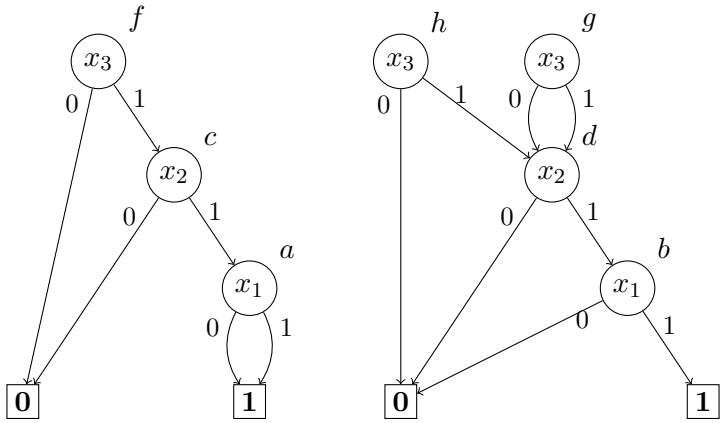
    * j: temporary node at level 1
    * $j[0]$: AND($a[0]$, $b[0]$) = AND(1, 0) = 0
    * $j[1]$: AND($a[1]$, $b[1]$) = AND(1, 1) = 1
    * j is a duplicate of b

    AND(a, b) = b
  - $i[1]$: AND($c[1]$, $d[1]$) = AND(a, b) = b
  - i is a duplicate of d

  AND(c, d) = d

- $h[1]$: AND($f[1]$, $g[1]$) = AND(c, d) = d
- h is a new node

After the operation, we have the following QRBDD:



To verify this, note that f encodes the function $x_3 \wedge x_2$, g encodes the function $x_2 \wedge x_1$, and h encodes the function $(x_3 \wedge x_2) \wedge (x_2 \wedge x_1) = x_3 \wedge x_2 \wedge x_1$.

OR

Now, suppose we have functions f and g already encoded in a QRBDD, and we want to build nodes in the QRBDD as needed to encode the "element-wise or" function

$$h(x_L, ..., x_1) \;=\; f(x_L, ..., x_1) \;\vee\; g(x_L, ..., x_1)$$

If f and g are indicator functions for sets $\mathcal{X}_f$ and $\mathcal{X}_g$, then we are determining $\mathcal{X}_h = \mathcal{X}_f \cup \mathcal{X}_g$. Again, we can use Shannon's expansion:

$$\begin{aligned} h(x_L, ..., x_1) &= f(x_L, ..., x_1) \;\vee\; g(x_L, ..., x_1) \\ &= ((\neg x_k \wedge f_{x_k=0}) \vee (x_k \wedge f_{x_k=1})) \;\vee\; ((\neg x_k \wedge g_{x_k=0}) \vee (x_k \wedge g_{x_k=1})) \\ &= (\neg x_k \wedge (f_{x_k=0} \vee g_{x_k=0})) \;\vee\; (x_k \wedge (f_{x_k=1} \vee g_{x_k=1})) \end{aligned}$$

Therefore, we have

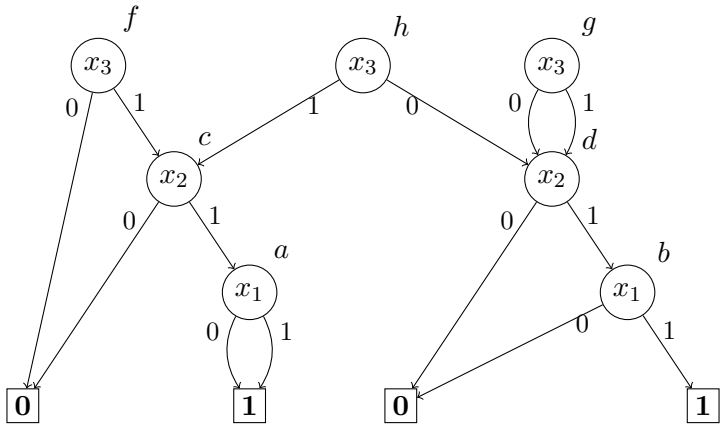
$$\begin{aligned} h_{x_k=0}(x_L, ..., x_1) &\equiv f_{x_k=0}(x_L, ..., x_1) \vee g_{x_k=0}(x_L, ..., x_1) \\ h_{x_k=1}(x_L, ..., x_1) &\equiv f_{x_k=1}(x_L, ..., x_1) \vee g_{x_k=1}(x_L, ..., x_1) \end{aligned}$$

and we obtain a recursive algorithm almost identical to AND except for different terminal conditions.

Algorithm 7.3 OR operation for QRBDDs, using a compute table

```
qrbdd OR(qrbdd f, qrbdd g) {
    if f = 0 return g;
    if g = 0 return f;
    if f = g return f;
    if ∃h such that ((f, ∨, g), h) ∈ CT return h;
    h ← new temporary node at same level as f and g;
    h[0] ← OR(f[0], g[0]);
    h[1] ← OR(f[1], g[1]);
    h' ← reduce(h);
    Add ((f, ∨, g), h') to CT;
    return h';
}
```

Example 7.7 For the QRBDD below (where nodes are named, and some terminal nodes are duplicated for convenience), it can be shown that $\mathsf{OR}(f, g) = h$.



NOT

Suppose we have function f already encoded in a QRBDD, and we want to build nodes in the QRBDD as needed to encode function

$$h(x_L, ..., x_1) \;=\; \neg f(x_L, ..., x_1)$$

If f is the indicator function for set $\mathcal{X}_f$, we are determining $\mathcal{X}_h = \mathbb{B}^L \setminus \mathcal{X}_f$, the complement of $\mathcal{X}_f$.

Using Shannon's expansion is unpleasant because of De Morgan's laws, but everything (eventually) works out nicely:

$$\begin{aligned}
h(x_L, ..., x_1) &= \neg f(x_L, ..., x_1) \\
&= \neg\left((\neg x_k \wedge f_{x_k=0}) \vee (x_k \wedge f_{x_k=1})\right) \\
&= \neg(\neg x_k \wedge f_{x_k=0}) \;\wedge\; \neg(x_k \wedge f_{x_k=1}) \\
&= (x_k \vee \neg f_{x_k=0}) \;\wedge\; (\neg x_k \vee \neg f_{x_k=1}) \\
&= x_k \wedge \neg x_k \quad \vee \quad x_k \wedge \neg f_{x_k=1} \quad \vee \quad \neg x_k \wedge \neg f_{x_k=0} \quad \vee \quad \neg f_{x_k=0} \wedge \neg f_{x_k=1}
\end{aligned}$$

Now, in the last equation, $x_k \wedge \neg x_k$ is always false and can be discarded. More difficult to see is that $\neg f_{x_k=0} \wedge \neg f_{x_k=1}$ is the "common portion" of $\neg f_{x_k=0}$ and $\neg f_{x_k=1}$ but this is already included in the other two terms. (For a more satisfactory derivation, one can show that two expressions a and b are equivalent by showing that $(a \wedge b) \vee (\neg a \wedge \neg b)$ is always true.) The result is

$$h(x_L, ..., x_1) \;=\; x_k \wedge \neg f_{x_k=1} \quad \vee \quad \neg x_k \wedge \neg f_{x_k=0}$$

which again gives us

$$\begin{aligned}
h_{x_k=0}(x_L, ..., x_1) &\equiv \neg f_{x_k=0}(x_L, ..., x_1) \\
h_{x_k=1}(x_L, ..., x_1) &\equiv \neg f_{x_k=1}(x_L, ..., x_1)
\end{aligned}$$

and a recursive procedure that is similar to the ones used for AND and OR.

Algorithm 7.4 NOT operation for QRBDDs, using a compute table

```
qrbdd NOT(qrbdd f) {
    if f = 0 return 1;
    if f = 1 return 0;
    if ∃h such that ((¬, f), h) ∈ CT return h;
    h ← new temporary node at same level as f;
    h[0] ← NOT(f[0]);
    h[1] ← NOT(f[1]);
    h' ← reduce(h);
    Add ((¬, f), h') to CT;
    return h';
}
```

However, Algorithm 7.4 has a problem: line 1 could create a long edge to terminal node **1**, which is not allowed. This can be fixed by inserting nodes along the long edge. Or, we can use a different type of BDD that allows long edges; we will discuss these later.

Cardinality

Given an function f encoded as a QRBDD, where f is an indicator function for set $\mathcal{X}_f$, how can we determine the cardinality of the set $\mathcal{X}_f$? Or, put another way, how can we determine the number of different variable assignments that cause f to evaluate to true?

Instead of using Shannon's expansion, we observe that for any variable x_k,

$$\#\text{ true assignments} = (\#\text{ true assignments with } x_k = 0) + (\#\text{ true assignments with } x_k = 1)$$

Similarly, we can note that the number of variable assignments that cause f to evaluate to true is equivalent to the number of paths from node f to terminal node **1**. Then we have that, for any non-terminal node p,

$$\#\text{ paths from } p \text{ to } \mathbf{1} = (\#\text{ paths from } p[0] \text{ to } \mathbf{1}) + (\#\text{ paths from } p[1] \text{ to } \mathbf{1})$$

Of course, once we know the number of paths from p to **1**, there is no need to recount. This gives us another recursive algorithm that utilizes a compute table, except that it returns an integer instead of a BDD node.

Algorithm 7.5 Cardinality operation for QRBDDs

```
integer Cardinality(qrbdd p) {
    if p = 0 return 0;
    if p = 1 return 1;
    if ∃((#, p), count) ∈ CT return count;
    count ← Cardinality(p[0]) + Cardinality(p[1]);
    Add ((#, p), count) to CT;
    return count;
}
```

Note that the cardinality is bounded by 2^L, but can easily approach this value in practice, so it is easy for this operation to overflow a 64-bit integer. There are two methods to deal with this:

1. Use a double-precision floating-point representation (i.e., a `double`) and accept the loss of precision, instead giving an approximate cardinality in some cases. Also note that the cardinality can exceed the range of a `double` (around 2^{1024}) in practice.

2. Use an arbitrary precision integer library, such as GMP, to get an exact count.

PreImage and PostImage

The last operations we need, for CTL model checking, are PreImage and PostImage. We already know how to encode a set of states (assuming boolean state variables): if we have a Petri net that can have at most one token in each place (we will see later that this is not deal-breaking restriction), then we can use an indicator function f to encode a set of markings. What about the edge relation? We need to have some kind of "compatible" encoding if we want to implement PreImage and PostImage using BDDs.

But a relation is a set of edges, and we know how to encode sets already. Formally, we want to encode a set of (source state, destination state) pairs. This can be done using an indicator function of the form $r : \mathbb{B}^L \times \mathbb{B}^L \to \mathbb{B}$ where

$$r(x_L, ..., x_1, x'_L, ..., x'_1) = 1 \text{ iff } ([x_L, ..., x_1], [x'_L, ..., x'_1]) \in \mathcal{R}$$

As done in the literature, we use unprimed variables to denote "source state" and primed variables to denote "destination state". Usually, though, it is better to "interleave" the variables, and instead use

$$r(x_L, x'_L, ..., x_2, x'_2, x_1, x'_1) = 1 \text{ iff } ([x_L, ..., x_1], [x'_L, ..., x'_1]) \in \mathcal{R}$$

because:

1. This tends to give a smaller (fewer nodes) BDD representation for r.
2. This allows for more efficient PreImage and PostImage algorithms.

Abusing notation, we say that nodes associated with x_k or x'_k are at level k or k', respectively.

To determine *PostImage*$(\mathcal{X}, \mathcal{R})$, where $\mathcal{X}$ is a set of states encoded as a QRBDD over L variables $x_L, ..., x_1$, and $\mathcal{R}$ is a set of edges between states encoded as a QRBDD over $2L$ variables $x_L, x'_L, ..., x_1, x'_1$, we could do the following steps.

1. Perform set intersection (the AND operation) on $\mathcal{X}$ and $\mathcal{R}$, ignoring the primed variables in $\mathcal{R}$. This gives us the set of *edges* that originate from some state in $\mathcal{X}$.
2. Perform set union (the OR operation) of the 0 and 1 children of all unprimed levels. This gives us the set of the destination states (regardless of the source states). This is exactly what we want for *PostImage*, except that the result is expressed using *primed* variables.
3. Relabel the primed variables as unprimed.

The above steps will work correctly and efficiently as long as the unprimed variables are in the same order in $\mathcal{X}$ and $\mathcal{R}$, and the primed variables in $\mathcal{R}$ are in the same order as the unprimed variables. In other words, $\mathcal{R}$ can use any order of variables, as long as the unprimed variables are in order $x_L, ..., x_1$ and the primed variables are in order $x'_L, ..., x'_1$.

If instead we require $\mathcal{R}$ to use the variable ordering $x_L, x'_L, ..., x_1, x'_1$, then instead of doing the above steps as three separate passes over the QRBDDs, we can do the above three steps *simultaneously* in a single pass over the QRBDDs. (Another equivalent view if it helps: imagine the QRBDD nodes in $\mathcal{X}$ as tiny, 2-element vectors, and the QRBDD pair of nodes for variables x_i, x'_i as tiny, 2×2 matrices; then the *PostImage* computation is vector-matrix multiplication done recursively by levels.) This gives us the following algorithm.

Algorithm 7.6 PostImage using QRBDDs

```
qrbdd PostImage(qrbdd x, qrbdd r) {
    • x is a set of states over L variables
    • r is a set of edges over 2L interleaved variables
    if x = 0 or r = 0 return 0;
    if x = 1 return 1;
    if ∃((x, postImage, r), y) ∈ CT return y;
    y ← new node at same level as x;
    y[0] ← OR( PostImage(x[0], r[0][0]), PostImage(x[1], r[1][0]) );
    y[1] ← OR( PostImage(x[0], r[0][1]), PostImage(x[1], r[1][1]) );
    y' ← reduce(y);
    Add ((x, postImage, r), y') to CT;
    return y';
}
```

The algorithm for *PreImage* will be similar.

7.4 Other types of decision diagrams

QRBDDs, as discussed above, have fairly simple algorithms due to the lack of "long edges". However, in practice, other types of decision diagrams are used. Below, we summarize common variations and give examples of how the operations must be modified.

7.4.1 "Fully reduced" BDDs

The perhaps most commonly-used form of decision diagrams are (fully) reduced, ordered BDDs[1]. For consistency and to avoid confusion, we refer to these as "fully reduced" BDDs.

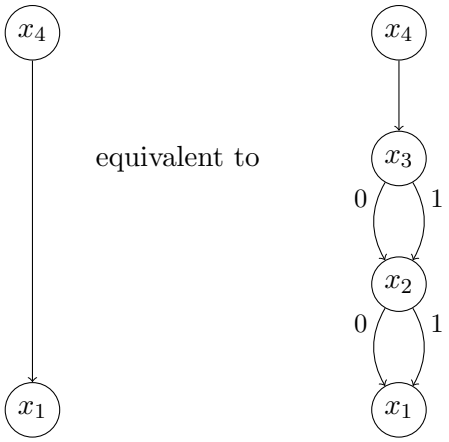
Definition 7.7 A non-terminal node p is *redundant* if $p[0] = p[1]$.

An (ordered) BDD is a *(fully) reduced* BDD (FRBDD) if

- It contains no duplicate nodes.
- It contains no redundant nodes.

Any QRBDD can be converted to a FRBDD by eliminating redundant nodes: any pointer to a redundant node p should be redirected to $p[0]$ (which by definition equals $p[1]$). Thus, a *long edge* in a FRBDD should be viewed as and edge containing a chain of implicit redundant nodes.

[1] R. Bryant. "Graph–Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, C–35 (8), August 1986, pages 677–691.



Property 7.8 If p is a redundant node with $p.var = x_k$, then the function f_p encoded by p is equal to the function encoded by its children.
Proof:

$$\begin{aligned} f_p(x_L, ..., x_1) &= (x_k \wedge f_{p[1]}(x_L, ..., x_1)) \ \vee \ (\neg x_k \wedge f_{p[0]}(x_L, ..., x_1)) \\ &= (x_k \wedge f_{p[0]}(x_L, ..., x_1)) \ \vee \ (\neg x_k \wedge f_{p[0]}(x_L, ..., x_1)) \\ &= (x_k \vee \neg x_k) \wedge f_{p[0]}(x_L, ..., x_1) \\ &= f_{p[0]}(x_L, ..., x_1) \end{aligned}$$

Canonicity

Property 7.9 Let p and q be nodes in the same FRBDD. Let f_p, f_q denote the functions encoded by p and q. If $f_p \equiv f_q$, then $p = q$.
Proof sketch: Using induction on the "top" level of p and q (i.e., induction on $\max(p.lvl, q.lvl)$), the proof is similar to that of Property 7.6, except there is an additional step: show that p and q must be at the same level, otherwise one of them will be a redundant node.

Operations

On the one hand, operations on FRBDDs can be more efficient than their QRBDD counterparts since there is no need to create redundant nodes to avoid long edges as required by the quasi-reduced property. On the other hand, FRBDDs *must* explicitly store level information in each non-terminal node, and FRBDD algorithms are slightly more complex due to long edges.

The algorithms given for QRBDDs can be modified for use with FRBDDs as follows.

- Add terminating conditions for the recursion that are not possible with QRBDDs.
- Node arguments may be at different levels; always process the higher level.
- When building a node at level k, for an argument function f encoded by node p we need to obtain its cofactors $f_{x_k=0}$ and $f_{x_k=1}$. If p is at level k, then these are given by nodes $p[0]$ and $p[1]$. If p is instead at a level below k, there is an imaginary redundant node at level k with both children pointing to p, and the cofactors are given by nodes p and p.
- As in the QRBDD case, we check for duplicate nodes using a procedure, reduce. In addition, for FRBDDs, reduce(p) also checks wheher $p[0] = p[1]$ and returns $p[0]$ if this is the case.

The AND operator, modified for use with FRBDDs, is shown below.

Algorithm 7.7 AND operation for FRBDDs

```
frbdd AND(frbdd f, frbdd g){
    if (f = 0) ∨ (g = 0) return 0;
    if f = 1 return g;
    if g = 1 return f;
    if f = g return f;
    if ∃h such that ((f, ∧, g), h) ∈ CT return h;
    k ← max(f.lvl, g.lvl);
    h ← new temporary node at level k;
    h[0] ← AND( (f.lvl = k) ? f[0] : f, (g.lvl = k) ? g[0] : g);
    h[1] ← AND( (f.lvl = k) ? f[1] : f, (g.lvl = k) ? g[1] : g);
    h' ← reduce(h);
    Add ((f, ∧, g), h') to CT;
    return h';
}
```

7.4.2 Zero–suppressed BDDs

Zero–suppressed BDDs (ZDDs)[2] are another type of BDD, with a different rule for long edges. In a ZDD, a long edge skips over a node if its 1-child points to terminal node **0**.

Definition 7.10 A non-terminal node p is a *high-zero* node if $p[1] = \mathbf{0}$.

