What is Model Checking?

Model Checking is a verification technique which automatically checks whether a *model* satisfies a given property. This is done by enumerating (either explicitly or symbolically) a set of *states* of the model, and checking that each state satisfies the property.

In practice, model checking requires three steps.

- Modeling: A formal model must be created to represent the actual system.
- Specification: The properties that the system should satisfy must be stated formally.
- Verification: The formal model is checked to see if it satisfies the specified properties. If a failure is detected, a *counter-example* is produced.

Modeling

A common formal model for systems is a *Kripke structure*.

Let a^* be a set of *atomic propositions*. In this context, an atomic proposition is anything that describes a property which may be true about the system being modeled (depending on what state the system is in). For our purposes, we will consider a^* to be a set of propositional symbols.

A Kripke structure M over a^* is a four-tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total (that is, for every state $s \in S$, there is a state $s' \in S$ such that $R(s, s')$).
4. $L : S \to \mathcal{P}(a^*)$ is a *labeling function* that labels each state with the set of atomic propositions true in that state.

A *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

State Graphs and Computation Trees

A *state transition graph* for a structure $M = (S, S_0, R, L)$ has a vertex for each state in S. If $s, t \in S$, then there is a directed edge from the vertex for s to the vertex for t iff $R(s, t)$.

The *image* $\mathit{Image}(X)$ of a set $X \subseteq S$ is the set $\{y \mid \exists\, x \in X.\ R(x, y)\}$. For a single state x, $\mathit{Image}(x)$ denotes $\{y \mid R(x, y)\}$.

A *computation tree* from a state s is an infinite tree in which each vertex is labeled by a state of M. The tree is built as follows.

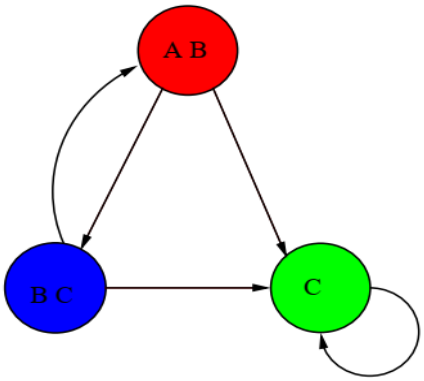
- The root of the tree is labeled by the state s.
- For each vertex v in the tree, if v is labeled by t, then there for each $t' \in \mathit{Image}(t)$, there is a child of t labeled by t'.

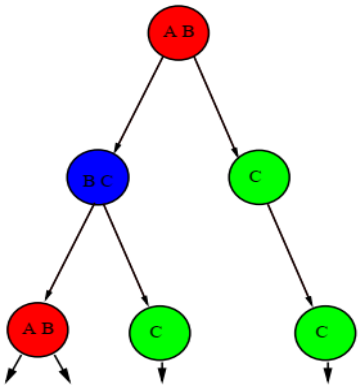
Example

Consider a Kripke structure $M = (S, S_0, R, L)$ over a^* where

- $a^* = \{A, B, C\}$
- $S = \{r, g, b\}$
- $S_0 = \{r\}$
- $R = \{(r, b), (r, g), (g, g), (b, r), (b, g)\}$
- $L(r) = \{A, B\}, L(g) = \{C\}, L(b) = \{B, C\}$

The state transition graph and computation tree from r are shown below.





Specifying Properties

Typically, properties of the model are specified using the logic *CTL** . *CTL* stands for *Computation Tree Logic* since its semantics are best understood in terms of computation trees.

There are two types of formulas in *CTL** : *state formulas* and *path formulas*. Let a^* be a set of atomic propositions. The syntax of *CTL** formulas is given by the following rules:

- If $p \in a^*$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are state formulas.
- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, **X** f, **F** f, **G** f, f **U** g, and f **R** g are path formulas.
- If f is a path formula, then **E** f and **A** f are state formulas.

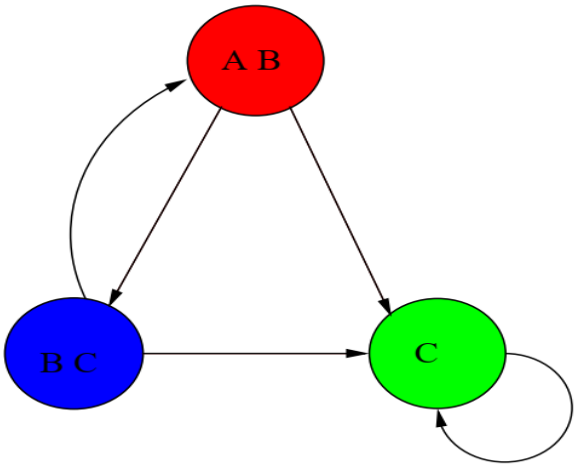
Notice that *CTL** includes propositional logic, but there are also seven new operators: **X** , **F** , **G** , **U** , **R** , **A** , and **E** .

Semantics of *CTL**

State formulas describe properties associated with a single state. For example, any propositional formula over the propositional symbols in a^* is a state formula.

We write $M, s \models f$ to mean that a state formula f is true in state s of the Kripke structure M.

For the initial state of our example, $A \wedge B$ and $\neg A \rightarrow C$ are true state formulas, but $A \rightarrow C$ is not.



Path formulas describe properties associated with a *path*. Recall that a path is a sequence of states $\pi = s_0 s_1 s_2$ such that $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

We write $M, \pi \models g$ to mean that a path formula g is true for path π of the Kripke structure M.

Any state formula is also a path formula and is interpreted as being true if and only if it is true in the first state of the path.

The operators **X**, **F**, **G**, **U**, and **R** are called *temporal operators*. They can be used to create path formulas from state formulas.

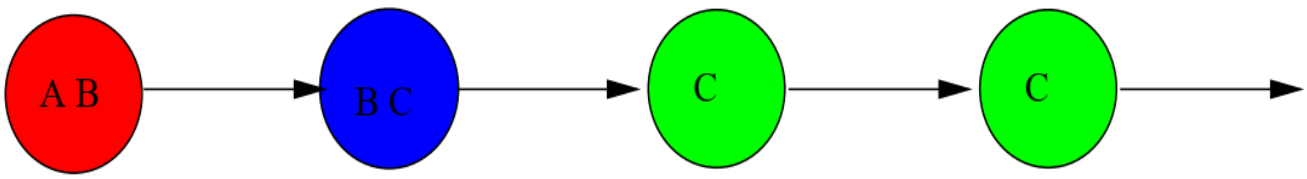
X operator

The **X** ("next time") operator specifies that a property holds in the second state of the path.