Formally, an (ordered) BDD is a ZDD if

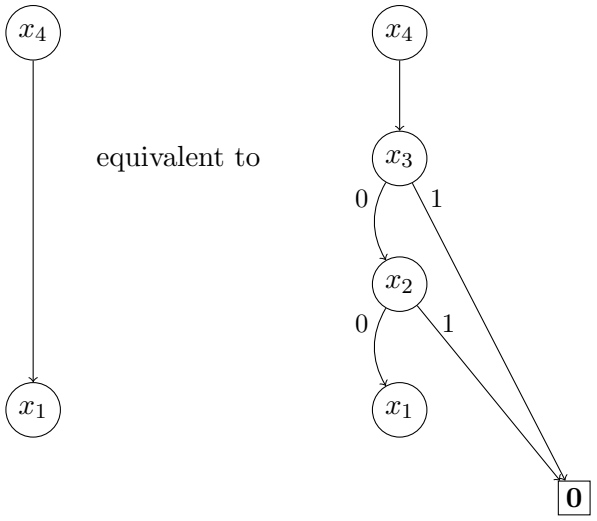
- There are no distinct duplicate nodes.
- There are no high-zero nodes.

Note that redundant nodes *must* be present in a ZDD, because long edges have a different meaning. For the same reason, the value of the function encoded by a ZDD must be defined in a different way: the function $f_p^n : \mathbb{B}^n \to \mathbb{B}$ encoded by ZDD node p at level k with respect to level $n \geq k$ is:

$$f_p^n = \begin{cases} \mathbf{1} & \text{if } n = k = 0 \text{ and } p = \mathbf{1} \\ \mathbf{0} & \text{if } p = \mathbf{0} \text{ or } n > k \wedge \exists h \in \{k+1, ..., n\},\ x_h = 1 \\ f_p^k & \text{if } n > k \wedge \forall h \in \{k+1, ..., n\},\ x_h = 0 \\ f_{p[x_k]}^{k-1} & \text{if } n = k > 0 \end{cases}.$$

It can be shown that ZDDs are a canonical representation.

[2] S. Minato. "Zero–suppressed BDDs and their applications", *Software Tools for Technology Transfer*, 3:156–160, 2001.



Operations

Care must be taken when designing operations for ZDDs. For a generic binary operation $\oplus$, if $x \oplus 0 = x$ or $x \oplus 0 = 0$, applying the operation to high-zero nodes results in a high-zero node. This means the algorithm can "skip levels" when following long edges, as was done with FRBDDs. However, if $x \oplus 0 \neq x$, applying the operation to high-zero nodes could produce a node that is not a high-zero node; for these operations, the algorithm must proceed "by levels".

As a concrete example, consider the (unary) negation operator. When applied to a high-zero node, negation will produce a node whose 1 child is to terminal node 1. An algorithm for negation, for ZDDs, is shown below. Note that it must be invoked from level $k = L$. Again, we must use a version of reduce(p) that checks both for duplicates and for $p[1] = 0$.

Algorithm 7.8 NOT operation for ZDDs

```
zdd NOT(integer k, zbdd f) {
    if k = 0 return f = 0 ? 1 : 0;
    if ∃h such that ((¬, k, f), h) ∈ CT return h;
    h ← new temporary node at level k;
    if f.lvl = k
        h[0] ← NOT(k − 1, f[0]);
        h[1] ← NOT(k − 1, f[1]);
    else
        h[0] ← NOT(k − 1, f);
        h[1] ← NOT(k − 1, f);
    endif
    h′ ← reduce(h);
    Add ((¬, k, f), h′) to CT;
    return h′;
}
```

The complexity of NOT for ZDDs is

$$\mathcal{O}(|f| \cdot L)$$

whereas for FRBDDs it is

$$\mathcal{O}(|f|)$$

because FRBDDs allow NOT to skip over long edges.

7.4.3 Identity reductions

Another type of long edge, for BDDs that encode relations, is an *identity* edge, which skips over levels x_k and x'_k when $x'_k = x_k$. If p is a node with $p.var = x_k$, $p[0].var = x'_k$, and $p[1].var = x'_k$, we say that p, $p[0]$, and $p[1]$ form an *identity pattern* if $p[0][1] = 0$, $p[1][0] = 0$, and $p[0][0] = p[1][1]$. An *identity reduced* BDD (IRBDD) is an ordered BDD over variables $x_L, x'_L, ..., x_1, x'_1$ such that:

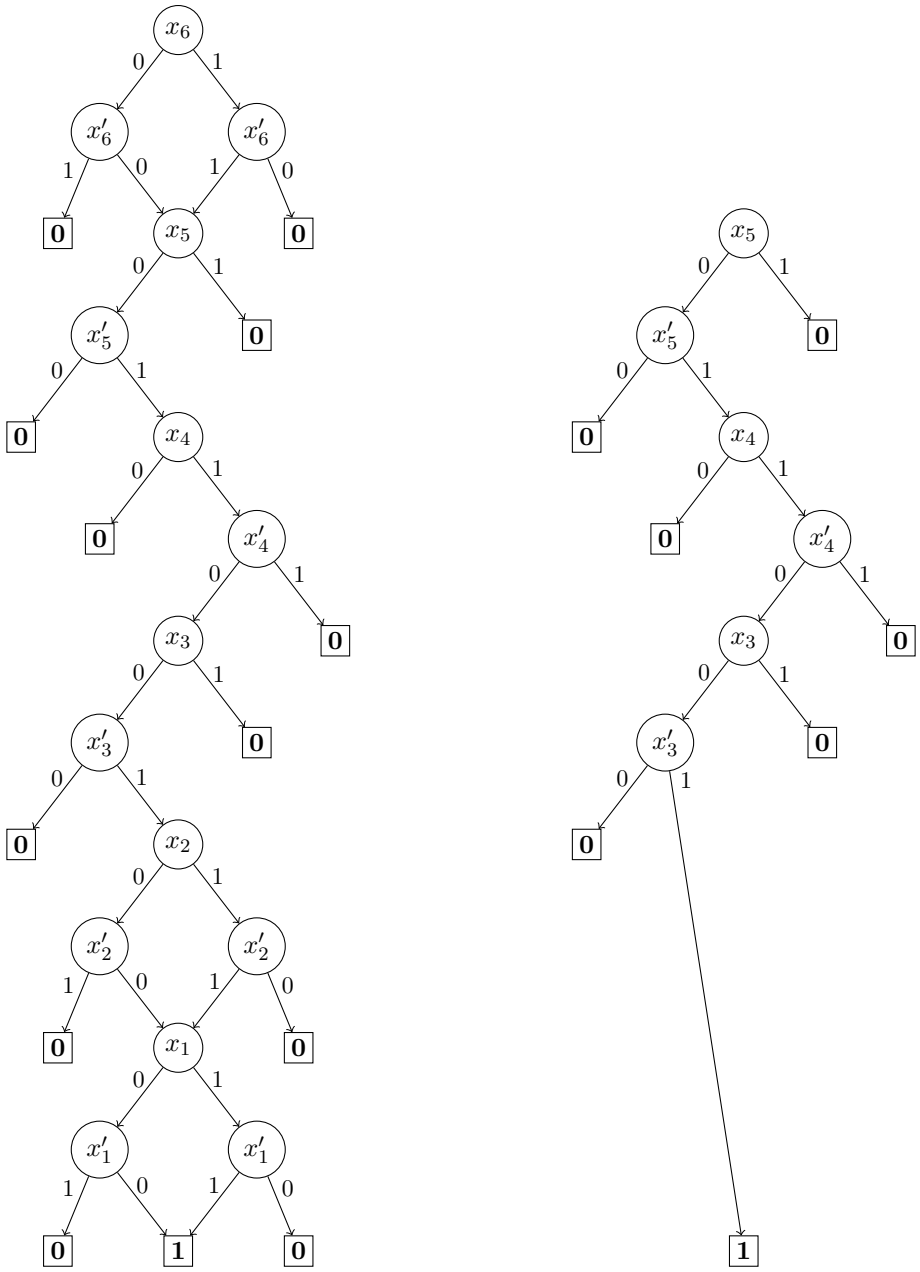
- There are no duplicate nodes.
- All edges from a non-terminal node at level k are to nodes at level k' or to terminal node **0**.
- There are no identity patterns.

It can be shown that IRBDDs are a canonical form.

Example 7.8 In a Petri net with six places, assuming all places can contain at most one token (we say that the Petri net is *safe*), if transition t has p_4 as an input place, and p_5 and p_3 as output places, the transition relation for t can be expressed as

$$(x'_6 = x_6) \wedge (x'_5 = x_5 + 1) \wedge (x'_4 = x_4 - 1) \wedge (x'_3 = x_3 + 1) \wedge (x'_2 = x_2) \wedge (x'_1 = x'_1).$$

Below, on the left, we have the QRBDD encoding this relation, while the right shows the IRBDD for this relation (terminal node **0** is drawn more than once, to make the graph more readable).



Operations

IRBDDs are efficient for *PreImage* and *PostImage* operations, as they can exploit identity patterns.

Algorithm 7.9 PostImage using IRBDDs for relations

qrbdd PostImage(qrbdd x, irbdd r) {
- x is a set of states over L variables
- r is a set of edges over $2L$ interleaved variables

if $x = \mathbf{0}$ or $r = \mathbf{0}$ **return 0**;
if $x = \mathbf{1}$ **return 1**;
if $r = \mathbf{1}$ **return** x;

```
    if ∃((x, postImage, r), y) ∈ CT return y;
    y ← new node at same level as x;
    if r.lvl = x.lvl
        y[0] ← OR( PostImage(x[0], r[0][0]), PostImage(x[1], r[1][0]) );
        y[1] ← OR( PostImage(x[0], r[0][1]), PostImage(x[1], r[1][1]) );
    else
        y[0] ← PostImage(x[0], r);
        y[1] ← PostImage(x[1], r);
    endif
    y' ← reduce(y);
    Add ((x, postImage, r), y') to CT;
    return y';
}
```

7.4.4 Multi-valued decision diagrams

What happens when we want to encode a function where variables are bounded integers, not necessarily booleans? More specifically, suppose we have a positive integer M, and want to encode a function $f : \{0, ..., M\}^L \rightarrow \mathbb{B}$. (In general, we could have a different bound M_k for each variable x_k; but for simplicity, take the maximum of all bounds and call it M.)

One possibility is to use a set of boolean variables to encode each integer variable. For example, we might use $b_{k,0}, ..., b_{k,n}$ to encode x_k using a binary encoding:

$$x_k = \sum_{j=0}^{n} 2^j \cdot b_{k,j}$$

which of course needs $n = \lceil \log_2(M+1) \rceil$ bits. Another choice is to use a "one hot" encoding using $b_{k,1}, ..., b_{k,M}$ where $b_{k,j} = 1$ iff the value of x_k is j:

$$x_k = \sum_{j=1}^{M} j \cdot b_{k,j}$$

with the restriction that at most one of $b_{k,1}, ..., b_{k,M}$ is set ($x_k = 0$ corresponds to no bits set). Then, we can use any of the BDD variations to encode our function over what are now boolean variables. ZDDs tend to be efficient when encoding sets with the "one hot" encoding: if at most one of $b_{k,1}, ..., b_{k,M}$ can be set, then the 1-child will often point to terminal node **0**.

The alternative is to instead modify our data structure, so that the non-terminal nodes can have a child for each value $0, ..., M$. This is called a multi-valued decision diagram (MDD). Then, we need to update some of our definitions to handle more than two children. Similarly, the operations can be updated to manage nodes with more than two children.

Definition 7.11 Non-terminal MDD nodes p and q are *duplicates* if

1. $p.var = q.var$, and
2. $\forall i \in \{0, ..., M\}$, $p[i] = q[i]$.

Definition 7.12 Non-terminal MDD node p is *redundant* if

1. $\forall i \in \{1, ..., M\}$, $p[i] = p[0]$.

Definition 7.13 An MDD is fully-reduced if

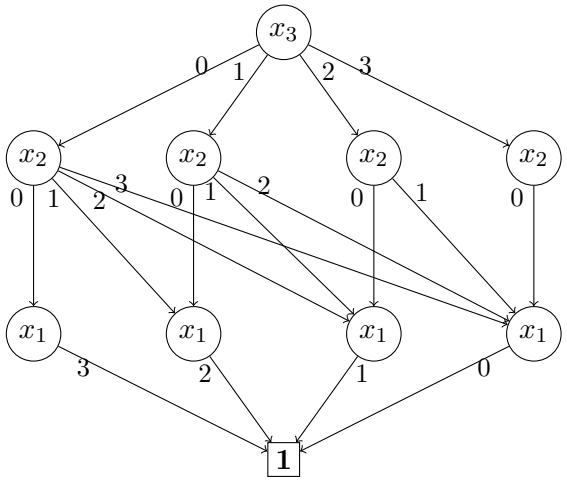
1. It contains no duplicate nodes.

2. It contains no redundant nodes.

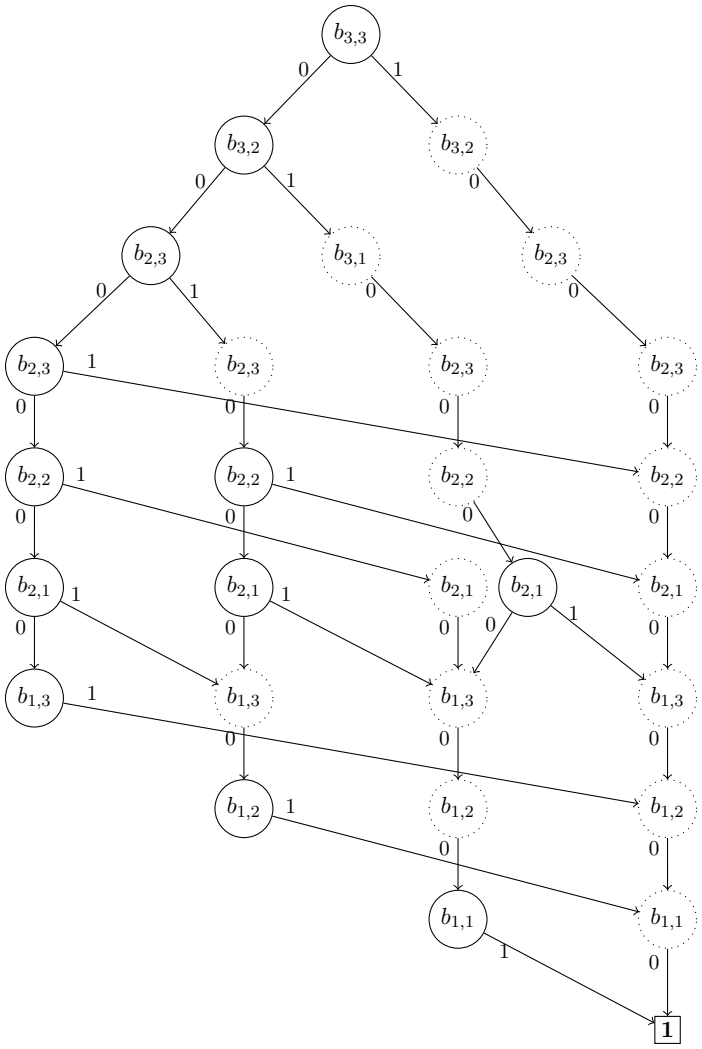
Algorithm 7.10 AND operation for (fully-reduced) MDDs

```
mdd AND(mdd f, mdd g) {
    if (f = 0) ∨ (g = 0) return 0;
    if f = 1 return g;
    if g = 1 return f;
    if f = g return f;
    if ∃h such that ((f, ∧, g), h) ∈ CT return h;
    k ← max(f.lvl, g.lvl);
    h ← new temporary node at level k;
    forall i ∈ {0, ..., M}
        h[i] ← AND( (f.lvl = k) ? f[i] : f, (g.lvl = k) ? g[i] : g);
    end for
    h' ← reduce(h);
    Add ((f, ∧, g), h') to CT;
    return h';
}
```

Example 7.9 Suppose we want to encode the set $\mathcal{X}$ with $(x_3, x_2, x_1) \in \mathcal{X}$ if and only if $x_3 + x_2 + x_1 = 3$. This can be done using the following MDD. For clarity, edges to terminal node **0** are not drawn. Note that there are four nodes at each level (except the top), to remember the value of the "sum so far".



Alternatively, we can use a "one-hot" encoding which gives us the following BDD. Again, edges to terminal node **0** are not drawn for clarity. The dotted nodes would be eliminated if we used a ZDD.



7.4.5 Complement edges

7.5 Encoding non-boolean functions

7.5.1 Multiterminal binary or multivalued decision diagrams

7.5.2 Edge values

7.6 CTL model checking with decision diagrams

The essential steps of CTL model checking using decision diagrams are as follows.

7.6.1 Initial state

STEP 1: Build a decision diagram for the initial state. Details will depend upon which type of decision diagram is used, but generally this step is trivial. A set containing a single element will be encoded by a decision diagram with a single path to terminal node **1**, where variable assignments along the path correspond to the element of the set.

7.6.2 Transition relation

STEP 2: Build the transition relation. This also can be efficient, because generally we do not care if the source states are reachable. In other words, it suffices to build a relation that gives correct outgoing edges for *any* "potential" state, not just for the "actual" (reachable) ones. Interestingly, the decision diagram encoding the potential relation will tend to be (sometimes enormously) smaller than the one encoding the actual relation, even if it encodes (sometimes enormously) more edges.

As hinted earlier, the transition relation for a Petri net can be built for each transition based on its input and output places: state variables corresponding to input places are decremented (i.e., $x'_i = x_i - 1$) and state variables corresponding to output places are incremented (i.e., $x'_j = x_j + 1$). Decision diagram patterns for these behaviors are simple enough to be built for each state variable (unprimed, primed) pair, (x_i, x'_i); we then "AND" them together, along with "no change" (i.e., $x'_k = x_k$) for places not affected by, and not affecting, the transition.

However, this assumes we know a bound on the number of tokens in each place. Is this possible?

1. Structural information of the Petri net can *sometimes* be used to discover token bounds on places.
2. There are "on-the-fly" algorithms that start with a bound (based on the initial marking) and expand the bound as needed during generation of the reachability set.

Either way, the "overall" transition relation is the union of the transition relations for each transition:

$$\mathcal{R} \;=\; \bigcup_{t \in \mathcal{T}} \mathcal{R}_t.$$

Thus, we can either:

- Combine these relations into a single, "monolithic" relation using set union (the OR operation on decision diagrams). This can sometimes be expensive.
- Use a "partitioned transition relation" (i.e., keep each of the $\mathcal{R}_t$ separate) and modify *PreImage* and *PostImage* algorithms to work when $\mathcal{R}$ is partitioned.

7.6.3 Reachable states

STEP 3: Generate the set of reachable states. Depending on the property, this is *usually* required.

Let $\mathcal{S}_d$ denote the set of states reachable in d or fewer steps from the initial state(s). Then $\mathcal{S}_0$ is the set of initial states, and we have:

$$\mathcal{S}_{d+1} \;=\; \mathcal{S}_d \cup \mathit{PostImage}(S_d, \mathcal{R})$$

Our goal is to determine the set $\mathcal{S} = \mathcal{S}_\infty$. More precisely, $\mathcal{S}$ is the least fixpoint of the function

$$f(\mathcal{X}) \;=\; \mathcal{S}_0 \cup \mathit{PostImage}(\mathcal{X}, \mathcal{R})$$

If our goal is to determine the set $\mathcal{S}$ with no other requirements, then this may be done in several different ways. Note that we may need to "un-learn" our experiences with explicit data structures. For example, the original algorithm to build $\mathcal{S}$ using decision diagrams was based on a *frontier set*, i.e., the set of states exactly at distance d from the initial states: $\mathcal{S}_d \setminus \mathcal{S}_{d-1}$.

Algorithm 7.11 Breadth-first reachability using a frontier set

$\mathcal{S} \leftarrow \mathcal{S}_0$;
$\mathcal{F} \leftarrow \mathcal{S}_0$;
while $\mathcal{F} \neq \emptyset$
  $\mathcal{G} \leftarrow \bigcup_t PostImage(\mathcal{F}, \mathcal{R}_t)$;
  $\mathcal{F} \leftarrow \mathcal{G} \setminus \mathcal{S}$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{F}$;
end while

Alternatively, we can apply PostImage to *all known states*, using the following algorithm.

Algorithm 7.12 Breadth-first reachability using least fixpoint

$\mathcal{S} \leftarrow \mathcal{S}_0$;
$\mathcal{S}' \leftarrow \emptyset$;
while $\mathcal{S} \neq \mathcal{S}'$
  $\mathcal{S}' \leftarrow \mathcal{S}$;
  $\mathcal{N} \leftarrow \bigcup_t PostImage(\mathcal{S}, \mathcal{R}_t)$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{N}$;
end while

Of these two algorithms, which would you think tends to be more efficient? In practice, Algorithm 7.12 tends to be *faster* than Algorithm 7.11. Why is that?

1. There is one fewer operation: Algorithm 7.12 does not need to perform set difference.

2. When using decision diagrams, computing PostImage on a larger set is *not necessarily* more expensive. It depends on the size and shape of the decision diagram; as we have already seen, "larger set" does not necessarily mean "larger decision diagram".

3. As $\mathcal{S}$ converges, the decision diagram for $\mathcal{S}$ might not change much from iteration to iteration. So, even if the decision diagram for $\mathcal{S}$ is large, calls to *PostImage* could still be relatively fast, if enough recursive calls are already in the compute table.

There is yet another possible improvement to Algorithm 7.12. Notice that, as "potentially new" states are discovered, they are collected into set $\mathcal{N}$ before adding them to $\mathcal{S}$. But why wait? If we add them immediately to $\mathcal{S}$, we obtain the following algorithm.

Algorithm 7.13 Reachability using chaining

$\mathcal{S} \leftarrow \mathcal{S}_0$;
$\mathcal{S}' \leftarrow \emptyset$;
while $\mathcal{S} \neq \mathcal{S}'$
  $\mathcal{S}' \leftarrow \mathcal{S}$;
  foreach $t \in \mathcal{T}$
    $\mathcal{S} \leftarrow \mathcal{S} \cup PostImage(\mathcal{S}, \mathcal{R}_t)$;
  end for
end while

Note that while this algorithm will correctly generate the set $\mathcal{S}$, it does *not* strictly build the sequence

$$\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, ...$$

which does not matter *if we only care about building* $\mathcal{S}$. Also, note that Algorithm 7.13 could require fewer iterations of the outer while loop (and never more). If firing t_1 tends to cause t_2 to become enabled, then transition t_1 should be considered before t_2 in the foreach loop, because states added due to the firing of t_1 are likely to themselves produce new states when t_2 is fired. This effect is called *chaining*, which can be seen as an *acceleration* technique that can be applied when the transition relation is encoded in a (disjunctively) partition way.

Saturation Algorithm

The basic idea of the *Saturation* algorithm to generate $\mathcal{S}$ is to perform the iteration of Algorithm 7.13, but instead of traversing the entire decision diagram at each step of the iteration, we compute least fixpoints on nodes as each node is built. But how can we compute a fixpoint on a *node*, rather than on the set of state encoded by the entire decision diagram? We use the following observation:

> If a transition t does not affect state variables $x_L, ..., x_{k+1}$, then we can always (conceptually) fire t on any level k node.

In other words, the enabling and firing of transition t at any level k node will be independent of the path taken to reach that node, because the transition does not depend on (nor does it affect) any variable above level k.

Note also that, if an *identity reduction* is used for the transition relations, then the statement "transition t does not affect state variables $x_L, ..., x_{k+1}$" is equivalent to the statement "the decision diagram for $\mathcal{R}_t$ has root node at or below level k".

More precisely (details omitted for space), saturation works as follows.

1. Group transitions together based on "top level of $\mathcal{R}_t$".
2. Saturate a node p at level k (after having saturated all its children) by repeatedly firing in it all transitions with top level k, along the lines of Algorithm 7.13; however, if any such firing creates a node q at a lower level, recursively saturate it before continuing the saturation of p (this means that the *PostImage* algorithm must be modified accordingly to saturate every node as it is built). "Reduce" node p after it is saturated.
3. The set union instead works as usual.

We then build $\mathcal{S}$ by saturating each node in the decision diagram for the initial state(s).

The saturation algorithm can be applied when the transition relation is encoded in a (disjunctively partition way) using IRBDDs, and tends to be much more efficient than either breadth-first or chaining state-space generation.

7.6.4 Check CTL properties

STEP 4: Some of the labeling algorithms discussed earlier can be re-written to use decision diagrams. As before, we evaluate formulas from inner-most, outward. However, at each step, we build the set of states satisfying a (sub)formula.

- **Atomic propositions.** These are usually based on the high-level model, and the decision diagrams encoding the sets of states satisfying these propositions can be built by patterns for simple formulas (e.g., $\phi \equiv (x_5 = 2)$) or by combining simple patterns with operations for more complex formulas (e.g., $\phi \equiv ((x_5 = 2) \wedge (x_4 > 0) \wedge (x_7 = x_2))$).

This gives us the set of *all* states satisfying the atomic proposition ϕ; to obtain the *reachable* states satisfying the proposition, we must intersect $[\![\phi]\!]$ with $\mathcal{S}$ (this can be expensive and is not always needed).

- **and, or, negation.** And, or, and negation are done using the AND, OR, and NOT operations on decision diagrams.

- EX. To determine the set $[\![\mathsf{EX}\,\phi]\!]$, first build the set $[\![\phi]\!]$, then use *PreImage*. To restrict this to *reachable* states satisfying $\mathsf{EX}\,\phi$, intersect the result with $\mathcal{S}$ (again, this is not always needed).

- EU. A straightforward algorithm for building $[\![\mathsf{E}\,p\,\mathsf{U}\,q]\!]$ is to start with the set $[\![q]\!]$, then iterate: compute the next set using *PreImage* and intersecting the result with $[\![p]\!]$. It is possible to use a *constrained* version of Saturation here: using $[\![q]\!]$ as the set of initial states and applying the transition relation in reverse, we constrain Saturation to only consider edges that lead to states within $[\![p]\!]$.

- EG. This is usually done using the iterative method for EG: start with the set $[\![p]\!]$; at each iteration, take the *PreImage* and intersect with $[\![p]\!]$. When this converges, we have the set $[\![\mathsf{EX}\,p]\!]$.

Since EX, EU, and EG are an adequate set of the temporal operators, we can check any CTL formula using decision diagrams.

7.7 Challenges with using decision diagrams

We will now look at two fundamental questions about ROMDDs.

1. FRBDDs (for example) are a canonical representation, *given a variable ordering.* This means that two different variable orders may give two different FRBDD representations. Does the choice of variable ordering affect the number of nodes in the FRBDD, and if so, how significantly?

2. We can use FRBDDs to solve the *Satisfiability* problem: build an FRBDD representing the formula to check; the formula is satisfiable if and only if the FRBDD representation is not terminal node **0**. What is the worst–case complexity of FRBDDs? If they are efficient, then we could have $P = NP$, since the *Satisfiability* problem is known to be NP–complete.

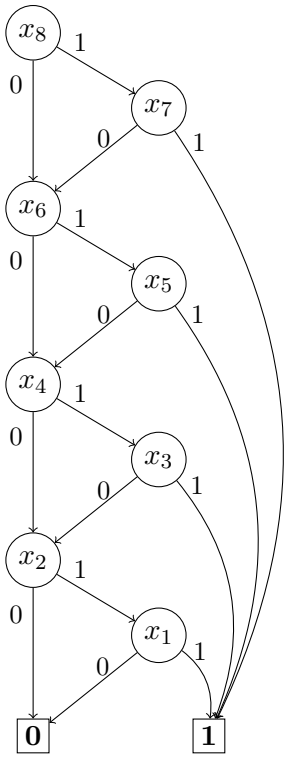
7.7.1 Variable ordering

We will answer the first question with examples taken from Bryant's 1986 article.

Example 7.10 Build the FRBDD for the (boolean) function

$$f(x_{2N}, ..., x_1) \;=\; x_{2N} \wedge x_{2N-1} \;\vee\; \cdots \;\vee\; x_4 \wedge x_3 \;\vee\; x_2 \wedge x_1$$

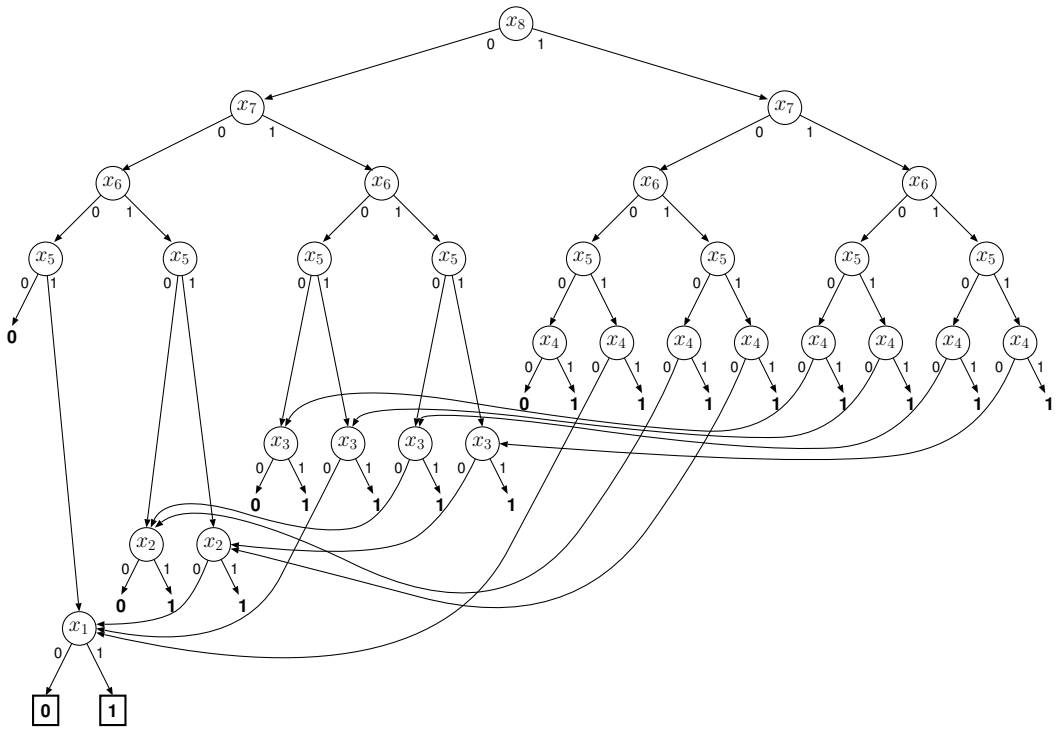
for the variable ordering $x_{2N}, ..., x_2, x_1$. You should obtain a BDD with $2N + 2$ nodes (including the terminal nodes). The case $N = 4$ is illustrated below.



Example 7.11 Build the ROBDD for the (boolean) function

$$f(x_{2N}, ..., x_1) \;=\; x_{2N} \wedge x_N \;\vee\; \cdots \;\vee\; x_{N+2} \wedge x_2 \;\vee\; x_{N+1} \wedge x_1$$

for the variable ordering $x_{2N}, ..., x_2, x_1$. This is the same function as the previous example, but with different variable ordering. The case $N = 4$ is illustrated below. We use several terminal nodes to clarify the illustration:



Assuming we merge all duplicate terminal nodes, the BDD will contain 2^{N+1} nodes.

As the above two examples illustrate, the size of a BDD can be *extremely* sensitive to the variable ordering. With one variable ordering, the number of BDD nodes grows as $\mathcal{O}(N)$, while with a different variable ordering, the number grows as $\mathcal{O}(2^N)$.

The second example also illustrates that ROBDDs can have exponential worst–case behavior if the variable ordering is not chosen wisely. Worse yet, we will see that there are functions for which *any* variable order results in a BDD of exponential size.

7.7.2 How to choose a good variable order

Unfortunately, the problem of finding an *optimal* variable ordering for a given function has been shown to be NP-complete.

In practice, a "good" variable order is found using one of two methods (or a combination of both).

1. **Static ordering:** Decide a "good" order before starting the computation, and keep that order throughout. There has been quite a bit of research on heuristics for finding good variable orders, for various problem domains. As a general rule of thumb, two variables that are "tightly coupled" (i.e., have a strong dependency) should remain close together in the chosen variable order. In Example 7.10, the tightly coupled variables are always adjacent in the chosen variable order ("and" is a stronger dependency than "or"). Instead, the order used in Example 7.11 has the coupled variables far apart; this requires the BDD to "remember" variable values along a path, which can only be done with distinct nodes.

2. **Dynamic ordering:** During the computation, periodically try to reorder variables to decrease the size of the BDD. Clearly this has a computational cost, so heuristics must be used

to decide both *when* to reorder and *how* to reorder. Typically, reordering is done by exchanging the order of two adjacent variables, as this operation can be done fairly locally (only rebuilding nodes at those two levels) in the BDD. Any desired order can then be achieved through a sequence of such pairwise exchanges.

7.7.3 Worst-case behavior

As Example 7.11 illustrates, there are FRBDDs with an exponential (in the number of variables) number of nodes for *some* variable order. But are there functions whose FRBDDs have an exponential number of nodes, *regardless* of the variable ordering? It has been shown[3] that the "hidden weighted bit function"

$$f(x_N, ..., x_1) \;=\; x_{x_N + \cdots + x_1}$$

requires a FRBDD representation with at least $\mathcal{O}(c^N)$ nodes, for $c \approx 1.14$, for *any* variable order. As such, FRBDDs cannot *always* solve the satisfiability problem in polynomial time, and require exponential time in the worst case.

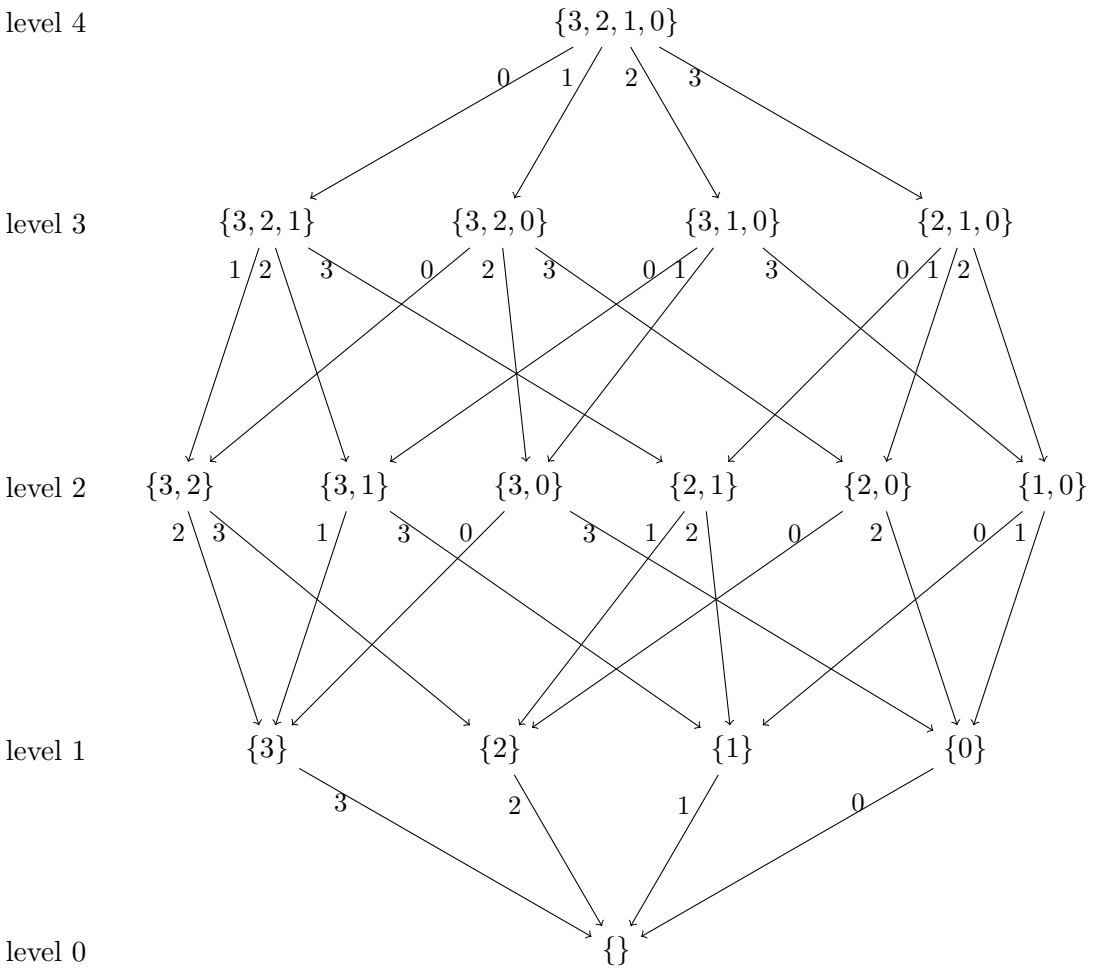
As an easier example, consider the set of permutations of the integers $\{0, 1, ..., M-1\}$. We can encode this set using the indicator function

$$f(x_M, ..., x_1) \;=\; 1 \quad \text{iff } \forall i \neq j,\; x_i \neq x_j$$

What does the MDD for f look like? First, note that all paths to a particular node $p \neq 0$ at level k must have the same set of values taken for variables $x_M, ..., x_{k+1}$, but in different orders. For example, assignments $x_M = 7, x_{M-1} = 3$ will lead to the same node as $x_M = 3, x_{M-1} = 7$ because we have the same set of values (namely, $\{0, ..., M-1\} \setminus \{3, 7\}$) to choose from for variables $x_{M-2}, ..., x_1$. Therefore, each node in the MDD corresponds to some subset of $\{0, ..., M-1\}$, from which the remaining variables can take their values. Furthermore, *every* possible subset $\mathcal{X}$ will correspond to some node, with $|\mathcal{X}|$ giving the level number of the node, because that's how many variables are left to assign. There will be one extra terminal node: $\emptyset$ corresponds to terminal **1**, while terminal node **0** is for illegal paths containing repeated values. The total number of nodes is therefore $2^M + 1$, and it can be shown that the total number of edges in the MDD is $M \cdot 2^{M-1}$. Also, note that *any* choice of variable order will produce exactly the same MDD.

Example 7.12 The MDD encoding the set of permutations of $\{0, 1, 2, 3\}$ is shown below. Instead of drawing a node, the figure below shows the set of values the remaining variables can be assigned from. Paths not shown go to terminal node **0**.

[3] R. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", in IEEE Transactions on Computers 40 (2), 1991.


level 4
level 3
level 2
level 1
level 0
{3, 2, 1, 0}
{3, 2, 1}
{3, 2, 0}
{3, 1, 0}
{2, 1, 0}
{3, 2}
{3, 1}
{3, 0}
{2, 1}
{2, 0}
{1, 0}
{3}
{2}
{1}
{0}
{}

Chapter 8

Linear Temporal Logic

Linear temporal logic, or LTL, is another commonly–used temporal logic. The name "linear", as opposed to "branching", refers to the fact that formulas describe conditions along a single, linear path, rather than allowing the possibility of the path to "branch" as it can in CTL. The temporal operators are the same, but there are subtle differences in the semantics. Outside reference: *Model Checking*, by Clarke, Grumberg, and Peled.