By repeatedly appplying this operator, we can specify that a property holds in the n^{th} state of the path.

Which of the following formulas are true for the path below? Note that a state formula is true for a path if it is true in the first state of the path.

- $A \wedge B$
- $\mathbf{X}(A \wedge B)$
- $\mathbf{X}(C)$
- $\mathbf{XX}(C)$

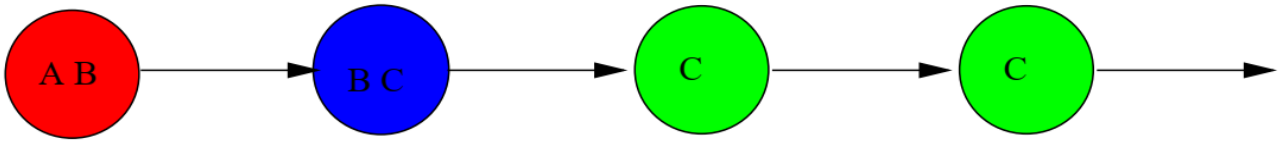


The **X** ("next time") operator specifies that a property holds in the second state of the path.

By repeatedly appplying this operator, we can specify that a property holds in the n^{th} state of the path.

Which of the following formulas are true for the path below? Note that a state formula is true for a path if it is true in the first state of the path.

- $A \wedge B$ *true*
- $\mathbf{X}(A \wedge B)$ *false*
- $\mathbf{X}(C)$ *true*
- $\mathbf{X}\mathbf{X}(C)$ *true*



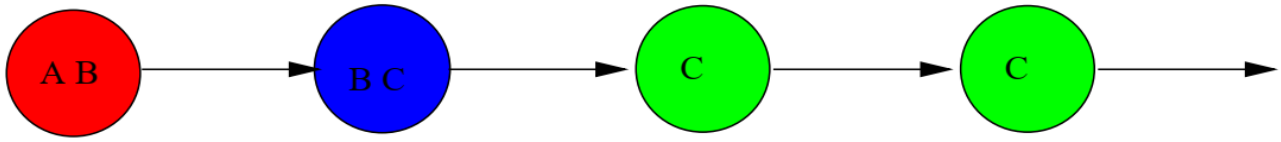
U operator

U ("until") is a binary operator which asserts that the first property holds for every state on a path up to but not necessarily including a state in which the second property holds.

Furthermore, there must exist a state on the path for which the second property holds.

Which of the following formulas are true for the path below?

- $A \mathbf{U} \neg B$ *false*
- $B \mathbf{U} C$ *true*
- $\mathbf{X}(\neg A \mathbf{U} \neg B)$ *true*
- $\mathbf{X}(C \mathbf{U} A)$ *false*



Other Temporal Operators

The other temporal operators can be defined in terms of the others:

- **F** $f = \mathit{true}$ **U** f ("eventually" or "in the future") asserts that f holds at some state on the path.
- **G** $f = \neg$**F** $\neg f$ ("always" or "globally") specifies that f holds at every state on the path.
- f **R** $g = \neg(\neg f$ **U** $\neg g)$ ("release") requires that g holds up to and including the first state where the f holds. Unlike **U**, the "release" property is true even if such a state does not exist.

Path Quantifiers

The *path quantifiers* **A** ("for all paths") and **E** ("there exists a path") are used to convert path formulas to state formulas.

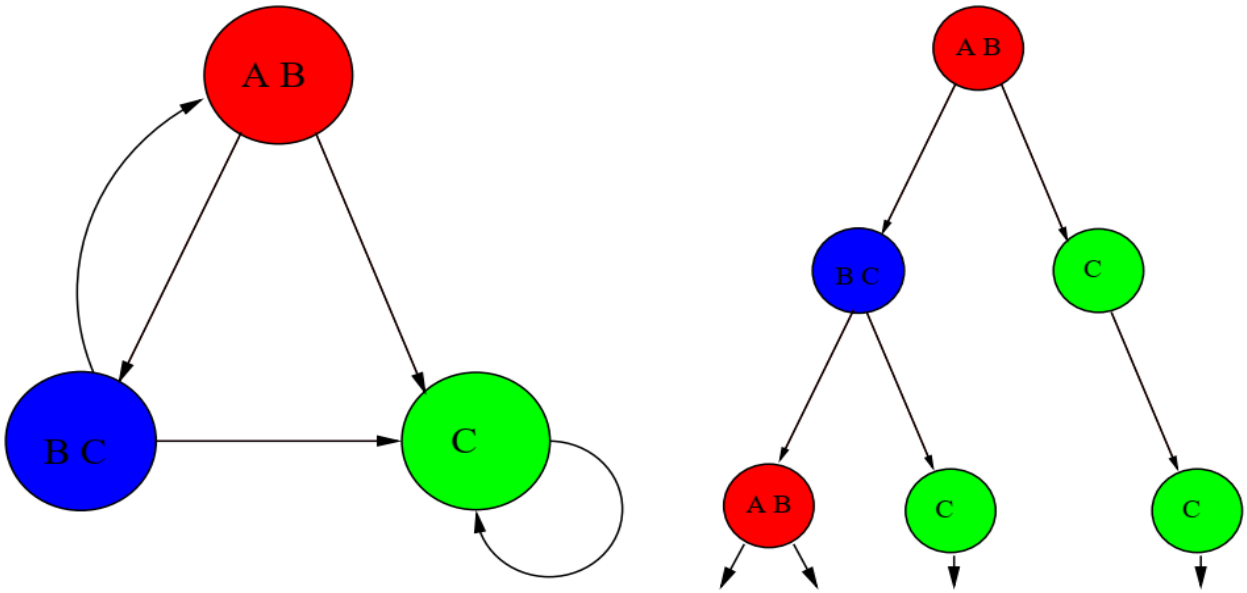
To interpret these formulas relative to a given state s, we consider the computation tree rooted at s.