8.1 LTL syntax

LTL deals entirely with *path formulas*: formulas which can be, conceptually, verified along a single, infinitely–long path. An LTL path formula ψ has the following syntax:

$$\psi \ ::= \ \mathtt{tt} \mid \mathtt{ff} \mid a \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\,\psi \mid \mathsf{F}\,\psi \mid \mathsf{G}\,\psi \mid \psi\,\mathsf{U}\,\psi$$

where a is an atomic proposition. Given an LTL path formula ψ and a model to check, we say the model satisfies the formula if, for all paths starting in an initial state, the path formula ψ holds. For consistency, we will write an overall LTL formula as $\mathsf{A}\,\psi$, where ψ is a path formula. Thus, for example, $\mathsf{A}\,\mathsf{FGX}\,p$ is a valid LTL formula, but $\mathsf{AF}\,\mathsf{AG}\,p$ is not. However, it is common to omit the implied A path quantifier. Also, note that some people use $\bigcirc$ instead of X, $\Box$ instead of G, and $\Diamond$ instead of F. Thus, while we write $\mathsf{A}\,\mathsf{FGX}\,p$, others could write $\Diamond\Box\bigcirc p$ to denote the same formula.

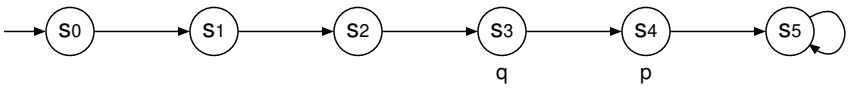
8.2 LTL semantics

We must give rules for which *paths* in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ satisfy an LTL path formula ψ, which we write as $M, \pi \models \psi$. When a path π does not satisfy path formula ψ, we instead write $M, \pi \not\models \psi$. For the following, assume $\pi = (p_0, p_1, ...)$, and let π^i denote the suffix of π starting with p_i, i.e., $\pi^i = (p_i, p_{i+1}, ...)$, for $i \geq 0$.

1. $M, \pi \models \mathtt{tt}$, for all paths.
2. $M, \pi \not\models \mathtt{ff}$, for all paths.
3. $M, \pi \models a \in \mathcal{A}$, if and only if $a \in L(p_0)$.
4. $M, \pi \models \neg\psi$, if and only if $M, \pi \not\models \psi$.
5. $M, \pi \models \psi_1 \wedge \psi_2$, if and only if $M, \pi \models \psi_1$ and $M, \pi \models \psi_2$.

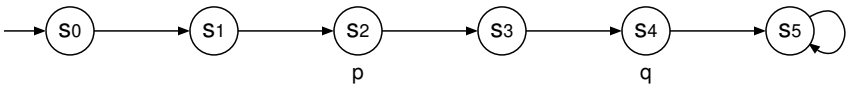
6. $M, \pi \models \psi_1 \vee \psi_2$, if and only if $M, \pi \models \psi_1$ or $M, \pi \models \psi_2$.
7. $M, \pi \models \mathsf{X}\psi$, if and only if $M, \pi^1 \models \psi$.
8. $M, \pi \models \mathsf{F}\psi$, if and only if there exists an $i \geq 0$ such that $M, \pi^i \models \psi$.
9. $M, \pi \models \mathsf{G}\psi$, if and only if, for all $i \geq 0$, $M, \pi^i \models \psi$.
10. $M, \pi \models \psi_1 \mathsf{U}\, \psi_2$, if and only if there exists a $j \geq 0$ such that
    (a) $M, \pi^j \models \psi_2$, and
    (b) $M, \pi^i \models \psi_1$ for all $i < j$.

Example 8.1 Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?



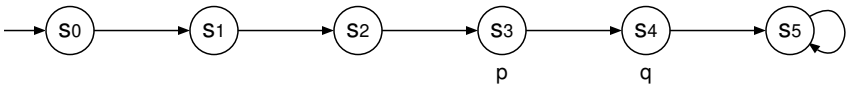
Solution: There is only one path, $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, ...)$. Check the path against the definition for U. Using $j = 3$, we have $\pi^j \models q$, and $\pi^i \models \mathsf{F}\,p$ for all $i < j$. So the answer is **yes**.

Example 8.2 Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?



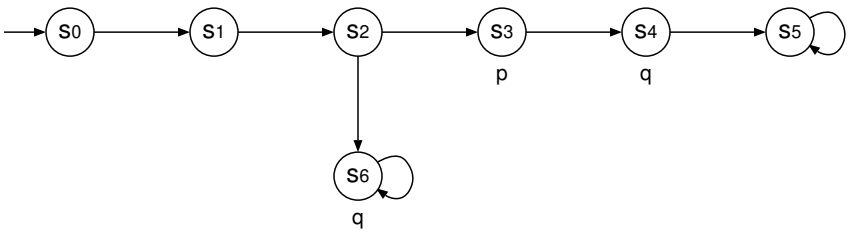
Solution: Again, there is only one possible path, $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, ...)$, to check against the definition for U. The only j where $\pi^j \models q$ is $j = 4$. But $\pi^3 = (s_3, s_4, s_5, s_5, ...)$ does not satisfy $\mathsf{F}\,p$. So the answer is **no**.

Example 8.3 Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?



Solution: Check the (only possible) path $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, ...)$ against the definition for U: using $j = 4$, we have $\pi^j \models q$. Then, check that $\pi^i \models \mathsf{F}\,p$ for all $i < j$. Since all of those hold, the answer is **yes**.

Example 8.4 Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?

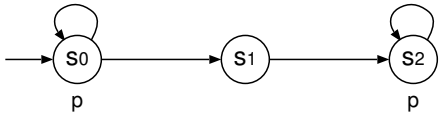


Solution: There are two paths to consider. For the path $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, ...)$, the analysis is the same as Example 8.3, and the path satisfies $(\mathsf{F}\, p)\, \mathsf{U}\, q$.

Now consider the other path, $\pi = (s_0, s_1, s_2, s_6, s_6, ...)$. For $j \geq 3$, we have $\pi^j = (s_6, s_6, ...)$ and $\pi^j \models q$. Now, does $\pi^i \models \mathsf{F}\, p$ for all $i < j$? For $i = 0$, the answer is **no**: $\pi^0 = (s_0, s_1, s_2, s_6, s_6, ...)$, and none of those states satisfy p, so $\pi^0 \not\models \mathsf{F}\, p$.

Thus, the answer is **no**.

Example 8.5 Does this model satisfy $\mathsf{A}\,\mathsf{FG}\, p$?



Solution: The paths in this model can be described by the number of times the edge from s_0 to s_0 is traversed before going to s_1. Let

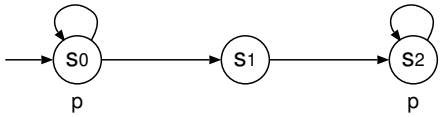
$$\begin{aligned} \pi_0 &= (s_0, s_1, s_2, s_2, ...) \\ \pi_1 &= (s_0, s_0, s_1, s_2, s_2, ...) \\ \pi_2 &= (s_0, s_0, s_0, s_1, s_2, s_2, ...) \\ &\vdots \end{aligned}$$

and finally, there is the possibility of never leaving s_0:

$$\pi_\infty = (s_0, s_0, s_0, ...)$$

For finite n, we have $\pi_n^{n+2} = (s_2, s_2, ...)$, thus $\pi_n^{n+2} \models \mathsf{G}\, p$, and $\pi_n \models \mathsf{FG}\, p$. But the path π_∞ also satisfies $\mathsf{G}\, p$, thus $\pi_\infty \models \mathsf{FG}\, p$. Thus, all paths satisfy $\mathsf{FG}\, p$, and the answer is **yes**.

Example 8.6 Does this model satisfy the CTL formula $\mathsf{AF}\,\mathsf{AG}\, p$?



Solution: Working inward, we first determine which states satisfy $\mathsf{AG}\, p$: only s_2. Next, we determine which states satisfy $\mathsf{AF}\,\mathsf{AG}\, p$, knowing that only $s_2 \models \mathsf{AG}\, p$. The answer is s_1 and s_2. Since the initial state s_0 is not in this set, the answer is **no**.

Property 8.1

$$\mathsf{AF}\,\mathsf{AG}\, p \not\equiv \mathsf{A}\,\mathsf{FG}\, p$$

Proof: from Example 8.5 and Example 8.6, there exists a Kripke structure that satisfies one formula but not the other.

The previous two examples illustrate the difference between CTL and LTL: in CTL, by nesting $\mathsf{AF}\,\mathsf{AG}\, p$, we must allow "branching" because we quantify over paths twice: for all paths, in the future, *for all paths*, globally. In contrast, the LTL nesting $\mathsf{A}\,\mathsf{FG}\, p$ (and any other nesting) only quantifies over paths once: for all paths, in the future, globally.

8.3 Equivalences

Note: these are *path formula* equivalences. To prove such an equivalence, we must show that for any path π, π satisfies the first path formula if and only if π satisfies the second path formula. To disprove an equivalence, it suffices to find a path that satisfies one formula but not the other.

8.3.1 Negations

Property 8.2

$$\neg \mathsf{A}\psi \equiv \mathsf{E}\neg\psi$$

Proof: Follows from a similar property for $\forall$ and $\exists$.

As a consequence of Property 8.2 and the fact that $\neg\psi$ is a valid path formula for any path formula ψ, it is possible to use LTL model checking to test for existence of a path satisfying a formula. Thus, from now on, we will consider $\mathsf{E}\psi$ to be a valid LTL formula, where ψ is a path formula.

Property 8.3

$$\neg \mathsf{X}\psi \equiv \mathsf{X}\neg\psi$$

Proof:

$$\begin{aligned} \pi \models \neg \mathsf{X}\psi &\Leftrightarrow \pi \not\models \mathsf{X}\psi \\ &\Leftrightarrow \pi^1 \not\models \psi \\ &\Leftrightarrow \pi^1 \models \neg\psi \\ &\Leftrightarrow \pi \models \mathsf{X}\neg\psi \end{aligned}$$

Property 8.4

$$\mathsf{F}\psi \equiv \mathtt{tt}\,\mathsf{U}\,\psi$$

Property 8.5

$$\neg \mathsf{G}\psi \equiv \mathsf{F}\neg\psi$$

We thus have that $\{\neg, \wedge, \mathsf{X}, \mathsf{U}\}$ is an adequate set of operators for expressing LTL path formulas. Compare this with the possible adequate sets of operators for CTL.

8.3.2 Conjunctions and disjunctions

Property 8.6

$$\begin{aligned} \mathsf{X}(\psi_1 \wedge \psi_2) &\equiv (\mathsf{X}\,\psi_1) \wedge (\mathsf{X}\,\psi_2) \\ \mathsf{X}(\psi_1 \vee \psi_2) &\equiv (\mathsf{X}\,\psi_1) \vee (\mathsf{X}\,\psi_2) \end{aligned}$$

Property 8.7

$$\mathsf{F}(\psi_1 \vee \psi_2) \equiv (\mathsf{F}\psi_1) \vee (\mathsf{F}\psi_2)$$

Property 8.8

$$\mathsf{G}(\psi_1 \wedge \psi_2) \equiv (\mathsf{G}\psi_1) \wedge (\mathsf{G}\psi_2)$$

Property 8.9

$$\begin{aligned} (\psi_1 \wedge \psi_2)\,\mathsf{U}\,\psi &\equiv (\psi_1\,\mathsf{U}\,\psi) \wedge (\psi_2\,\mathsf{U}\,\psi) \\ \psi\,\mathsf{U}\,(\psi_1 \vee \psi_2) &\equiv (\psi\,\mathsf{U}\,\psi_1) \vee (\psi\,\mathsf{U}\,\psi_2) \end{aligned}$$

8.3.3 Redundant nesting

Property 8.10

$$\mathsf{FF}\psi \equiv \mathsf{F}\psi$$

Property 8.11

$$\mathsf{GG}\psi \equiv \mathsf{G}\psi$$

Property 8.12

$$\psi_1 \,\mathsf{U}\, (\psi_1 \,\mathsf{U}\, \psi_2) \equiv \psi_1 \,\mathsf{U}\, \psi_2$$

8.3.4 Recursion

Property 8.13

$$\mathsf{F}\,\psi \equiv \psi \vee \mathsf{XF}\,\psi$$

Property 8.14

$$\mathsf{G}\,\psi \equiv \psi \wedge \mathsf{XG}\,\psi$$

Property 8.15

$$\psi_1 \,\mathsf{U}\, \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \mathsf{X}\,(\psi_1 \,\mathsf{U}\, \psi_2))$$

8.4 LTL model checking by tableau

The basic idea of the tableau method is as follows.

1. Start with an LTL path formula ψ.
2. For each state in the Kripke structure, consider all possible combinations of "which subformulas of ψ hold". Label each vertex with a Kripke structure state, and which subformulas hold. These can be viewed either as "properties that hold" or as "obligations" along paths.
3. Eliminate impossible combinations. For example, if p holds and q holds, then $p \wedge q$ must also hold.
4. Draw edges between vertices according to the Kripke structure.
5. Eliminate logically impossible edges, based on subformula labels.

The resulting graph is called the tableau graph, and it can be used to determine if $s \models \mathsf{E}\,\psi$ in the original Kripke structure. The rest of this section fills out the details of this method.

8.4.1 Subformulas and closure

Rewrite the original formula ψ to contain only the operators $\neg$, $\wedge$, X, and U. Then, build a set of relevant subformulas of ψ, called the *closure* of ψ and denoted $C(\psi)$, recursively as follows.

- $C(\mathtt{tt}) = C(\mathtt{ff}) = \emptyset$.
- If $\psi = p$, an atomic proposition, then $C(\psi) = \{p\}$.
- If $\psi = \neg\phi$, then $C(\psi) = C(\phi)$.

- If $\psi = \phi_1 \wedge \phi_2$, then $C(\psi) = \{\psi\} \cup C(\phi_1) \cup C(\phi_2)$.
- If $\psi = \mathsf{X}\,\phi$, then $C(\psi) = \{\psi\} \cup C(\phi)$.
- If $\psi = \phi_1 \,\mathsf{U}\, \phi_2$, then $C(\psi) = \{\psi\} \cup C(\phi_1) \cup C(\phi_2)$.

Definition 8.16 *The* length *of a path formula ψ, denoted as $|\psi|$, is given by*

$$|\psi| \quad \equiv \quad |C(\psi)|.$$

Note that $|\psi|$ is at most the total number of operators and atomic propositions in the formula ψ.

8.4.2 Building the tableau graph

We build a tableau graph $T = (V, E)$ where a vertex in the graph is a pair $v = (s_v, L_v)$, where s_v is a state of the Kripke structure and L_v is a labeling for each of the subformulas in $C(\psi)$, either that are known to hold, or are *obligated* to hold. We denote this as a set of subformulas that hold, so formally, we have $V \subseteq \mathcal{S} \times 2^{C(\psi)}$. We say a vertex $v = (s_v, L_v)$ satisfies ϕ, written $v \models \phi$, if

- $\phi \in L_v$, for $\phi \in C(\psi)$; or
- $\neg\phi \notin L_v$, for $\neg\phi \in C(\psi)$,

we do this to reduce the size of $C(\psi)$, to include "positives only". Note that we implicitly have $v \models \texttt{tt}$ and $v \not\models \texttt{ff}$. We do not allow inconsistent labelings, according to the following rules. For any vertex $v = (s_v, L_v)$:

L0. If p is an atomic proposition, then $p \in L_v$ if and only if $p \in L(s_v)$.

L1. For $\phi = \phi_1 \wedge \phi_2$, $v \models \phi$ if and only if $v \models \phi_1$ and $v \models \phi_2$.

L2. If $v \models \phi_2$, then $v \models \phi_1 \,\mathsf{U}\, \phi_2$, provided $\phi_1 \,\mathsf{U}\, \phi_2 \in C(\psi)$.

L3. If $v \models \phi_1 \,\mathsf{U}\, \phi_2$, then $v \models \phi_1$ or $v \models \phi_2$.

We add edges to the tableau graph according to the following rules. We add an edge from vertex $v = (s_v, L_v)$ to vertex $v' = (s'_v, L'_v)$ if and only if:

E0. $(s_v, s'_v) \in \mathcal{R}$, i.e., the edge matches one in the Kripke structure, and

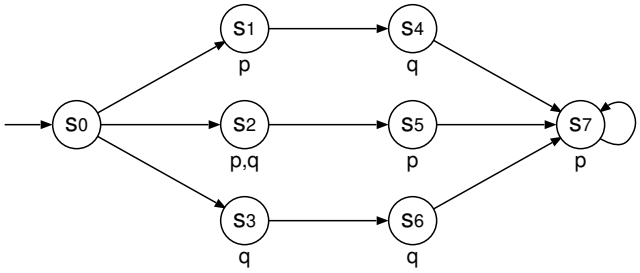
E1. for every formula of the form $\mathsf{X}\,\phi \in C(\psi)$,

$$\mathsf{X}\,\phi \in L_v \qquad \text{if and only if} \qquad v' \models \phi, \qquad \text{and}$$

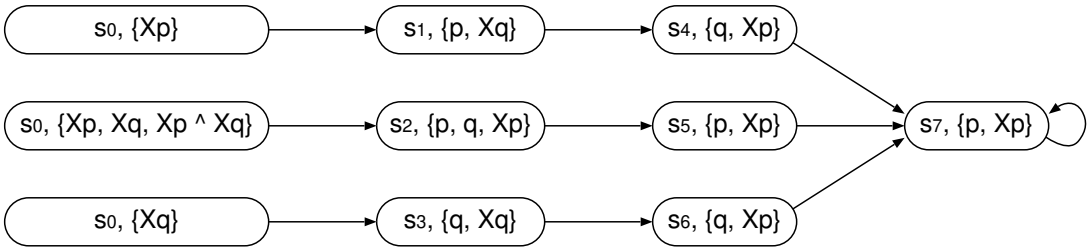
E2. for every formula of the form $\phi_1 \,\mathsf{U}\, \phi_2 \in C(\psi)$, if $v \models \phi_1$ and $v \not\models \phi_2$, then

$$\phi_1 \,\mathsf{U}\, \phi_2 \in L_v \qquad \text{if and only if} \qquad \phi_1 \,\mathsf{U}\, \phi_2 \in L'_v.$$

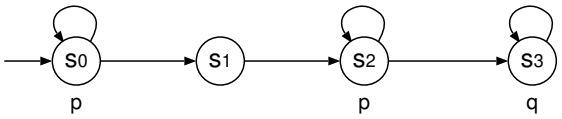
Example 8.7 For the Kripke structure below, draw the tableau graph for formula $\psi = \mathsf{X}\,p \wedge \mathsf{X}\,q$.



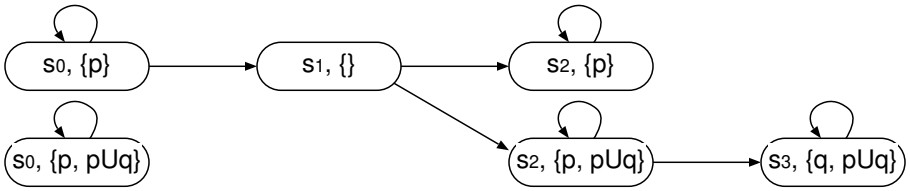
Solution: First, note that we have $C(\psi) = \{p, q, \mathsf{X}\, p, \mathsf{X}\, q, \mathsf{X}\, p \wedge \mathsf{X}\, q\}$. We obtain the following tableau graph, where vertices with no outgoing edges are omitted from the drawing to save space.



Example 8.8 For the Kripke structure below, draw the tableau graph for formula $\psi = p\,\mathsf{U}\,q$.



Solution: We have $C(\psi) = \{p, q, p\,\mathsf{U}\,q\}$. For states satisfying p, we can choose to label vertices with $p\,\mathsf{U}\,q$ or not. For states satisfying q, we must label with $p\,\mathsf{U}\,q$. For states satisfying neither p nor q, we cannot label with $p\,\mathsf{U}\,q$. Observint the edge rules, we obtain the following tableau graph.



Property 8.17 For a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$, the tableau graph for formula ψ has at most

$$2^{|\psi|} \cdot |\mathcal{S}| \quad \text{states, and} \quad 2^{|\psi|} \cdot |\mathcal{R}| \quad \text{edges.}$$

8.4.3 Model checking with the tableau graph: theory

Suppose we have a tableau graph $T = (V, E)$ for Kripke structure M and formula ψ, and want to determine if $M, s \models \mathsf{E}\,\psi$ for some state s. What we would like to do is say something like

$$M, s \models \mathsf{E}\,\psi \quad \text{if and only if} \quad \exists v = (s, L_v) \in V, v \models \psi.$$

There is a small problem with this, though. It is best illustrated by example.

Example 8.9 From the Kripke structure in Example 8.7, we have $s_0 \models \mathsf{E}(\mathsf{X}p \wedge \mathsf{X}q)$, and in the tableau graph, there is a vertex with state s_0 labeled with $\mathsf{X}p \wedge \mathsf{X}q$. No other states satisfy $\mathsf{E}(\mathsf{X}p \wedge \mathsf{X}q)$, and no other vertices are labeled with $\mathsf{X}p \wedge \mathsf{X}q$. This works fine.

Example 8.10 From the Kripke structure in Example 8.8, we have $s_0 \not\models \mathsf{E}\,p\,\mathsf{U}\,q$, but in the tableau graph, there *is* a vertex with state s_0 labeled with $p\,\mathsf{U}\,q$.

Looking carefully at Example 8.10, we can see the problem: it is possible to have a vertex labeled with a formula $\phi_1\,\mathsf{U}\,\phi_2$, and yet have no path to a vertex satisfying ϕ_2.

Definition 8.18 An *eventuality sequence* is an infinite path $\pi = (v_0, v_1, ...)$ in a tableau graph T such that, for any vertex $v_i = (s_i, L_i)$ in π, if there is a formula of the form $\phi_1\,\mathsf{U}\,\phi_2$ in L_i, then there is a vertex v_j in π with $v_j \models \phi_2$, with $i \leq j$.

Note: by construction of the tableau graph (rule E2), on an eventuality sequence, once we reach a vertex labeled with a formula $\phi_1\,\mathsf{U}\,\phi_2$, all following vertices will be labeled with $\phi_1\,\mathsf{U}\,\phi_2$, until we reach a vertex satisfying ϕ_2. Because it is an eventuality sequence, it is guaranteed that we will eventually satisfy ϕ_2.

Example 8.11 Continuing Example 8.8, looking at the tableau graph, the infinite sequence

$$((s_0, \{p, p\,\mathsf{U}\,q\}), (s_0, \{p, p\,\mathsf{U}\,q\}, ...)$$

is not an eventuality sequence, because we never reach a vertex labeled with q. The infinite sequence

$$((s_0, \{p\}), (s_1, \emptyset), (s_2, \{p\}), (s_2, \{p\}), ...)$$

is an eventuality sequence, because there are no $\phi_1\,\mathsf{U}\,\phi_2$ formulas that never reach ϕ_2. Also, the infinite sequence

$$((s_0, \{p\}), (s_1, \emptyset), (s_2, \{p, p\,\mathsf{U}\,q\}), (s_2, \{p, p\,\mathsf{U}\,q\}), (s_3, \{q, p\,\mathsf{U}\,q\}), (s_3, \{q, p\,\mathsf{U}\,q\}, ...)$$

is an eventuality sequence.

Property 8.19 $M, s \models \mathsf{E}\,\psi$ if and only if the tableau graph for ψ contains an eventuality sequence starting from a vertex v with state s, and $v \models \psi$.

Proof $\rightarrow$: If $M, s \models \mathsf{E}\,\psi$, then there exists a path $\pi = (s_0 = s, s_1, s_2, ...)$ with $\pi \models \psi$. For all i, let $v_i = (s_i, L_i)$ where, for all $\phi \in C(\psi)$, $\phi \in L_i$ if and only if $\pi^i \models \phi$. Note that v_i is a valid vertex.

First, we must show that $(v_0, v_1, ...)$ is a path in the tableau graph. For each (v_i, v_{i+1}), rule E0 is satisfied because π is a path. Rule E1 is satisfied because $\pi^i \models \mathsf{X}\,\phi$ iff $\pi^{i+1} \models \phi$. Rule E2 is satisfied because if $\pi^i \models \phi_1$ and $\pi^i \not\models \phi_2$, then by Property 8.15 we have $\pi^i \models \phi_1\,\mathsf{U}\,\phi_2$ if and only if $\pi^{i+1} \models \phi_1\,\mathsf{U}\,\phi_2$. Finally, we must show that $(v_0, v_1, ...)$ is an eventuality sequence:

$$\begin{array}{lll} \phi_1\,\mathsf{U}\,\phi_2 \in L_i & \rightarrow & \pi^i \models \phi_1\,\mathsf{U}\,\phi_2 \qquad\qquad\qquad\qquad \text{(by construction of } L_i\text{)} \\ & \rightarrow & \exists j \geq i, \pi^j \models \phi_2, \pi^{i'} \models \phi_1, \forall i \leq i' < j \\ & \rightarrow & \exists j \geq i, \pi^j \in L_j, \pi^{i'} \in L_{i'}, \forall i \leq i' < j \quad \text{(by construction of } L_i, ..., L_j\text{)} \end{array}$$

Proof $\leftarrow$: Suppose there is an eventuality sequence $(v_0 = v, v_1, v_2, ...)$, where $v_i = (s_i, L_i)$. Let $\pi = (s_0, s_1, ...)$. We will prove the stronger claim that $\pi^i \models \psi$ if and only if $v_i \models \psi$, by induction on the structure of ψ.

In the base case, $\psi = p$, an atomic proposition, and by labeling rule L0, $v_i \models p$ iff $s_i \models p$.

Now, assume it holds for all subformulas of ψ, and prove it holds for ψ.

If $\psi = \neg\phi$, then $\pi^i \models \psi$ iff $\pi^i \not\models \phi$. By the inductive hypothesis, $\pi^i \models \phi$ iff $v_i \models \phi$, and it follows that $\pi^i \models \neg\phi$ iff $v_i \models \neg\phi$.

If $\psi = \phi_1 \wedge \phi_2$, then by the inductive hypothesis, $\pi^i \models \phi_1$ iff $v_i \models \phi_1$, and $\pi^i \models \phi_2$ iff $v_i \models \phi_2$. From labeling rule L1, $v_i \models \phi_1 \wedge \phi_2$ iff $v_i \models \phi_1$ and $v_i \models \phi_2$. It follows that $\pi^i \models \phi_1 \wedge \phi_2$ iff $v_i \models \phi_1 \wedge \phi_2$.

If $\psi = \mathsf{X}\,\phi$, then we have the following.

$$\begin{array}{rcll} \pi^i \models \mathsf{X}\,\phi & \Leftrightarrow & \pi^{i+1} \models \phi & \text{(definition of } \mathsf{X}) \\ & \Leftrightarrow & v_{i+1} \models \phi & \text{(by inductive hypothesis)} \\ & \Leftrightarrow & v_i \models \mathsf{X}\,\phi & \text{(by edge rule E1)} \end{array}$$

If $\psi = \phi_1 \,\mathsf{U}\, \phi_2$, then we have

$$\begin{array}{rcll} v_i \models \phi_1 \,\mathsf{U}\, \phi_2 & \rightarrow & v_i \models \phi_2 \vee v_i \not\models \phi_2, v_i \models \phi_1 & \text{(labeling rule L3)} \\ & \rightarrow & v_i \models \phi_2 \vee v_i \not\models \phi_2, v_i \models \phi_1, v_{i+1} \models \phi_1 \,\mathsf{U}\, \phi_2 & \text{(edge rule E2)} \\ & \rightarrow & \pi^i \models \phi_2 \vee \pi^i \models \phi_1, v_{i+1} \models \phi_1 \,\mathsf{U}\, \phi_2 & \text{(inductive hypothesis)} \\ & \rightarrow & \pi^i \models \phi_2 \vee (\pi^i \models \phi_1, \pi^{i+1} \models \phi_2 \vee (\pi^{i+1} \models \phi_1 ...)) & \\ & \rightarrow & \exists j \geq i, \pi^j \models \phi_2, \pi^{i'} \models \phi_1, \forall i \leq i' < j & \text{(eventuality sequence)} \\ & \rightarrow & \pi^i \models \phi_1 \,\mathsf{U}\, \phi_2 & \\ \pi^i \models \phi_1 \,\mathsf{U}\, \phi_2 & \rightarrow & \exists j \geq i, \pi^j \models \phi_2, \pi^{i'} \models \phi_1, \forall i \leq i' < j & \\ & \rightarrow & \exists j \geq i, v_j \models \phi_2, v_{i'} \models \phi_1, \forall i \leq i' < j & \text{(inductive hypothesis)} \\ & \rightarrow & \exists j \geq i, v_j \models \phi_2, v_j \models \phi_1 \,\mathsf{U}\, \phi_2, v_{i'} \models \phi_1, \forall i \leq i' < j & \text{(labeling rule L2)} \\ & \rightarrow & \exists j \geq i, v_j \models \phi_2, v_{i'} \models \phi_1 \,\mathsf{U}\, \phi_2, \forall i \leq i' \leq j & \text{(edge rule E2)} \\ & \rightarrow & v_i \models \phi_1 \,\mathsf{U}\, \phi_2 & \end{array}$$

Example 8.12 Continuing Example 8.7, we have $s_0 \models \mathsf{E}\,(\mathsf{X}\,p \wedge \mathsf{X}\,q)$, because in the tableau graph, we have the eventuality sequence

$$((s_0, \{\mathsf{X}\,p, \mathsf{X}\,q, \mathsf{X}\,p \wedge \mathsf{X}\,q\}), (s_2, \{p, q, \mathsf{X}\,p\}), (s_5, \{p, \mathsf{X}\,p\}), (s_7, \{p, \mathsf{X}\,p\}), ...)$$

which starts with a vertex for state s_0, labeled with $\mathsf{X}\,p \wedge \mathsf{X}\,q$.

Example 8.13 Continuing Example 8.8, we have $s_0 \not\models \mathsf{E}\,p \,\mathsf{U}\, q$, because there is no eventuality sequence starting with a vertex for state s_0 labeled with $p \,\mathsf{U}\, q$.

How do we find infinitely–long paths that satisfy a property in a finite graph? We look at the strongly connected components (SCCs). This motivates the following definition and theorem.

Definition 8.20 A non-trivial[1] SCC in a tableau graph is *self–fulfilling* if for every vertex $v = (s_v, L_v)$ in the SCC, if there is a formula of the form $\phi_1 \,\mathsf{U}\, \phi_2$ in L_v, then there exists a vertex v' in the SCC with $v' \models \phi_2$.

Property 8.21 For any vertex v in a self–fulfilling SCC $\mathcal{C}$, there is an eventuality sequence starting with v.

Proof: We construct an eventuality sequence as follows. If v is not labeled with any formula of the

[1] A SCC is *non-trivial* if it contains more than one state, or is a single state with a loop.

form $\phi_1 \mathsf{U} \phi_2$, then choose some successor of v in $\mathcal{C}$, and repeat. Otherwise, for each such formula, there exists a vertex that satisfies ϕ_2. Add a path from v that visits each of these vertices. Repeat the process from the final vertex. Clearly, the resulting infinitely–long sequence is an eventuality sequence.

Property 8.22 There is an eventuality sequence starting from vertex v if and only if there is a path in the tableau graph from v to a self–fulfilling SCC.

Proof $\rightarrow$: Suppose there is an eventuality sequence starting from v. Let $\mathcal{I}$ be the set of vertices that appear infinitely often in the sequence. Because the graph is finite, we must have $|\mathcal{I}| > 0$, and there is some non-trivial SCC $\mathcal{C}$ such that $\mathcal{I} \subseteq \mathcal{C}$. We will show that $\mathcal{C}$ is self–fulfilling, by contradiction. Suppose there exists a vertex $v' \in \mathcal{C}$, labeled with $\phi_1 \mathsf{U} \phi_2$, and no vertex exists in $\mathcal{C}$ satisfying ϕ_2. By construction of the tableau graph, all successors of v' must also be labeled with $\phi_1 \mathsf{U} \phi_2$, thus all vertices in $\mathcal{C}$ are labeled with $\phi_1 \mathsf{U} \phi_2$. But since $\mathcal{I} \subseteq \mathcal{C}$, if we look at the "end" of the eventuality sequence (after the point where all remaining vertices are in $\mathcal{I}$), we have vertices labeled with $\phi_1 \mathsf{U} \phi_2$ and no vertex satisfying ϕ_2. But this is impossible, in an eventuality sequence.

Proof $\leftarrow$: Suppose there is a path in the tableau graph from v to a self–fulfilling SCC $\mathcal{C}$. Consider some path $(v_0 = v, v_1, v_2, ..., v_j, v')$ where $v' \in \mathcal{C}$. From Property 8.21, there is an eventuality sequence $(v', v'_1, v'_2, ...)$. We show that $(v_0, ..., v_j, v', v'_1, v'_2, ...)$ is also an eventuality sequence. For any vertex v_i, if it is labeled with $\phi_1 \mathsf{U} \phi_2$, then either ϕ_2 holds for some vertex before reaching v', or not. If not, then v' must also be labeled with $\phi_1 \mathsf{U} \phi_2$, and since $(v', v'_1, v'_2, ...)$ is an eventuality sequence, there is some v'_j with $v'_j \models \phi_2$.

Property 8.23 $M, s \models \mathsf{E} \psi$ if and only if, in the tableau graph for ψ, there is a path from a vertex with state s, labeled with ψ, to a self–fulfilling SCC.

Proof: Follows immediately from Property 8.19 and Property 8.22.

8.4.4 Model checking with the tableau graph: algorithm

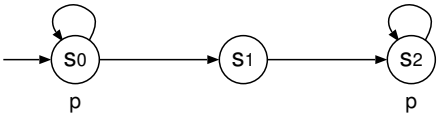
Based on the above discussion, we have the following algorithm for determining which states in a Kripke structure satisfy $\mathsf{E} \psi$ for an LTL path formula ψ.

1. Build the tableau graph $T = (V, E)$.
2. Determine the SCCs for T.
3. For each non-trivial SCC, determine if it is self–fulfulling or not:
   (a) Build a list of all formulas of the form $\phi_1 \mathsf{U} \phi_2$ that label vertices in the SCC.
   (b) For each formula in the list, check that some state in the SCC satisfies ϕ_2.
4. Label all self–fulfilling SCC vertices as *green*, and any vertex that can reach a green vertex as *green*.
5. $s \models \mathsf{E} \psi$ if and only if there is a green vertex (s, L) with $\psi \in L$.

What is the complexity of this algorithm? Step (1) is $\mathcal{O}(|T|) = \mathcal{O}(|V| + |E|)$, as is Step (2). Step (3) is $\mathcal{O}(|V|)$ in the worst case, Step (4) is $\mathcal{O}(|T|)$, and Step (5) gives us an answer for every state in $\mathcal{O}(|V|)$. Thus, from Property 8.17, the (worst case) complexity to check every state in Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ against LTL formula ψ is

$$\mathcal{O}(2^{|\psi|}(|\mathcal{S}| + |\mathcal{R}|)).$$

Example 8.14 For the Kripke structure below, which states satisfy $\mathsf{A}\,\mathsf{FG}\,p$?



Solution: First, rewrite the formula in terms of E and U:

$$\begin{aligned}
\mathsf{A}\,\mathsf{FG}\,p &\equiv \neg\mathsf{E}\neg\mathsf{FG}\,p \\
&\equiv \neg\mathsf{E}\neg\mathsf{F}\neg\mathsf{F}\neg p \\
&\equiv \neg\mathsf{E}\neg\mathsf{F}\neg(\mathtt{tt}\,\mathsf{U}\,\neg p) \\
&\equiv \neg\mathsf{E}\neg(\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p))
\end{aligned}$$

The subformulas of ψ are:

$$\begin{aligned}
C(\psi) &= C(\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)) \\
&= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)\} \cup C(\mathtt{tt}) \cup C(\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)) \\
&= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)\} \cup C(\mathtt{tt}\,\mathsf{U}\,\neg p) \\
&= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p), \mathtt{tt}\,\mathsf{U}\,\neg p\} \cup C(\mathtt{tt}) \cup C(\neg p) \\
&= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p), \mathtt{tt}\,\mathsf{U}\,\neg p, p\}.
\end{aligned}$$

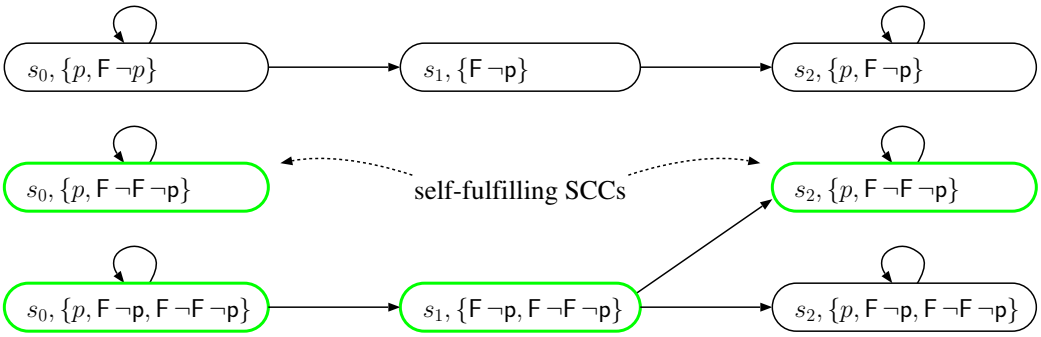
For shorthand, though, we will use $C(\psi) = \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$. Now we can build the tableau graph. First, we determine which vertices are in the graph according to the labeling rules, noting that $\neg p \to \mathsf{F}\,\neg p$ and $\neg\mathsf{F}\,\neg p \to \mathsf{F}\,\neg\mathsf{F}\,\neg p$. That leaves us with the vertices

$$\begin{aligned}
V = \{\ & (s_0, \{p, \mathsf{F}\,\neg p\}), (s_0, \{p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), (s_0, \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), \\
& (s_1, \{\mathsf{F}\,\neg p\}), (s_1, \{\mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), \\
& (s_2, \{p, \mathsf{F}\,\neg p\}), (s_2, \{p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), (s_2, \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\})\ \}.
\end{aligned}$$

Rewriting the U edge restriction from vertex v to v' for our specific formulas, we obtain:

1. If $v \models p$, then $\mathsf{F}\,\neg p$ labels both v and v', or neither.
2. If $v \models \mathsf{F}\,\neg p$, then $\mathsf{F}\,\neg\mathsf{F}\,\neg p$ labels both v and v', or neither.

These rules give us the following tableau graph.