- **A** (f) specifies that the path formula f is true for *every* path through the tree starting at s.
- **E** (f) specifies that the path formula f is true for *some* path through the tree starting s.

Path Quantifiers Example

Which of the following formulas are true for the initial state of our example?

- $\mathbf{EG}(C)$ *false*
- $\mathbf{AF}(C)$ *true*
- $\mathbf{AG}(C \vee \mathbf{X}(C))$ *true*
- $\mathbf{EX}(\mathbf{AG}(C))$ *true*



CTL and *LTL*

There are two well-known sublogics of CTL^* : *CTL* and *LTL* . They differ only in the allowed syntax. The differences are summarized below.

Syntax of CTL^* :

- State formula α: $p \in a^* \mid \neg\alpha \mid \alpha \vee \alpha \mid \alpha \wedge \alpha \mid \mathbf{E}(\beta) \mid \mathbf{A}(\beta)$
- Path formula β: $\alpha \mid \neg\beta \mid \beta \vee \beta \mid \beta \wedge \beta \mid \mathbf{X}(\beta) \mid \mathbf{F}(\beta) \mid \mathbf{G}(\beta) \mid \beta\,\mathbf{U}\,\beta \mid \beta\,\mathbf{R}\,\beta$

Syntax of *CTL* :

- State formula α: $p \in a^* \mid \neg\alpha \mid \alpha \vee \alpha \mid \alpha \wedge \alpha \mid \mathbf{E}(\beta) \mid \mathbf{A}(\beta)$
- Path formula β: $\mathbf{X}(\alpha) \mid \mathbf{F}(\alpha) \mid \mathbf{G}(\alpha) \mid \alpha\,\mathbf{U}\,\alpha \mid \alpha\,\mathbf{R}\,\alpha$

Syntax of *LTL* :

- State formula α: $\mathbf{A}(\beta)$
- Path formula β: $p \in a^* \mid \neg\beta \mid \beta \vee \beta \mid \beta \wedge \beta \mid \mathbf{X}(\beta) \mid \mathbf{F}(\beta) \mid \mathbf{G}(\beta) \mid \beta\,\mathbf{U}\,\beta \mid \beta\,\mathbf{R}\,\beta$

The three logics have different expressive powers.

- There is no *CTL* formula that is equivalent to the *LTL* formula $\mathbf{A}(\mathbf{F}\,\mathbf{G}\,p)$.
- There is no *LTL* formula that is equivalent to the *CTL* formula $\mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,p)$.
- The disjunction of these two formulas $\mathbf{A}(\mathbf{F}\,\mathbf{G}\,p) \vee \mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,p)$ is a CTL^* formula that cannot be expressed in either *CTL* or *LTL*.

CTL is a common choice for specifying model checking properties.

Some methods use a more restricted logic which allows only universal path quantifiers. The restriction of CTL^* to universal quantifiers is called $ACTL^*$, and the restriction of *CTL* to universal path quantifiers is called *ACTL*.

Typical *CTL* Formulas

Here are some examples of the kinds of formulas that might arrise in specifying properties of an actual system.

- $\mathbf{E}\,\mathbf{F}\,(\mathit{Start} \wedge \neg \mathit{Ready})$: It is possible to get to a state where *Start* holds but *Ready* does not hold.
- $\mathbf{A}\,\mathbf{G}\,(\mathit{Req} \rightarrow \mathbf{A}\,\mathbf{F}\,\mathit{Ack})$: If a request occurs, then it will eventually be acknowledged.
- $\mathbf{A}\,\mathbf{G}\,(\mathbf{A}\,\mathbf{F}\,\mathit{DeviceEnabled})$: The device is enabled (*DeviceEnabled* is true) infinitely often on every computation path.
- $\mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,\mathit{Restart})$: From any state it is possible to get to the *Restart* state.

Fairness

Often, we are only interested in the correctness along *fair* computation paths.

Although it is possible to describe this using *CTL** , we can also augment the Kripke structure with *fairness constraints* and continue using *CTL* for specification.

A *fair Kripke structure* $M = (S, S_0, R, L, F)$ includes a set F of fairness constraints such that $F \subseteq \mathcal{P}(S)$ (i.e. F is a set of subsets of S).

A path π is *fair* if for every $P \in F$, some element of P occurs infinitely often on π.

The semantics with respect to a fair Kripke structure impose the following additional constraints:

- A propositional state formula is true in state s only if there is a fair path in the computation tree starting with s.
- The formula **E** (f) is true in state s if and only if there exists a fair path starting from s that satisfies f.
- The formula **A** (f) is true in state s if and only if all fair paths from s satisfy f.

Model Checking

The basic model checking problem is the following.

Given a Kripke structure M and a formula f expressing some desired property of M, find the set of states $\{s \in S | M, s \models f\}$.

The system satisfies its specification if this set includes the set of initial states S_0.

The first algorithms for model checking used an *explicit* representation of the state transition graph for the Kripke structure.

Explicit State *CTL* Model Checking

First, the formula f is expressed using only the operators $\neg$, $\vee$, **X** , **U** , **G** , and **E** .

We inductively define a procedure *Check* (f) which labels each state s in the state transition graph with the set *label* (s) of subformulas of f which are true in that state.

For atomic propositions p, *Check* (p) just labels each state s such that $p \in L(s)$.

For nontrivial formulas, there are five possible operators to consider: $\neg$, $\vee$, **EX**, **E U**, and **EG**.