In the graph, the indicated SCCs are self–fulfilling because the vertices satisfy $\neg\mathsf{F}\,\neg p$. The others are not, because the vertices are all labeled with p, thus $\mathsf{F}\,\neg p$ cannot be satisfied. From the green vertices, we see that all states satisfy $\mathsf{E}\,\mathsf{F}\,\neg\mathsf{F}\,\neg p \equiv \mathsf{E}\,\mathsf{FG}\,p$ (there exists a path where, eventually, we never see $\neg p$), and no state satisfies $\mathsf{E}\,\neg\mathsf{FG}\,p$. Thus, all states satisfy $\mathsf{A}\,\mathsf{FG}\,p$.

8.5 LTL model checking with Büchi Automata

Another algorithm for LTL model checking is based on Büchi Automata[2]. Before we can understand this algorithm, we must first cover some automata theory.

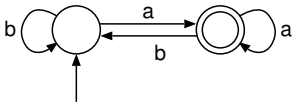
8.5.1 Büchi Automata

Basic idea

A Büchi Automaton is a finite state machine for accepting infinte words. The idea is similar to deterministic and non-deterministic finite automata, except those are languages of *finite* words. The idea is:

- Start in an initial state.
- Follow edges in finite automaton, according to "next input symbol"
- If we end up in an "accepting" state at the end of the input, then the word is accepted; otherwise it is not accepted.

Example 8.15 The DFA below accepts finite words over alphabet $\{a, b\}$ that end in "a". It is deterministic because in every state, there is at most one outgoing edge for each symbol.



Büchi automata are the same idea, except there's a catch: because words are infinitely long, we cannot "end up" in a state. We will need more interesting acceptance criteria. But first, some definitions and terminology.

Definition 8.24 A (non-deterministic) *ω-automaton* is:

- Σ, a finite alphabet (symbols in the input words)
- Q, a finite set of states
- Q_0, a non-empty set of initial states with $Q_0 \subseteq Q$.
- $\Delta \subseteq Q \times \Sigma \times Q$, the transition relation
- An acceptance condition, depending on the specific type of automaton (discussed below)

Definition 8.25 An *input* is an infinite word $\alpha = \sigma_0\sigma_1\sigma_2... \in \Sigma^\omega$.

Definition 8.26 For an input word $\alpha = \sigma_0\sigma_1\sigma_2...$, a *run* is an infinite sequence of states $\rho = q_0, q_1, q_2, ... \in Q^\omega$ such that $q_0 \in Q_0$ and

$$(q_i, \sigma_i, q_{i+1}) \in \Delta, \quad \forall i$$

Definition 8.27 For a given run $\rho = q_0, q_1, q_2, ... \in Q^\omega$, the set of states visited infinitely often is denoted

$$Inf(\rho) = \{q \mid q \in Q, \exists \text{ infinitely many } i \text{ such that } q_i = q\}$$

Note that the set $Inf(\rho)$ can never be empty because Q is finite and ρ is not.

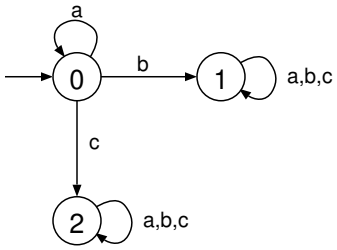
[2]See for example: M.Vardi and P. Wolper. "An Automata–Theoretic Approach to Automatic Program Verification". In *LICS'1986*, pp. 332–344, 1986.

Standard Büchi automata

Definition 8.28 A *(standard) Büchi automaton* is an ω-automaton with

- F, a finite set of states with $F \subseteq Q$. A run ρ is accepting if and only if $Inf(\rho) \cap F \neq \emptyset$.

Example 8.16 Consider the (standard) Büchi automaton given by:

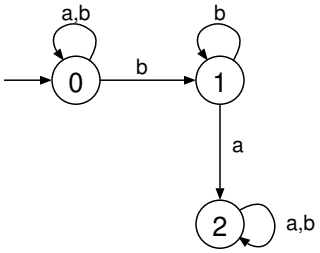


with $\Sigma = \{a, b, c\}$, $Q = \{0, 1, 2\}$, $Q_0 = \{0\}$, $F = \{1\}$. This is a deterministic automaton because no state has more than one outgoing edge for the same symbol. Consider the input word $\alpha = aaabccccc^{\omega}$. This produces the run $0, 0, 0, 0, 1, 1, 1, 1^{\omega}$. This run is accepting because state $1 \in F$ is visited infinitely many times. We can characterize the words that produce accepting runs as words where b eventually appears, and appears before any c.

Definition 8.29 The *language accepted by* automaton A, denoted $L(A)$, is defined by

$$L(A) = \{\alpha \mid \exists \text{ an accepting run in } A \text{ for } \alpha\}$$

Example 8.17 Consider the (standard) Büchi automaton given by:



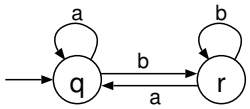
with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2\}$, $Q_0 = \{0\}$, $F = \{1\}$. This is non-deterministic because in state 0, if symbol b appears, we can either stay in state 0 or go to state 1. Consider the input word $\alpha = babbbb^{\omega}$. There are many possible runs for this word, including:

- $\rho = 0, 1, 2, 2, 2^{\omega}$, which *is not* accepting because $Inf(\rho) = \{2\}$ and so $Inf(\rho) \cap F = \emptyset$.
- $\rho = 0, 0, 0, 0, 0, 1, 1, 1, 1, 1^{\omega}$, which *is* accepting because $Inf(\rho) = \{1\}$ and so $Inf(\rho) \cap F \neq \emptyset$.

Because there is *some* accepting run for $babbbb^{\omega}$, we have $babbbb^{\omega} \in L(A)$. Note that

$$L(A) = \{\alpha \mid \alpha \text{ ends with only } b\text{'s}\}.$$

Example 8.18 Consider the (standard) Büchi automaton A given by:



with $\Sigma = \{a, b\}$, $Q = \{q, r\}$, $Q_0 = \{q\}$, $F = \{r\}$. What is $L(A)$?

Solution: State r is visited infinitely often if and only if the input word contains infinitely many b's. Thus,

$$L(A) = \{\alpha \mid \alpha \text{ contains infinitely many } b\text{'s}\}.$$

Generalized Büchi automata

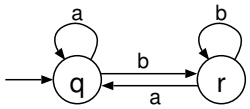
Definition 8.30 A *generalized Büchi automaton* is an ω-automaton with

- $\mathcal{F}$, a finite set of subsets of Q. A run ρ is accepting if and only if

$$Inf(\rho) \cap F \neq \emptyset, \quad \forall F \in \mathcal{F}$$

Note that a standard Büchi automaton is the special case of a generalized Büchi automaton where $\mathcal{F} = \{F\}$.

Example 8.19 Consider the generalized Büchi automaton A given by:



with $\mathcal{F} = \{\{q\}, \{r\}\}$. What is $L(A)$?

Solution: For example, input word $\alpha = abababababab...$ produces the run $\rho = q, r, q, r, q, r, q, r, ...$, with $Inf(\rho) = \{q, r\}$. Since $Inf(\rho) \cap \{q\} \neq \emptyset$ and $Inf(\rho) \cap \{r\} \neq \emptyset$, this is an accepting run and $\alpha \in L(A)$.

Notice that state q is visited infinitely often if and only if the input word contains infinitely many a's. State r is visited infinitely often if and only if the input word contains infinitely many b's. Since an accepting run must visit *both* state q and state r infinitely often,

$$L(A) = \{\alpha \mid \alpha \text{ contains infinitely many } a\text{'s and } b\text{'s}\}.$$

Also notice, this is different from using a standard Büchi automaton with $F = \{q, r\}$, which requires *either* that state q *or* state r is visited infinitely often.

Note: there are many other types of ω-automata, with various accepting conditions. For LTL model checking, we need only standard and generalized Büchi automata.

8.5.2 Some important algorithms for Büchi Automata

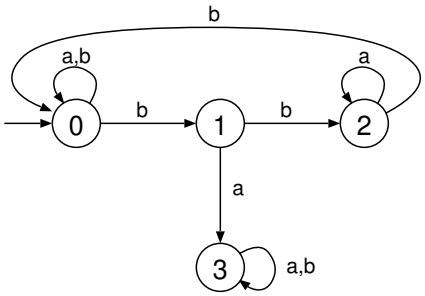
Language Emptiness

Definition 8.31 The *language emptiness problem*: given a standard Büchi automaton A, determine if $L(A) = \emptyset$.

This problem is *decidable*. Clearly, $L(A) \neq \emptyset$ if and only if there exists *some* accepting run ρ for some input word. An accepting run must visit some state $q \in F$ infinitely often. Since there are only finitely many states, this can happen only if there is a cycle containing q, and we can reach q from an initial state. Thus we have the following algorithm.

1. Treating the Büchi automaton as a directed graph (ignoring the symbols), determine the SCCs.
2. For any state in F that belongs to a non-trivial SCC, mark it as *green*.
3. For any state that can reach a green state, mark it as green.
4. $L(A) \neq \emptyset$ iff there exists a green state in Q_0.

Example 8.20 Consider the (standard) Büchi automaton A given by:

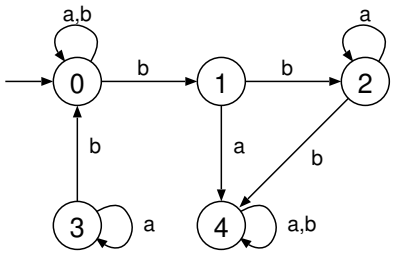


with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $Q_0 = \{0\}$, $F = \{1\}$. Is $L(A) = \emptyset$?

Solution: Running the algorithm:

1. The non-trivial SCCs are $\{0, 1, 2\}$ and $\{3\}$.
2. State $1 \in F$ belongs to a non-trivial SCC, so mark it as green.
3. States 0 and 2 can reach 1, so mark those as green.
4. Since $0 \in Q_0$ and 0 is green, we conclude that $L(A) \neq \emptyset$.

Example 8.21 Consider the (standard) Büchi automaton A given by:



with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3, 4\}$, $Q_0 = \{0\}$, $F = \{1, 3\}$. Is $L(A) = \emptyset$?

Solution: Running the algorithm:

1. The non-trivial SCCs are $\{0\}$, $\{2\}$, $\{3\}$, and $\{4\}$.

2. State $3 \in F$ belongs to a non-trivial SCC, so mark it as green. (State 1 does not.)
3. No other states can reach a green state.
4. Since $\{0\} = Q_0$ and 0 is not green, we conclude that $L(A) = \emptyset$.

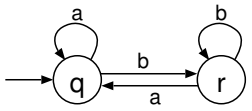
Converting from generalized to standard Büchi

Given a generalized Büchi automaton A, can we build a standard Büchi automaton A' such that $L(A) = L(A')$?

Answer: YES. First, the idea.

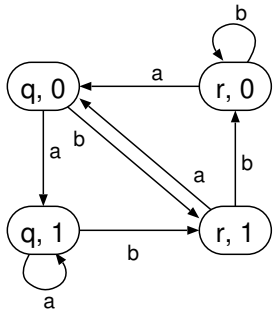
- Suppose the generalized Büchi automaton has $\mathcal{F} = \{F_0, F_1, ..., F_{n-1}\}$ i.e., n sets of states that must be visited infinitely often. Then, our standard Büchi automaton state will be: which generalized Büchi automaton state we're in, plus a counter that goes from 0 to $n-1$.
- When the counter is i, if we visit a state in F_i, then increment the counter (modulo n) when we consume the next input symbol.
- When the counter wraps around to 0, we have seen a state in F_i for all i.
- Because each set F_i is finite, if the counter wraps around to 0 infinitely many times, then the generalized Büchi acceptance condition is met.

Example 8.22 Consider the generalized Büchi automaton A from Example 8.19:



with $\mathcal{F} = \{F_0, F_1\}$, $F_0 = \{q\}$, $F_1 = \{r\}$. Build an equivalent standard Büchi automaton.

Solution We can add a counter with values $\{0, 1\}$. If the counter is 0 and the state is q, then outgoing edges should set the counter to 1. If the counter is 1 and the state is r, then outgoing edges should set the counter to 0. That gives us:



Note: between any two visits to state $(q, 0)$, we must visit state $(r, 1)$ because all paths from $(q, 0)$ to $(q, 0)$ pass through $(r, 1)$. Thus, if $(q, 0)$ is visited infinitely often, so is $(r, 1)$. So our standard Büchi automaton can use accepting condition $F = \{(q, 0)\}$.

Given a generalized Büchi automaton $A = (\Sigma, Q, Q_0, \Delta, \mathcal{F})$ with $\mathcal{F} = \{F_0, ..., F_{n-1}\}$, we can build the standard Büchi automaton $A' = (\Sigma, Q', Q'_0, \Delta', F')$ with

- $Q' = Q \times \{0, 1, ..., n-1\}$, i.e., state plus the counter
- $Q'_0 = Q_0 \times \{0\}$, i.e., the counter starts at 0
- $F' = F_0 \times \{0\}$
- $(q_i, c), a, (q_j, d) \in \Delta'$ if and only if both:
  - $(q_i, a, q_j) \in \Delta$
  - If $q_i \in F_i$ and $c = i$, then $d = (c+1) \bmod n$, otherwise $d = c$.

  where the first part of the rule handles edges in the original automaton and the second part of the rule handles the counter.

It can be shown that $L(A) = L(A')$.

Product construction

Given two standard Büchi automata

- $A_1 = (\Sigma, Q_1, Q_{01}, \Delta_1, F_1)$
- $A_2 = (\Sigma, Q_2, Q_{02}, \Delta_2, F_2)$

define the product automaton $A_1 \times A_2 = (\Sigma, Q, Q_0, \Delta, \mathcal{F})$ as

- $Q = Q_1 \times Q_2$
- $Q_0 = Q_{01} \times Q_{02}$
- $\mathcal{F} = \{F_1 \times Q_2, Q_1 \times F_2\}$
- $(q_1, q_2), a, (q'_1, q'_2) \in \Delta$ if and only if $q_1, a, q'_1 \in \Delta_1$ and $q_2, a, q'_2 \in \Delta_2$.

Note that this is a *generalized* Büchi automaton.

Property 8.32 Word α produces run $\rho_1 = q_0, q_1, q_2, ...$ in A_1, and produces run $\rho_2 = r_0, r_1, r_2, ...$ in A_2, if and only if it produces run $\rho_1 \times \rho_2 = (q_0, r_0), (q_1, r_1), ...$ in $A_1 \times A_2$.

Proof $\rightarrow$: Let ρ_1 be a run for α in A_1, and ρ_2 be a run for α in A_2. By definition of a run, we have $(q_i, a, q_{i+1}) \in \Delta_1 \ \forall i$ and $(r_i, a, r_{i+1}) \in \Delta_2 \ \forall i$. By construction of Δ, we have $((q_i, r_i), a, (q_{i+1}, r_{i+1})) \in \Delta \ \forall i$. Thus $\rho_1 \times \rho_2$ is a run for α in $A_1 \times A_2$.

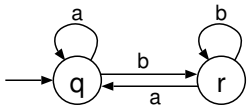
Proof $\leftarrow$: Let $\rho_1 \times \rho_2 = (q_0, r_0), (q_1, r_1), ...$ be a run for α in $A_1 \times A_2$. Then we have $((q_i, r_i), a, (q_{i+1}, r_{i+1})) \in \Delta \ \forall i$. But by definition of Δ, this says $(q_i, a, q_{i+1}) \in \Delta_1 \ \forall i$ and $(r_i, a, r_{i+1}) \in \Delta_2 \ \forall i$. Thus ρ_1 is a run for α in A_1, and ρ_2 is a run for α in A_2.

Property 8.33 $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$.

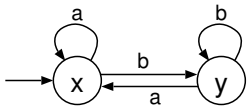
Proof:

$$\begin{array}{rcl}
\alpha \in L(A_1 \times A_2) & \leftrightarrow & \text{There is an accepting run } \rho_1 \times \rho_2 \text{ on } \alpha \text{ in } A_1 \times A_2 \\
& \leftrightarrow & \text{Run } \rho_1 \times \rho_2 \text{ on } \alpha \text{ in } A_1 \times A_2 \text{ has} \\
& & \mathit{Inf}(\rho_1 \times \rho_2) \cap (F_1 \times Q_2) \neq \emptyset \text{ and} \\
& & \mathit{Inf}(\rho_1 \times \rho_2) \cap (Q_1 \times F_2) \neq \emptyset \\
& \leftrightarrow & \text{Run } \rho_1 \text{ on } \alpha \text{ in } A_1 \text{ has } \mathit{Inf}(\rho_1) \cap F_1 \neq \emptyset \\
& & \text{and run } \rho_2 \text{ on } \alpha \text{ in } A_2 \text{ has } \mathit{Inf}(\rho_2) \cap F_2 \neq \emptyset \\
& \leftrightarrow & \text{There is an accepting run } \rho_1 \text{ on } \alpha \text{ in } A_1, \text{ and} \\
& & \text{there is an accepting run } \rho_2 \text{ on } \alpha \text{ in } A_2 \\
& \leftrightarrow & \alpha \in L(A_1) \text{ and } \alpha \in L(A_2)
\end{array}$$

Example 8.23 Consider the Büchi automaton A_1 given by:



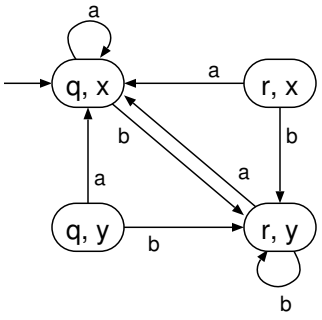
with $F = \{r\}$, and the Büchi automaton A_2 given by:



with $F = \{x\}$. It can be seen that

$$\begin{array}{rcl}
L(A_1) & = & \{\alpha \mid \alpha \text{ contains infinitely many } b\text{'s}\} \\
L(A_2) & = & \{\alpha \mid \alpha \text{ contains infinitely many } a\text{'s}\}
\end{array}$$

The product automaton $A_1 \times A_2$ is given by:

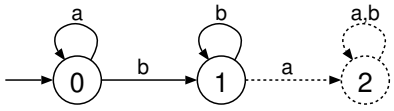


with $\mathcal{F} = \{\{rx, ry\}, \{qx, rx\}\}$, and note that

$$L(A_1 \times A_2) \;=\; \{\alpha \mid \alpha \text{ contains infinitely many } a\text{'s and } b\text{'s}\}.$$

Also notice that states (q, y) and (r, x) are unreachable! If we eliminate those states, we obtain a generalized Büchi automaton that is essentially the same as the one in Example 8.19.