- *Check* $(\neg g)$ simply calls *Check* (g) and then labels with $\neg g$ every state not labeled with g.
- *Check* $(g_1 \vee g_2)$ calls *Check* (g_1) and *Check* (g_2) and then labels with $g_1 \vee g_2$ every state labeled with either g_1 or g_2.
- *Check* $(\mathbf{EX}\, g)$ calls *Check* (g) and then labels with $\mathbf{EX}\, g$ every state that has some successor labeled by g.
- *Check* $(\mathbf{E}(g_1\, \mathbf{U}\, g_2)) = \mathit{CheckEU}(g_1, g_2)$
- *Check* $(\mathbf{EG}(g)) = \mathit{CheckEG}(g)$

Explicit State Model Checking: E U

```
procedure CheckEU(f1, f2)
  Check(f1); Check(f2);
  T := {s | f2 ∈ label(s)};
  for each s ∈ T do label(s) := label(s) ∪ {E(f1 U f2)};
  while T ≠ ∅ do
    choose s ∈ T; T := T − {s};
    for each t such that R(t, s) do
      if E(f1 U f2) ∉ label(t) and f1 ∈ label(t) then
        label(t) := label(t) ∪ {E(f1 U f2)};
        T := T ∪ {t};
      end if
    end for
  end while
```

Explicit State Model Checking: EG

```
procedure CheckEG(f1)
  Check(f1);
  S' := {s | f1 ∈ label(s)};
  SCC := {C | C is a nontrivial SCC of S'};
  T := ⋃_{C∈ SCC} {s | s ∈ C};
  for each s ∈ T do label(s) := label(s) ∪ {EG(f1)};
  while T ≠ ∅ do
    choose s ∈ T; T := T − {s};
    for each t such that t ∈ S' and R(t, s) do
      if EG(f1) ∉ label(t) then
        label(t) := label(t) ∪ {EG(f1)};
        T := T ∪ {t};
      end if
    end for
  end while
```

Boolean Functions

Recall our definition of Boolean functions.

For $k \geq 0$, a *k-place Boolean function* is a function from $\{\mathbf{F}, \mathbf{T}\}^k$ to $\{\mathbf{F}, \mathbf{T}\}$. A *Boolean function* is anything which is a k-place Boolean function for some k.

Boolean functions can be represented by propositional formulas. However, as we saw earlier, the representation is not always efficient.

Binary Decision Diagrams are an efficient data structure for representing and performing operations on Boolean functions.

Boolean Function Notation

Assume all functions are n-place Boolean functions on variables $x_1, \ldots, x_n$.

Identity: x_i

Negation: $\overline{f}$

Conjunction: $f \cdot g$

Disjunction: $f + g$

Definitions

Let f be an n-place Boolean function.

A *restriction* or *cofactor* of f is formed by replacing one of its arguments by a constant:

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

The *Shannon expansion* of a function around variable x_i is given by

$$f = x_i \cdot f|_{x_i=1} + \overline{x}_i \cdot f|_{x_i=0}.$$

The function resulting when some argument x_i of function f is replaced by function g is called a *composition* of f and g, and is denoted $f|_{x_i=g}$:

$$f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n).$$

Some functions may not depend on all their arguments. The *dependency set* of a function f, denoted I_f, contains those arguments on which the function depends:

$$I_f = \{i \mid f|_{x_i=0} \neq f|_{x_i=1}\}.$$

Binary Decision Trees

An *binary decision tree* is a rooted, directed tree with two types of vertices: *terminal vertices* and *nonterminal vertices*.

Each nonterminal vertex v is labeled by a variable $\mathit{var}(v)$ and has two successors:

- $\mathit{low}(v)$ corresponding to the case where $\mathit{var}(v)$ is assigned 0, and
- $\mathit{high}(v)$ corresponding to the case where $\mathit{var}(v)$ is assigned 1.

Terminal vertices v have no children and are labeled by $\mathit{value}(v) \in \{0, 1\}$.

A binary decision tree T defines a Boolean function f_v for each vertex v in the tree, defined as follows

- If v is a terminal vertex, then
  - If $\mathit{value}(v) = 1$, then $f_v = 1$.
  - If $\mathit{value}(v) = 0$, then $f_v = 0$.
- If v is a nonterminal vertex and $\mathit{var}(v) = x_i$, then

$$f_v(x_1, \ldots, x_n) = \overline{x_i} \cdot f_{\mathit{low}(v)}(x_1, \ldots, x_n) + x_i \cdot f_{\mathit{high}(v)}(x_1, \ldots, x_n).$$

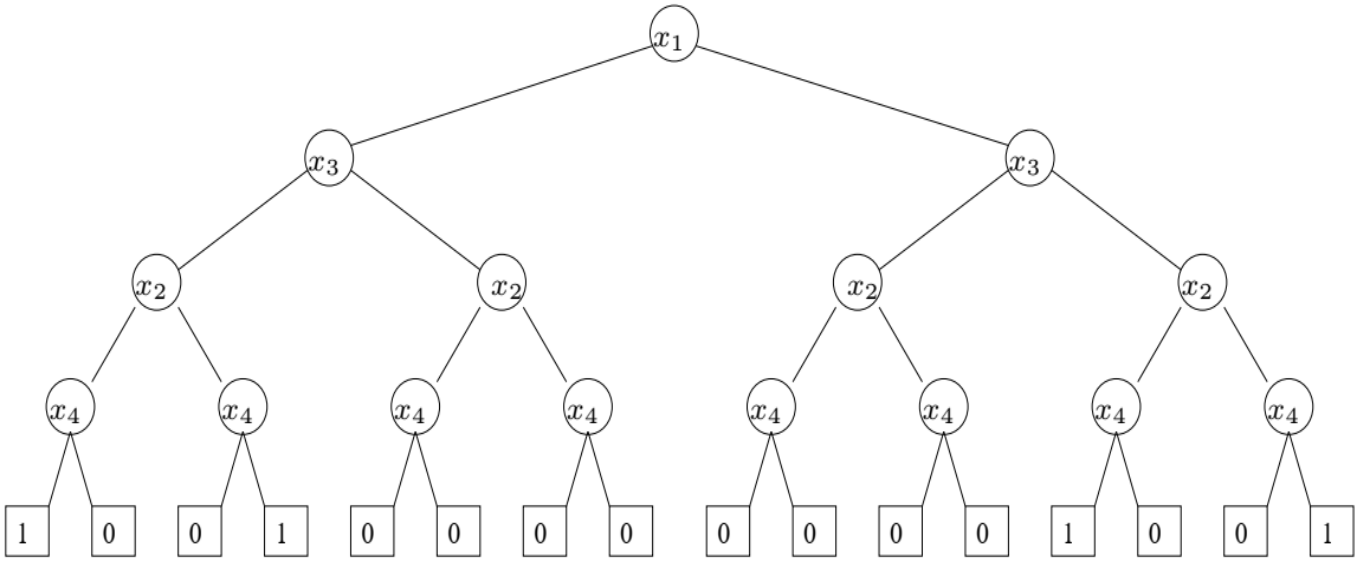
The Boolean function defined by T is the function $f_{\mathit{root}(T)}$ where $\mathit{root}(T)$ denotes the root vertex of T.