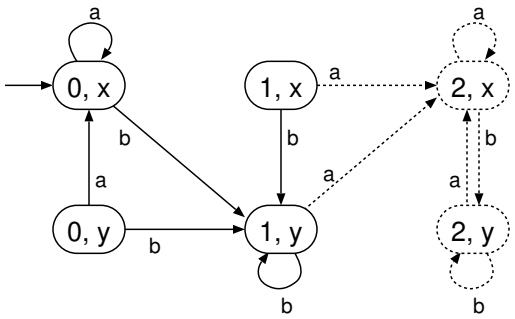
Example 8.24 Consider the Büchi automaton A_3 given by:



with $F = \{1\}$. It can be seen that

$$L(A_3) \;=\; \{\alpha \mid \alpha \text{ contains finitely many } a\text{'s, followed by infinitely many } b\text{'s}\}.$$

Note that we might draw this without state 2; any run in which symbol a is consumed in state 1 would either produce a finite run, or go to an imaginary "bad" state (in this case, state 2) from which it is impossible to reach any state in F. In any case, such a run would not be an accepting run. For this example, we include state 2 with dotted lines to show that its omission does not affect the result.
Using A_2 from Example 8.23, the product automaton $A_3 \times A_2$ is:



with $\mathcal{F} = \{\{\text{states with } 1\}, \{\text{states with } x\}\} = \{F_1, F_2\}$, where $F_1 = \{(1,x),(1,y)\}$ and if state 2 is omitted, $F_2 = \{(0,x),(1,x)\}$ otherwise $F_2 = \{(0,x),(1,x),(2,x)\}$. Now, state $(1,x)$ cannot be reached from the initial state, so to satisfy F_1 we must visit state $(1,y)$ infinitely often. But we cannot do this *and* visit state $(0,x)$ or $(2,x)$ infinitely often. So, there is no accepting run for this automaton. This makes sense, because

$$\begin{aligned} L(A_3 \times A_2) \;=\; \{\alpha \;\mid\; & \alpha \text{ contains finitely many } a\text{'s, followed by infinitely many } b\text{'s} \\ & \textit{and } \alpha \text{ contains infinitely many } a\text{'s}\} \;=\; \emptyset. \end{aligned}$$

8.5.3 LTL model checking

LTL model checking using Büchi automata is based on the following idea[3]:

1. We will use the alphabet $\Sigma = 2^{\mathcal{P}}$, i.e., each symbol is a subset of $\mathcal{P}$, the set of atomic propositions. The infinite words as input can be thought of as encoding the set of propositions that hold in each step.

2. Build a standard Büchi automaton A_M from the Kripke structure M, to encode the paths possible from the Kripke structure. More specifically we build A_M such that

$$\begin{aligned} \mathit{Lang}(A_M) \;=\; \{\alpha \;\mid\; & \text{There is a path } \pi \text{ through the Kripke structure that produces} \\ & \text{the sequence of sets of atomic propositions } \alpha\text{; formally} \\ & L(\pi_i) = \alpha_i, \forall i\}. \end{aligned}$$

[3]M. Vardi and P. Wolper. "An automata–theoretic approach to automatic program verification". In *LICS'1986*, pp. 332–344, 1986.

We will discuss this in more detail, below.

3. For path formula ψ, build a standard Büchi automaton A_ψ, to encode the paths (in any Kripke structure) that satisfy ψ. More specifically,

$$Lang(A_\psi) \;=\; \{\alpha \quad | \quad \text{If path } \pi \text{ has } L(\pi_i) = \alpha_i, \forall i \text{ then } \pi \models \psi\}$$

We will discuss this in more detail, below. Note that the algorithm will give a *generalized* Büchi automaton, and we may need to run the conversion algorithm to get an equivalent *standard* Büchi automaton.

4. Build the product automaton $A_M \times A_\psi$. Note that $Lang(A_M \times A_\psi)$ corresponds to the set of paths both through the Kripke structure M and that satisfy ψ.

5. Do a language emptiness check on automaton $A_M \times A_\psi$. We have $M \models \mathsf{E}\,\psi$ if and only if $Lang(A_M \times A_\psi) \neq \emptyset$.

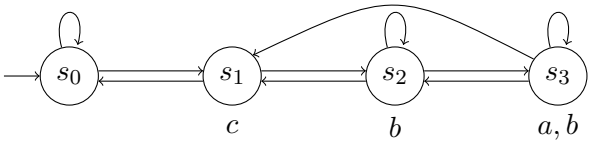
We have already discussed all the steps above, except for step (2) which is easy, and step (3) which is difficult.

Kripke to Büchi conversion:

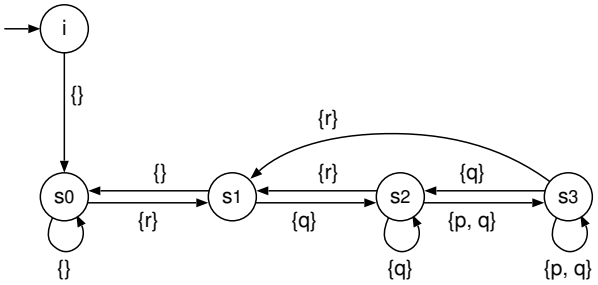
Informally, the idea is to make a copy of the Kripke structure, except we add a new state i which serves as the initial state, and edges in the Büchi automaton are labeled with the set of atomic propositions that hold in the destination state. Formally, given a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ with atomic propisitions $\mathcal{P}$, build Büchi automaton $A_M = (\Sigma, Q, Q_0, \Delta, F)$ with

- $\Sigma = 2^{\mathcal{P}}$ (symbols are *sets* of atomic propositions)
- $Q = \mathcal{S} \cup \{i\}$ (i is a new initialization state)
- $Q_0 = \{i\}$
- $F = Q$ (every state is accepting)
- $(s, L(s'), s') \in \Delta$ if and only if $(s, s') \in \mathcal{R}$ or $s = i, s' \in \mathcal{S}_0$.

Example 8.25 Using Kripke structure for the CD player from Example 3.1 shown below,



we build the following Büchi automaton:



Since all states are accepting, all runs are accepting runs, and correspond to paths in the Kripke structure. For example, the Kripke structure path

$$s_0, s_1, s_2, s_1, s_2, s_1, s_2, ...$$

produces the sequence of sets

$$\{\}, \{r\}, \{q\}, \{r\}, \{q\}, \{r\}, \{q\}, ...$$

and note that this word is accepted by the Büchi automaton.

Path formula to Büchi conversion:

The idea here is:

- Use an algorithm that recursively processes formula ψ, and builds a graph
- From the graph, we will build a generalized Büchi automaton. Essentially, we will need to label the graph edges and determine the set $\mathcal{F}$.
- Formula ψ cannot contain F or G operators (we can rewrite these using U).
- Formula ψ must be written in "negation normal form", which means negations are allowed only directly before atomic propositions. Any formula can be written this way, using the following transformations:

$$\begin{aligned}
\neg(\psi_1 \wedge \psi_2) &\equiv (\neg\psi_1) \vee (\neg\psi_2) \\
\neg(\psi_1 \vee \psi_2) &\equiv (\neg\psi_1) \wedge (\neg\psi_2) \\
\neg\mathsf{X}\,\psi &\equiv \mathsf{X}\,\neg\psi \\
\neg(\psi_1\,\mathsf{U}\,\psi_2) &\equiv (\neg\psi_1)\,\mathsf{R}\,(\neg\psi_2) \\
\neg(\psi_1\,\mathsf{R}\,\psi_2) &\equiv (\neg\psi_1)\,\mathsf{U}\,(\neg\psi_2)
\end{aligned}$$

  Operator R is the "release" operator, which we have not seen before, and is defined as the dual of U. Formally,

$$\pi \models p\,\mathsf{R}\,q \quad \text{iff} \quad \forall j \geq 0, \text{if } \forall 0 \leq i < j, \pi^i \not\models p \text{ then } \pi^j \models q.$$

  and note that if we take the negation of this, we have[4]

$$\begin{aligned}
\pi \not\models p\,\mathsf{R}\,q &\quad \text{iff} \quad \neg\left(\forall j \geq 0, (\forall 0 \leq i < j, \pi^i \not\models p) \rightarrow (\pi^j \models q)\right) \\
\pi \not\models p\,\mathsf{R}\,q &\quad \text{iff} \quad \exists j \geq 0,\ \neg\left((\forall 0 \leq i < j, \pi^i \not\models p) \rightarrow (\pi^j \models q)\right) \\
\pi \not\models p\,\mathsf{R}\,q &\quad \text{iff} \quad \exists j \geq 0, (\forall 0 \leq i < j, \pi^i \not\models p) \wedge (\pi^j \not\models q) \\
\pi \not\models p\,\mathsf{R}\,q &\quad \text{iff} \quad \exists j \geq 0, (\pi^j \models \neg q) \wedge (\forall 0 \leq i < j, \pi^i \models \neg p) \\
\pi \not\models p\,\mathsf{R}\,q &\quad \text{iff} \quad \pi \models \neg p\,\mathsf{U}\,\neg q.
\end{aligned}$$

  Also, note that

$$\mathsf{G}\,q \equiv \neg\mathsf{F}\,\neg q \equiv \neg(\mathtt{tt}\,\mathsf{U}\,\neg q) \equiv \mathtt{ff}\,\mathsf{R}\,q$$

  which also follows easily from the definition of R above.

  Since all the above transformations "push negations inside", we can apply them repeatedly until all negations are in front of atomic propositions.

[4]Recall that $\neg(a \rightarrow b) \equiv a \wedge \neg b$.

Example 8.26 Write $\mathsf{FG}\,p$ in negative normal form, without operators F and G.

Solution:

$$\mathsf{FG}\,p \;\equiv\; \mathtt{tt}\,\mathsf{U}\,(\mathtt{ff}\,\mathsf{R}\,p)$$

Example 8.27 Write $\neg\mathsf{FG}\,p$ in negative normal form, without operators F and G.

Solution:

$$\neg\mathsf{FG}\,p \;\equiv\; \neg\,(\mathtt{tt}\,\mathsf{U}\,(\mathtt{ff}\,\mathsf{R}\,p)) \;\equiv\; \mathtt{ff}\,\mathsf{R}\,\neg(\mathtt{ff}\,\mathsf{R}\,p) \;\equiv\; \mathtt{ff}\,\mathsf{R}\,(\mathtt{tt}\,\mathsf{U}\,\neg p)$$

The translation algorithm builds a graph, where each node in the graph has the following data.

- **Old:** the set of (sub)formulas already processed for the node.
- **New:** the set of (sub)formulas still to process for the node.
- **Next:** the set of (sub)formulas that must hold in the next node in the graph.
- **Incoming:** the set of edges pointing to this node.

Note that for a node n, $n.Old \cup n.New$ is the set of formulas that should hold on paths starting here. The algorithm is given in detail below. The idea is to "expand" nodes, recursively. At each step, we will either

- replace a node with a new one (we will modify nodes "in place" instead); or
- split a node in half, which happens whenever there is an "OR". In this case we "clone" a node by copying all data, and after splitting we further expand the nodes, separately.

Algorithm 8.1 ψ to graph translation

```
nodeset translate(formula f)
{
    n ← new node;
    n.Old ← ∅;
    n.New ← {f};
    n.Next ← ∅;
    n.Incoming ← {init};
    return expand(n, ∅);
}

nodeset expand(node n, nodeset N)
{
    if n.New == ∅ then
        if ∃m ∈ N, m.Old == n.Old ∧ m.Next == n.Next then
            // m duplicates n; use m instead and redirect edges
            m.Incoming ← m.Incoming ∪ n.Incoming;
            // discard n by not adding it to N here
            return N;
        else
            // Add n to the graph and keep processing
            N ← N ∪ {n};
```

```
            n' ← new node;
            n'.Old ← ∅;
            n'.New ← n.Next;
            n'.Next ← ∅;
            n'.Incoming ← {n};
            return expand(n', N);
        endif
    endif

    // n.New is not empty; more processing to do

    choose some f ∈ n.New;
    n.New ← n.New \ {f};
    if f ∈ n.Old then
        // No need to process f again
        return expand(n, N);
    endif
    n.Old ← n.Old ∪ {f};

    // Remainder of algorithm: process f based on the type of formula

    if f ∈ P or ¬f ∈ P or f == ff or f == tt then
        // f is a trivial formula
        if f == ff or ¬f ∈ n.Old then
            // Logical impossibility, discard this node
            return N;
        else
            // No additional processing needed for this formula, keep expanding
            return expand(n, N);
        endif

    elseif f == g ∧ h then
        n.New ← n.New ∪ {g, h};
        return expand(n, N);

    elseif f == g ∨ h then
        // Split
        n' ← clone of n; // Copy all data from n into new node n'
        n.New ← n.New ∪ {g};
        n'.New ← n'.New ∪ {h};
        return expand(n, expand(n', N));

    elseif f == X g then
        n.Next ← n.Next ∪ {g};
        return expand(n, N);
```

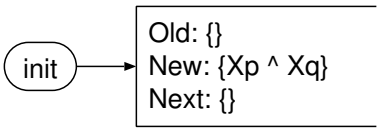
```
    elseif f == g U h then
        // Split, using recurrence    g U h  ≡  h ∨ (g ∧ X(g U h))
        n' ← clone of n;
        n.New ← n.New ∪ {h};
        n'.New ← n'.New ∪ {g};
        n'.Next ← n'.Next ∪ {g U h};
        return expand(n, expand(n', N));

    elseif f == g R h then
        // Split, using recurrence    g R h  ≡  (g ∧ h) ∨ (h ∧ X(g R h))
        n' ← clone of n;
        n.New ← n.New ∪ {g, h};
        n'.New ← n'.New ∪ {h};
        n'.Next ← n'.Next ∪ {g R h};
        return expand(n, expand(n', N));
    endif
}
```

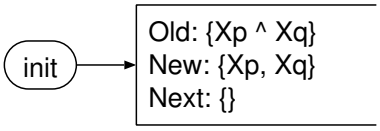
Example 8.28 Run the translation algorithm on the formula $\psi = \mathsf{X}\,p \wedge \mathsf{X}\,q$.

Solution:

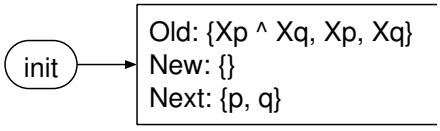
1. Start with a node "init", pointing to a new node n with $n.New = \{\mathsf{X}\,p \wedge \mathsf{X}\,q\}$, and then call expand() on that node:



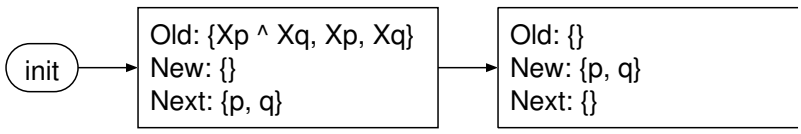
2. Choose formula $f = \mathsf{X}\,p \wedge \mathsf{X}\,q$ from $n.New$; remove it from $n.New$, add it to $n.Old$, and then process the "and": add $\mathsf{X}\,p$ and $\mathsf{X}\,q$ to $n.New$. That gives us:



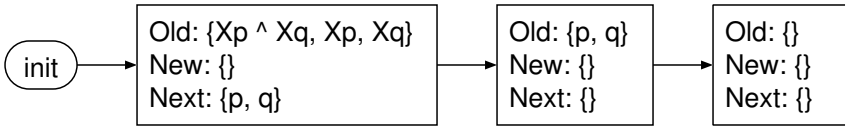
3. Choose (say) formula $f = \mathsf{X}\,p$ from $n.New$; remove it from $n.New$, add it to $n.Old$, and process the "X": add p to $n.Next$. Do the same for $\mathsf{X}\,q$. That gives us:



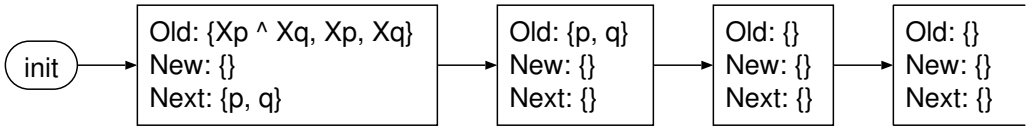
4. Since $n.New$ is now empty, we complete the node by adding it to the graph and adding a child node n' with $n'.New = n.Next$. That gives us:



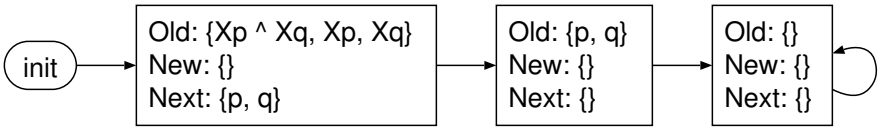
5. Choose formula $f = p$ and process it (just move it from *New* to *Old*). Do the same with $f = q$. Since *New* is now empty, complete the node by adding it to the graph and adding a child node n' with $n'.New = n.Next$. That gives us:



6. Since this node has an empty *New*, complete it by adding it to the graph and adding a child node. That gives us:



7. When this node is completed, we discover that it duplicates its parent node, so we discard it and redirect all incoming edges to the parent node. We terminate with the graph:



Now, how do we convert the graph into a Büchi automaton? Informally, the idea is:

- Label edges with all subsets of atomic propositions that are consistent with propositions and their negations listed in the destination node's *Old* set.
- Define an accepting condition similar to the tableau rule where until formulas must be eventually satisfied.

Formally, if we use Algorithm 8.1 to build a nodeset N for a formula ψ, we build the Büchi automaton $(\Sigma, Q, Q_0, \Delta, \mathcal{F})$ as follows.

- $\Sigma = 2^{\mathcal{P}}$
- $Q = N \cup \{init\}$
- $Q_0 = \{init\}$
- $(q, A, q') \in \Delta$ if and only if $q \in q'.Incoming$ (i.e., there's an edge from q to q' in the graph) AND $A \subseteq \mathcal{P}$ is a set of propositions that satisfy the conjunction of all propositions and negated propositions in $q'.Old$.

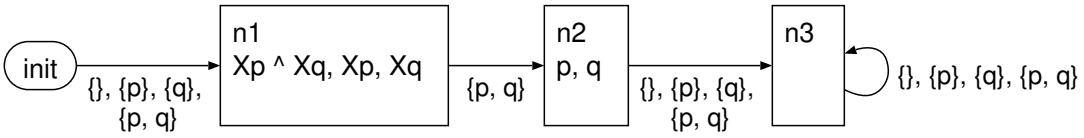
- $\mathcal{F} = \{F_1, F_2, ..., F_u\}$ where each F_i is an accepting set for some until subformula and u is the total number of until subformulas. Specifically, for each subformula $g \,\mathsf{U}\, h$, add an accepting set

$$F_i = \{q \mid h \in q.Old \quad \vee \quad g \,\mathsf{U}\, h \notin q.Old\}$$

  If there are no until subformulas, then use $\mathcal{F} = \{Q\}$.

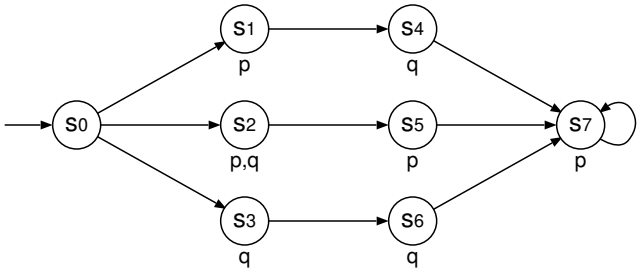
Example 8.29 What is the Büchi automaton for $\psi = X\, p \wedge X\, q$?

Solution: Convert the graph obtained in Example 8.28 into a Büchi automaton. That gives us

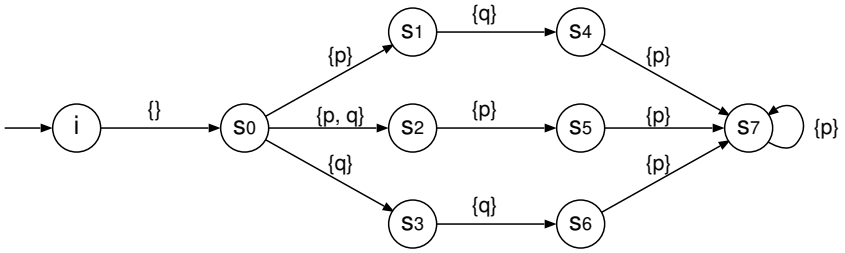


where each node is given a name and we show its *Old* set. Note that $\Sigma = \{\{\}, \{p\}, \{q\}, \{p, q\}\}$ and $\mathcal{F} = \{init, n1, n2, n3\}$ (making this a standard Büchi automaton) because there are no until formulas.

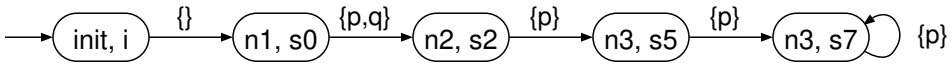
Example 8.30 Does the Kripke structure below satisfy $\mathsf{E}\,(\mathsf{X}\, p \wedge \mathsf{X}\, q)$?



Solution: First, convert the Kripke structure into a Büchi automaton:



Then, take the product of this and the Büchi automaton obtained in Example 8.29. If we start from the initial state and consider only states that can be reached, we obtain:



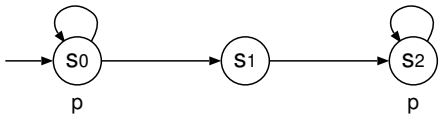
with $\mathcal{F} = \{F_\psi \times Q_K, Q_\psi \times F_K\}$ where Q_K, Q_ψ are the states in the automata for the Kripke structure and the formula ψ, and F_K, F_ψ are the accepting states in the automata for the Kripke structure and the formula ψ. In this case, $F_K = Q_K$ and $F_\psi = Q_\psi$ and so we have $\mathcal{F} = \{\text{set of all states}\}$. Thus, this is also a standard Büchi automaton where F is the set of all states.

Finally, we check if the langage accepted by this automaton is empty. Running the algorithm:

1. There is one non-trivial SCC, namely the state (n_3, s_7).
2. This state belongs to F, so mark it as green.
3. All states can reach this one, so mark everything else as green.
4. The initial state is green, thus $L(A) \neq \emptyset$.

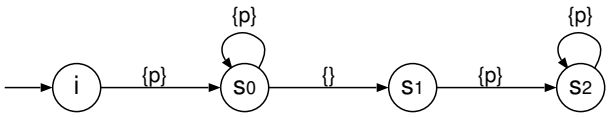
Because the language is non-empty, there exists a path satisfying the formula. Thus, the model satisfies $\mathsf{E}(\mathsf{X}\,p \wedge \mathsf{X}\,q)$.

Example 8.31 Use Büchi automata to show this model satisfies $\mathsf{A}\,\mathsf{FG}\,p$.



Solution: We know from Example 8.5 that the property holds. Following the steps:

1. The alphabet we need is $\Sigma = \{\{\}, \{p\}\}$.
2. Build a standard Büchi automaton A_M for the Kripke structure:



   Recall that F is the set of all states.

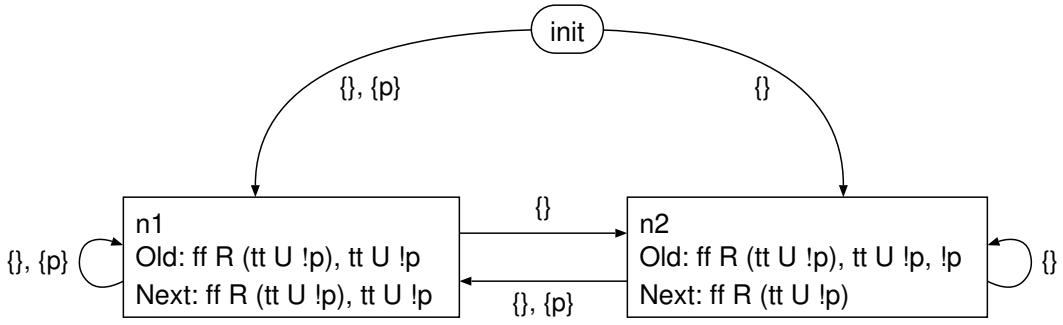
3. Build a standard Büchi automaton A_ψ. First, we need to convert our formula into an "existence" property:

$$\mathsf{A}\,\mathsf{FG}\,p \;\equiv\; \neg\mathsf{E}\,\neg\mathsf{FG}\,p$$

   Now, write $\psi = \neg\mathsf{FG}\,p$ in negative normal form, without operators F and G. From Example 8.27, we have

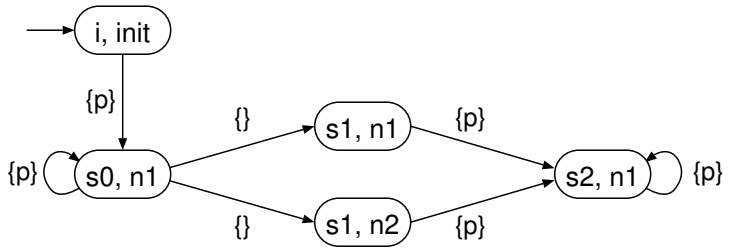
$$\mathsf{A}\,\mathsf{FG}\,p \;\equiv\; \neg\mathsf{E}\,\neg\mathsf{FG}\,p \;\equiv\; \neg\mathsf{E}\,\mathtt{ff}\,\mathsf{R}\,(\mathtt{tt}\,\mathsf{U}\,\neg p)$$

   Next, run the translation algorithm for $\psi = \neg\mathsf{E}\,\mathtt{ff}\,\mathsf{R}\,(\mathtt{tt}\,\mathsf{U}\,\neg p)$. This gives us the automaton:



   There is a single until formula, $\mathtt{tt}\,\mathsf{U}\,\neg p$, giving us an accepting set of states whose *Old* set either contains $\neg p$, or does not contain $\mathtt{tt}\,\mathsf{U}\,\neg p$. Thus $\mathcal{F} = \{\{n_2\}\}$ or equivalently we have a standard Büchi automaton with $F = \{n_2\}$.

4. Build the product automaton $A_M \times A_\psi$, which gives us (if we eliminate unreachable states):



   with acceptance condition $\mathcal{F} = \{\{\text{all states}\}, \{\text{states with } n_2\}\}$ which is equivalent to a standard Büchi automaton with $F = \{(s_1, n_2)\}$.

5. Is $L(A_M \times A_\psi)$ empty?

   YES: there is no way to visit (s_1, n_2) infinitely often.

   $\rightarrow$ the model *does not* satisfy $\mathsf{E}\,\psi \;=\; \mathsf{E}\,\neg\mathsf{FG}\,p$

   $\rightarrow$ the model *does* satisfy $\neg\mathsf{E}\,\psi \;=\; \neg\mathsf{E}\,\neg\mathsf{FG}\,p \;=\; \mathsf{A}\,\mathsf{FG}\,p$

Complexity of LTL model checking with Büchi automata:

1. The size of the alphabet is $\mathcal{O}(2^{|\mathcal{P}|})$, which can be represented with $|\mathcal{P}|$ bits.
2. The size of A_M and the cost to build it is linear in the size of the Kripke structure: $\mathcal{O}(|\mathcal{S}|+|\mathcal{R}|)$.
3. It can be shown that the worst case size of A_ψ is $\mathcal{O}(2^{|\psi|})$.
4. The size of $A_M \times A_\psi$ (and the cost to build it) is at most the product of the sizes of A_M and A_ψ, which is at most $\mathcal{O}(2^{|\psi|}(|\mathcal{S}| + |\mathcal{R}|))$.
5. Checking for language emptiness can be done in time that is linear in the size of the automaton we are checking.

Thus, using Büchi automata has the same worst-case complexity as using the tableau method.

8.6 Is a faster algorithm possible?

Both the tableau–based methods and Büchi–based methods for LTL model checking have worst-case complexity $\mathcal{O}(2^{|\psi|}(|\mathcal{S}|+|\mathcal{R}|))$. Can we do better? How "hard" is LTL model checking, anyway? We will look at a fragment of LTL in some detail, and summarize other results from the literature[5].

Definition 8.34 $\Phi(\mathcal{X})$ denotes the set of all path formulas using operators from the set $\mathcal{X}$.

For example, given any LTL path formula, we can find an equivalent one in the set $\Phi(\neg, \wedge, \mathsf{X}, \mathsf{U})$.

[5]Some references:

- A. Sistla and E. Clarke, "The Complexity of Propositional Linear Temporal Logics". *Journal of the Association for Computing Machinery* 32 (3), pp. 733–749. 1985.
- M. Bauland *et al.*, "The Tractability of Model Checking for LTL: The Good, the Bad, and the Ugly Fragments". *ACM Transactions on Computational Logic* 12 (2). 2011.

8.6.1 Model checking $\Phi(\neg, \wedge, \mathsf{F})$

Definition 8.35 For any sequence of states $\sigma = (s_0, s_1, s_2, ...)$, define:

$Inf(\sigma)$**:** the set of states appearing infinitely often in σ

$tail(\sigma)$**:** the smallest j such that $s_i \in Inf(\sigma)$, $\forall i \geq j$

$size(\sigma) = tail(\sigma) + |Inf(\sigma)|$

Property 8.36 For any paths $\pi = (p_0, p_1, ...)$ and $\sigma = (s_0, s_1, ...)$ with $tail(\pi) = tail(\sigma)$, $p_0 = s_0$, $p_j = s_j$ $\forall j < tail(\pi)$, and $Inf(\pi) = Inf(\sigma)$, then for all $\phi \in \Phi(\neg, \wedge, \mathsf{F})$,

1. $\forall i < tail(\pi)$,
$$\pi^i \models \phi \quad \text{if and only if} \quad \sigma^i \models \phi.$$

2. $\forall i, i' \geq tail(\pi)$, with $p_i = s_{i'}$,
$$\pi^i \models \phi \quad \text{if and only if} \quad \sigma^{i'} \models \phi.$$

Proof by induction on formula structure: In the base case, ϕ is an atomic proposition, and the result holds trivially.
Now, assume it holds for all subformulas of ϕ, and prove it holds for ϕ. The cases $\phi = \neg\phi_1$, $\phi = \phi_1 \wedge \phi_2$ are trivial. Now, consider $\phi = \mathsf{F}\,\phi_1$ and suppose $\pi^i \models \mathsf{F}\,\phi_1$. Then by definition, there is a $j \geq i$ such that $\pi^j \models \phi_1$.

1. Suppose $i < tail(\pi)$. If $j < tail(\pi)$, then we have $p_j = s_j$ and by the inductive hypothesis, $\sigma^j \models \phi_1$; thus, $\sigma^i \models \mathsf{F}\,\phi_1$. Otherwise, $j \geq tail(\pi)$, and we have $p_j \in Inf(\pi) = Inf(\sigma)$. Thus, there exists a $j' \geq tail(\pi)$ such that $s_{j'} = p_j$. By the inductive hypothesis, $\sigma^{j'} \models \phi_1$, and since $j' > i$, $\sigma^i \models \mathsf{F}\,\phi_1$.

2. Suppose $i \geq tail(\pi)$. Then both p_i and p_j appear infinitely often in π and in σ. Thus, for any i' such that $p_i = s_{i'}$, there exists $j' \geq i'$ such that $p_j = s_{j'}$. By the inductive hypothesis, $\sigma^{j'} \models \phi_1$, thus $\sigma^{i'} \models \mathsf{F}\,\phi_1$.

Property 8.37 (Sistla and Clarke) There is a path satisfying $\psi \in \Phi(\neg, \wedge, \mathsf{F})$ if and only if there is a path $\pi \models \psi$ with $size(\pi) \leq |\psi|$.

Property 8.38 (Sistla and Clarke) The problem of checking whether some path satisfies a formula ψ in $\Phi(\neg, \wedge, \mathsf{F})$ is in NP.

Proof: We give a nondeterministic algorithm that verifies if a path satisfies ψ, as follows.

1. Guess a finite sequence $(s_0, s_1, ..., s_n)$, these will form the first part of the path π.

2. Guess the states in $Inf(\pi)$. From Property 8.36, the path will satisfy ψ or not, regardless of the order in which these states are visited. Also, note that Property 8.37 says that the total number of guesses in these first two steps is at most the size of formula ψ.

3. Verify that $(s_0, s_1, ..., s_n)$ is a valid path in the Kripke structure, and that there is an edge from s_n to some state in $Inf(\pi)$.

4. Verify that the subgraph of the Kripke structure, containing only the states $Inf(\pi)$, is strongly–connected.

5. Label the states in $(s_0, ..., s_n)$ and all states in $Inf(\pi)$ using the following recursive algorithm.

   (a) For atomic proposition p, state s is labeled with p iff $p \in L(s)$.

   (b) For $\phi = \neg\phi_1$, label s with ϕ iff s is not labeled with ϕ_1.

   (c) For $\phi = \phi_1 \wedge \phi_2$, label s with ϕ iff s is labeled with ϕ_1 and ϕ_2.

   (d) For $\phi = \mathsf{F}\,\phi_1$, if some $s' \in Inf(\pi)$ is labeled with ϕ_1, then label all s with ϕ. If s_i is labeled with ϕ_1, then label all s_j with ϕ, for $j \leq i$.

6. The algorithm returns "yes" if s_0 is labeled with ψ.

Note that the above algorithm requires polynomial time in the size of the Kripke structure, and of the formula ψ.

Property 8.39 (Sistla and Clarke) The problem of checking whether some path satisfies a formula ψ in $\Phi(\neg, \wedge, \mathsf{F})$ is NP–Complete.

Proof: From Property 8.38, we know this problem is in NP. We complete the proof by showing how an existing NP–Complete problem (namely, 3SAT) can be transformed to checking a formula in $\Phi(\neg, \wedge, \mathsf{F})$.

Suppose we are given a 3SAT instance with variables $x_1, x_2, ..., x_n$, and formula $f = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_i is the disjunction of exactly three literals:

$$C_i \quad = \quad l_{i,1} \vee l_{i,2} \vee l_{i,3}$$

where $l_{i,j}$ is either x_k or $\neg x_k$, for some $1 \leq k \leq n$.
We build a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ as follows, with atomic propositions $c_1, ..., c_m$.

- $\mathcal{S} = \{y_0, x_0, x'_0, y_1, x_1, x'_1, ..., x_n, x'_n, y_n\}$
- $\mathcal{S}_0 = \{y_0\}$
- $\mathcal{R} = \{(y_0, x_0), (y_0, x'_0), (x_0, y_1), (x'_0, y_1), ..., (y_{n-1}, x_n), (y_{n-1}, x'_n), (x_n, y_n), (x'_n, y_n), (y_n, y_n)\}$
- $L(y_i) = \emptyset$
- $c_j \in L(x_i)$ if and only if $l_{j,1} = x_i$ or $l_{j,2} = x_i$ or $l_{j,3} = x_i$ (clause C_j contains x_i).
- $c_j \in L(x'_i)$ if and only if $l_{j,1} = \neg x_i$ or $l_{j,2} = \neg x_i$ or $l_{j,3} = \neg x_i$ (clause C_j contains $\neg x_i$).

It can be proved that f is satisfiable if and only if some path satisfies the formula

$$\psi = \mathsf{F}\,c_1 \wedge \mathsf{F}\,c_2 \wedge \cdots \wedge \mathsf{F}\,c_m.$$

Note that this transformation is linear in the size of the 3SAT instance.

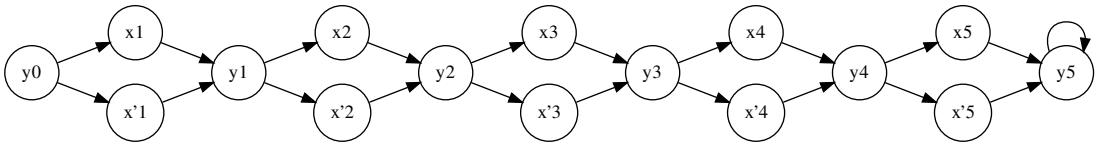


Figure 8.1: Kripke structure for a 3SAT instance with $n = 5$ variables

8.6.2 Summary of other results

Property 8.40 (Sistla and Clarke) The problem of checking whether some path satisfies a formula ψ is PSPACE–complete for

- $\psi \in \Phi(\neg, \wedge, \mathsf{F}, \mathsf{X})$
- $\psi \in \Phi(\neg, \wedge, \mathsf{U})$
- $\psi \in \Phi(\neg, \wedge, \mathsf{U}, \mathsf{X})$

Property 8.41 (Bauland *et al.*) The problem of checking whether some path satisfies a formula $\psi \in \Phi(\mathsf{U})$ is NP–complete.

8.7 Fairness

We do not need to do anything extra to deal with fairness in LTL, because fairness constraints can be written directly into the LTL path formula. For example, using the fairness constraints we discussed with CTL, we can specify a set of states $\mathcal{C}$ that we want to occur infinitely often by defining an atomic proposition c that is true on the states in $\mathcal{C}$. Then, the formula

$$\mathsf{GF}\, c$$

is satisfied only by paths where c holds infinitely often along the path. Thus, if we have an LTL path formula ψ, and want to know if it holds for some *fair* path, we can simply write

$$\mathsf{E}\,(\mathsf{GF}\, c \wedge \psi)$$

without needing any special quantifiers as we did for CTL. Or, if we want to quantify over all *fair* paths, we can write something like

$$\mathsf{A}\,(\mathsf{GF}\, c \rightarrow \psi)$$

which, again, will be true if there are no fair paths.

In fact, LTL allows us to specify more detailed fairness conditions. For example, suppose we have a Kripke structure that models three cooperating threads, then we can express the fairness constraint that all threads enter their critical sections infinitely often as

$$\mathsf{GF}\, c_1 \wedge \mathsf{GF}\, c_2 \wedge \mathsf{GF}\, c_3$$

where atomic proposition c_i holds whenever thread i is in its critical section. Or, we can write

$$\mathsf{GF}\, c_1 \rightarrow (\mathsf{GF}\, c_2 \wedge \mathsf{GF}\, c_3)$$

which says, if thread 1 enters its critical section infinitely often, then so do threads 2 and 3.

The above are examples of *strong fairness*. There is also a notion of *weak fairness*, which instead says

$$\mathsf{FG}\, p_1 \rightarrow \mathsf{GF}\, p_2$$

or, if p_1 eventually holds forever, then p_2 occurs infinitely often.

Property 8.42

$$\mathsf{FG}\, p \quad \rightarrow \quad \mathsf{GF}\, p$$

Proof:

$$\begin{array}{lll} \pi \models \mathsf{FG}\, p & \rightarrow & \exists i \geq 0, \forall j \geq i, \pi^j \models p \\ & \rightarrow & \forall j \geq 0, \exists k \geq j, \pi^k \models p \quad (\text{if } j < i, \text{ use } k = i, \text{ else use } k = j) \\ & \rightarrow & \pi \models \mathsf{GF}\, p \end{array}$$

Property 8.43 Strong fairness implies weak fairness.

Proof: Let $A = \mathsf{FG}\, p_1$, $B = \mathsf{GF}\, p_1$, and $C = \mathsf{GF}\, p_2$. Property 8.42 says $A \rightarrow B$, and is known to be true. Strong fairness says $B \rightarrow C$. Thus, we have

$$\mathsf{GF}\, p_1 \rightarrow \mathsf{GF}\, p_2 \quad \Leftrightarrow \quad B \rightarrow C \quad \Leftrightarrow \quad (A \rightarrow B)(B \rightarrow C) \quad \Rightarrow \quad (A \rightarrow C) \quad \Leftrightarrow \quad \mathsf{FG}\, p_1 \rightarrow \mathsf{GF}\, p_2$$