Example

A binary decision tree for the two-bit comparator, given by the formula

$$f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \wedge (x_2 \leftrightarrow x_4),$$

is shown below (left is *low*, right is *high*).



Truth Assignments and Binary Decision Trees

To find the value of the function associated with the tree for a given truth assignment, simply traverse the tree from the root as follows.

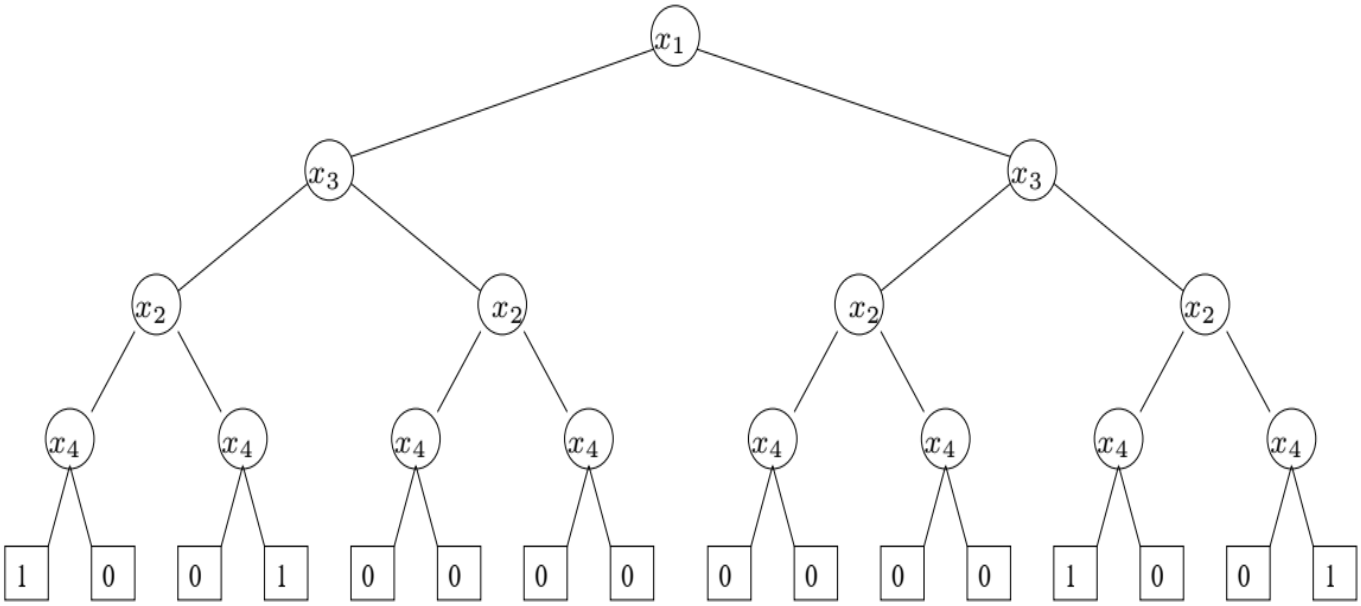
- if *var*(v) is assigned 0, move to *low*(v).
- if *var*(v) is assigned 1, move to *high*(v).

The value that labels the terminal vertex is the value of the function for this assignment.

Truth Assignments and Binary Decision Trees

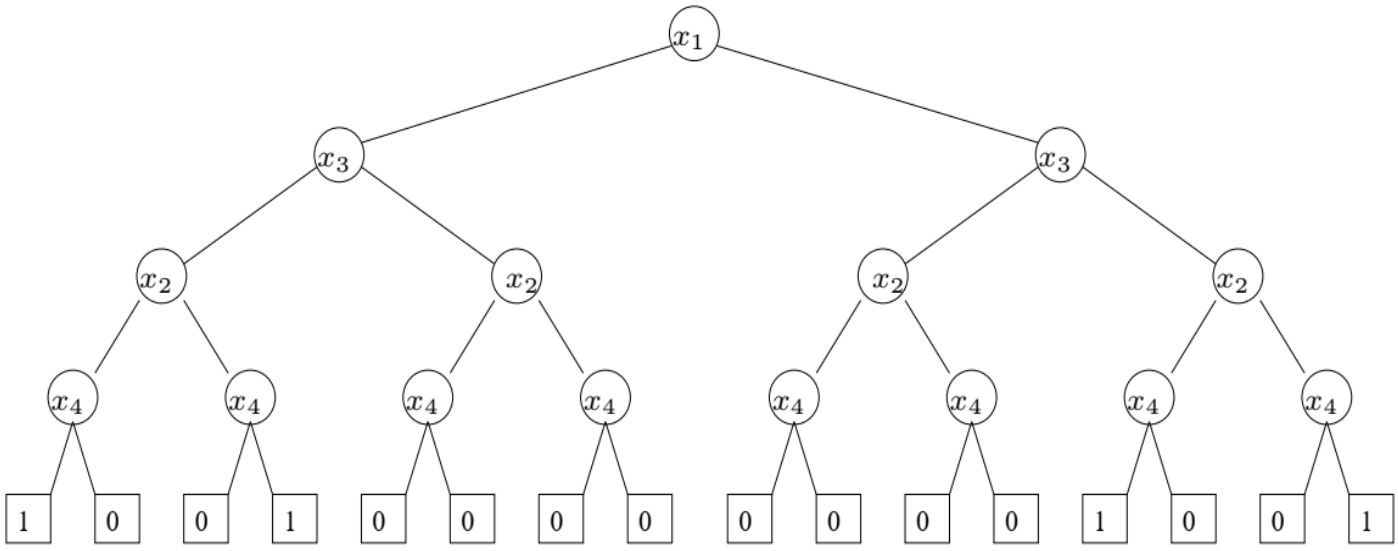
What is $f(1, 0, 1, 0)$, where

$f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \wedge (x_2 \leftrightarrow x_4)$?



What is $f(1,0,1,0)$, where

$f(x_1,x_2,x_3,x_4) = (x_1 \leftrightarrow x_3) \wedge (x_2 \leftrightarrow x_4)$?



The path leads to a terminal vertex labeled with 1, so $f(1,0,1,0) = 1$.

A More Concise Representation

Binary decision trees do not provide a very concise representation for Boolean functions.

There is typically a lot of redundancy in such trees.

In the previous example, there are eight subtrees with roots labeled by x_4, but only three are distinct.

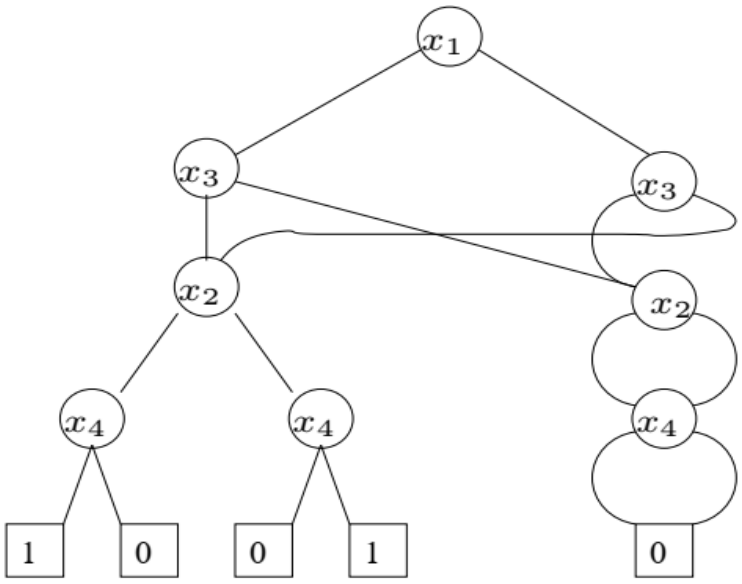
This observation leads to a natural improvement: merge isomorphic subtrees.

The result is a directed acyclic graph (DAG), called a *binary decision diagram* (BDD).

Note that the function represented is unchanged.

Example

After merging isomorphic subtrees, the example looks like this.

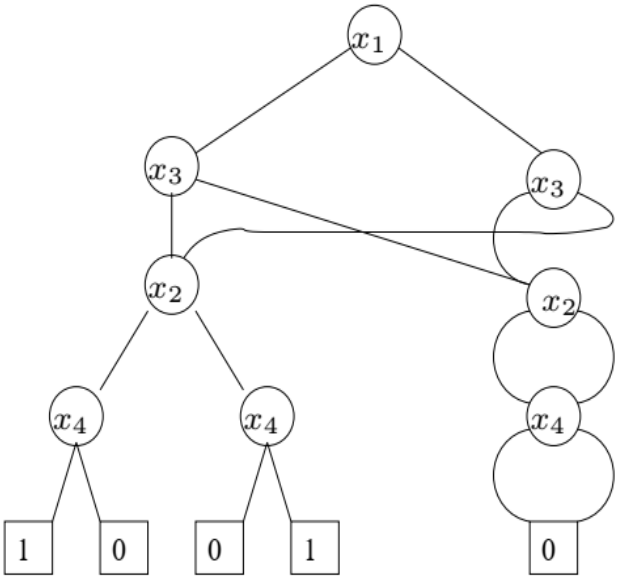


An *ordered binary decision diagram* (OBDD) has the additional property that for some ordering $\prec$ of the variables $x_1, \ldots, x_n$, $\mathit{var}(v) \prec \mathit{var}(\mathit{low}(v))$ and $\mathit{var}(v) \prec \mathit{var}(\mathit{high}(v))$ for each vertex v.

In his original paper, Bryant called these *function graphs*.

Our comparator example is an OBDD which uses the variable ordering:

$x_1 \prec x_3 \prec x_2 \prec x_4$.



Reduced Binary Decision Diagrams

The representation can be made even more concise by eliminating vertices v for which $\mathit{low}(v) = \mathit{high}(v)$. A BDD which contains no such vertices is called *reduced*.

Reduced Ordered Binary Decision Diagrams (ROBDD's) have become the data structure of choice for representing Boolean functions, and are now the most common type of BDD.

The primary advantage of ROBDD's is that they are *canonical*.

Theorem

For any n-place Boolean function f, there is a unique ROBDD (on n variables) denoting f and any other OBDD denoting f contains more vertices.

Proof

By induction on the size of I_f.

Canonicity of ROBDD's

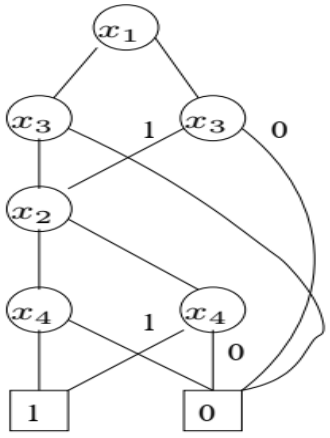
Given any OBDD, an equivalent ROBDD can be computed in linear time by applying a procedure called *Reduce*.

The fact that ROBDD's are canonical make several important Boolean function operations trivial:

- Two Boolean functions are equivalent if they have isomorphic ROBDD's.
- Satisfiability can be determined by simply checking if the ROBDD has a terminal labeled with 1.
- A tautology is represented by the ROBDD with a single vertex labeled 1.

Example

The ROBDD for the comparator example is:



From now on, when we refer to BDD's, we mean ROBDD's.

Note that the size of a BDD depends very much on the variable ordering.

Variable Ordering

In general, finding an optimal ordering is known to be $\mathcal{NP}$-complete.

There are Boolean functions that have exponential size BDD's for any variable ordering (multiplier).

However, heuristics have been developed for finding a good variable ordering when such an ordering exists.

Heuristics try to group *related* variables together.

For example, when converting a circuit to a BDD, the variables in a sub-circuit are related because together they determine the output of that sub-circuit.

Thus, these variables should usually be grouped together.

This can be done by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit.

Dynamic Variable Ordering

A technique called *dynamic reordering* can be useful if no obvious ordering heuristic applies.

When this technique is used, the BDD package internally tries a variety of reorderings and keeps the best one.

Uses various techniques to try to find minimum BDD sizes without getting stuck in a local minimum.

Logical Operations on BDD's

We begin with the operation of restricting some argument x_i of the Boolean function f to a constant value b.

Recall the definition of the restriction or cofactor of f:

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

If f is represented by a BDD, the BDD for the restriction $f|_{x_i=b}$ is computed by a depth-first traversal of the BDD.

For any vertex v which has a pointer to a vertex w such that *var* $(w) = x_i$, we replace the pointer by *low* (w) if b is 0 and *high* (w) if b is 1.

After this transformation, *Reduce* is applied to ensure that the result is canonical.

Logical Operations

All 16 binary propositional connectives can be implemented efficiently on Boolean functions that are represented as BDD's.

In fact, the complexity of these operations is linear in the size of the argument BDDs.

The key idea for efficient implementation of these operations is the *Shannon expansion*:

$$f = x_i \cdot f|_{x_i=1} + \overline{x}_i \cdot f|_{x_i=0}.$$

Bryant gives a uniform algorithm called *Apply* for computing all 16 binary operations.

Let $\odot$ be an arbitrary binary operation, and let f and f' be two Boolean functions. To compute $f \odot f'$:

1. If $\mathit{root}(f)$ and $\mathit{root}(f')$ are both terminal vertices, then $f \odot f' = \mathit{value}(\mathit{root}(f)) \odot \mathit{value}(\mathit{root}(f'))$.
2. If $\mathit{var}(\mathit{root}(f)) = \mathit{var}(\mathit{root}(f'))$, then use the Shannon expansion. Let $x = \mathit{var}(\mathit{root}(f)) = \mathit{var}(\mathit{root}(f'))$:

   $$f \odot f' = x \cdot (f|_{x=1} \odot f'|_{x=1}) + \overline{x} \cdot (f|_{x=0} \odot f'|_{x=0}).$$

   Notice that this effectively breaks the problem into two subproblems which are solved recursively.

   The root of the resulting BDD will be a vertex v labeled by x.

   The first part of this expression computes $\mathit{high}(v)$, and the second part of the expression computes $\mathit{low}(v)$.

Computing $f \odot f'$, continued. Let $x = \mathit{var}(\mathit{root}(f))$ and $x' = \mathit{var}(\mathit{root}(f'))$:

3. If $x < x'$, then $f'|_{x=0} = f'|_{x=1} = f'$ since f' does not depend on x. In this case, the Shannon expansion simplifies to:

   $$f \odot f' = x \cdot (f|_{x=1} \odot f') + \overline{x} \cdot (f|_{x=0} \odot f').$$

   The BDD is then computed recursively as in the second case.

4. If $x > x'$, then $f|_{x'=0} = f|_{x'=1} = f$ since f does not depend on x'. In this case, the Shannon expansion simplifies to:

   $$f \odot f' = x' \cdot (f \odot f'|_{x'=1}) + \overline{x'} \cdot (f \odot f'|_{x'=0}).$$

   The BDD is computed recursively as before.

By using dynamic programming, it is possible to make the algorithm polynomial.

- A hash table is used to record all previously computed subproblems.
- Before any recursive call, the table is checked to see if the subproblem has been solved.
- If it has, the result is obtained from the table; otherwise, the recursive call is performed.
- The result must be reduced to ensure that it is in canonical form.

BDD Extensions

A single DAG can be used to represent a collection of Boolean functions:

- The same variable ordering is used for all of the functions.
- All identical subgraphs are merged.
- Two functions are identical iff they have the same root.
- Checking equivalence can be done in constant time.

Another useful extension adds labels to the edges in the DAG to denote Boolean negation. This makes it unnecessary to use different subgraphs for a formula and its negation.

A single DAG can be used to represent a collection of Boolean functions:

- The same variable ordering is used for all of the functions.
- All identical subgraphs are merged.
- Two functions are identical iff they have the same root.
- Checking equivalence can be done in constant time.

Another useful extension adds labels to the edges in the DAG to denote Boolean negation. This makes it unnecessary to use different subgraphs for a formula and its negation.

How does this extension affect canonicity?

BDD's and Finite Automata

BDD's can also be viewed as *deterministic finite automata*.

An n-argument Boolean function can be identified with the set of strings in $\{0,1\}^n$ that evaluate to 1.

This is a finite language. Finite languages are regular. Hence, there is a minimal DFA that accepts the language.

The DFA provides a canonical form for the original Boolean function.

Logical operations on Boolean functions correspond to standard constructions from automata theory.

Representing Finite Relations

BDD's are extremely useful for obtaining concise representations of relations over finite domains.

If R is an n-ary relation over $\{0,1\}$, then R can be represented by the BDD for its *characteristic function*:

$f_R(x_1,\ldots,x_n) = 1$ iff $R(x_1,\ldots,x_n)$.

- J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
- J.R. Burch, E.M. Clark, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, 1994.

Representing Relations

If R is an n-ary relation over the domain D, where D has 2^m elements for some $m > 1$.

To represent R as a BDD, we encode elements of D using a bijection $\phi : \{0,1\}^m \rightarrow D$ that maps each Boolean vector of length m to an element of D.

We construct a new Boolean relation R' of arity $m \times n$ according to the following rule:

$$R'(\vec{x}_1, \ldots, \vec{x}_n) = R(\phi(\vec{x}_1), \ldots, \phi(\vec{x}_n)),$$

where $\vec{x}_i$ is a vector of m Boolean variables which encodes the variable x_i that takes values in D.

R can now be represented as the BDD for the characteristic function $f_{R'}$ of R'.

A common application of this technique is to use a BDD to represent a set of elements of D (since sets can be viewed as unary relations).

Symbolic Model Checking

We can represent a Kripke structure $M = (S, S_0, R, L)$ using BDD's.

Suppose for simplicity that $|S| = 2^m$. Let ϕ be a 1-1 mapping from $\{0,1\}^m$ to S. We can construct the Boolean function f_{S_0} over the variables $\vec{x}$ such that $f_{S_0}(x_1, \ldots, x_m) = 1$ iff $\phi(x_1, \ldots, x_m) \in S_0$.

To represent R, we use the additional *next-state variables* $\vec{y}$. The BDD for R corresponds to the function f_R:

$f_R(x_1, \ldots, x_m, y_1, \ldots, y_m) = 1$ iff $(\phi(x_1, \ldots, x_m), \phi(y_1, \ldots, y_m)) \in R$.

To represent L, we create a BDD L_p for each atomic proposition p which represents the set of all states $s \in S$ such that $p \in L(s)$.

In explicit state model checking, we labeled each state of a Kripke structure with the *CTL* formulas true in that state.

In *symbolic model checking*, BDD's are used to represent the Kripke structure as well as the sets of states for which a given *CTL* formula holds.

As before, we inductively define a function *Check* . However, instead of labeling states, the function *Check* (f) returns a BDD which represents the set of states for which f holds.

For the base case, *Check* $(p) = L_p$ for $p \in a^*$.

For the inductive case, we again consider the operators $\neg$, $\vee$, **EX**, **E U**, and **EG**.

- *Check* $(\neg g) = \overline{\textit{Check}\,(g)}$
- *Check* $(g_1 \vee g_2) =$ *Check* $(g_1) +$ *Check* (g_2)
- *Check* $(\mathbf{EX}\, g) =$ *CheckEX* $($*Check* $(g))$
- *Check* $(\mathbf{EG}\,(g)) =$ *CheckEG* $($*Check* $(g))$
- *Check* $(\mathbf{E}\,(g_1\, \mathbf{U}\, g_2)) =$ *CheckEU* $($*Check* $(g_1),$ *Check* $(g_2))$

Predicate Transformers and Fixpoints

CheckEX (g) is implemented by computing the pre-image of g:

CheckEX $(g) = \exists\, \vec{y}.\, [(\exists\, \vec{x}.\, ((\vec{x} = \vec{y}) \cdot g)) \cdot f_R))]$

The functions *CheckEG* and *CheckEU* depend on *fixpoint* calculations.

A set $S' \subseteq S$ is a fixpoint of a funciton $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ if $\tau(S') = S'$. The function τ is called a *predicate transformer*.

A predicate transformer τ is *monotonic* if $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$.

Let $\tau^0(Z) = Z$ and let $\tau^{i+1}(Z) = \tau(\tau^i(Z))$.

If τ is monotonic and S is finite, then for every $Z \subseteq S$, there exists an integer k_Z such that $\tau^{k+1}(Z) = \tau^k(Z)$.

The *least fixpoint* $\mu.\,\tau$ of τ is defined to be $\tau^{k_\emptyset}(\emptyset)$.

The *greatest fixpoint* $\nu.\,\tau$ of τ is defined to be $\tau^{k_S}(S)$.

Notice that

$\emptyset \subseteq \tau(\emptyset) \subseteq \cdots \subseteq \mu.\,\tau$ and $S \supseteq \tau(S) \supseteq \cdots \supseteq \nu.\,\tau$.

CheckEG and *CheckEU*

Suppose f and g are subsets of S represented by BDD's.

Let $\tau_f(Z) = f \cdot \mathit{CheckEX}\,(Z)$, and let $\sigma_g(Z) = g + Z$.

Each of these is a monotonic predicate transformer from $\mathcal{P}(S)$ to $\mathcal{P}(S)$.

We can now define *CheckEG* and *CheckEU*:

- $\mathit{CheckEG}\,(g) = \nu.\,\tau_g$
- $\mathit{CheckEU}\,(g_1, g_2) = \mu.\,(\sigma_{g_2} \circ \tau_{g_1})$, where $(f_1 \circ f_2)(x) = f_1(f_2(x))$.

These can be implemented by repeatedly applying the predicate transformer functions until the set of states remains unchanaged. This is easy to detect since equivalence of BDD's can be determined in constant time.

Fairness in Symbolic Model Checking

Consider the formula **EG** (f) given fairness constraints $F = \{P_1, P_2, \ldots, P_n\}$.

The formula is true for a state s iff there exists a path beginning with s on which f is always true and in which at least one state from each set of formulas in F appears infinitely often.

The set of states for which this formula holds is the largest set Z with the following two properties:

- all of the states in Z satisfy f
- for each $P_k \in F$ and all states $s \in Z$, there is a path from s to a state $t \in Z \cap P_k$ such that all states on the path satisfy f.

Thus, we can compute $\mathbf{EG}(f)$ as follows:

- $\tau_f = f \cdot \bigwedge_{k=1}^{n} \mathit{CheckEX}(\mathit{CheckEU}(f, Z \cdot P_k))$
- $\mathit{CheckFairEG}(f) = \nu.\, \tau_f$

CheckFair

Given the function *CheckFairEG*, we can now define the more general function *CheckFair* as follows:

- $\mathit{CheckFair}(p) = L_p \cdot \mathit{CheckFairEG}(S)$ for $p \in a^*$
- $\mathit{CheckFair}(\neg g) = \overline{\mathit{CheckFair}(g)}$
- $\mathit{CheckFair}(g_1 \vee g_2) = \mathit{CheckFair}(g_1) + \mathit{CheckFair}(g_2)$
- $\mathit{CheckFair}(\mathbf{EX}\, g) = \mathit{CheckEX}(\mathit{Check}(g) \cdot \mathit{CheckFairEG}(S))$
- $\mathit{CheckFair}(\mathbf{EG}(g)) = \mathit{CheckFairEG}(\mathit{Check}(g))$
- $\mathit{CheckFair}(\mathbf{E}(g_1\, \mathbf{U}\, g_2)) = \mathit{CheckEU}(g_1, g_2 \cdot \mathit{CheckFairEG}(S))$

Counterexamples and Witnesses

One of the most important features of *CTL* model-checking algorithms is the ability to find *counterexamples* and *witnesses*.

A counterexample is produced when a formula with a universal path quantifier is false.

A witness is produced when a formula with an existential path quantifier is true.