Outline

- Many-Sorted Logic
- Temporal Logic and Model Checking
  - Modeling
  - Specification
  - $CTL^*$, $CTL$, and $LTL$
  - Explicit State Model Checking
  - Symbolic Model Checking
  - Fairness

Sources:

- Enderton, 4.3.

Many-Sorted Logic

In standard first-order logic, the domain of a model consists of a single set of objects.

However, often it is more intuitive to have several sets, one for each of several different kinds of objects.

Many-sorted logic allows us to do exactly this.

A sort is very much like a type in programming.
Many-Sorted Logic: Syntax

The syntax of many-sorted logic is identical to that of regular first-order logic.
However, additional information is given which can be used to determine whether a formula is well-sorted.

Let $S = \{ s_1, \ldots, s_n \}$ be a set of sorts.
For each variable $v_k$, we associate a sort $sort(v_k) = s_i$.
Suppose $\Sigma$ is a signature.
- Each constant $c \in \Sigma$ has an associated sort $sort(c) = s_i$.
- Each function symbol $f \in \Sigma$ of arity $n$ has a sort which is an $n + 1$-tuple $sort(f) = \langle s_{i_1}, \ldots, s_{i_n}, s_{i_{n+1}} \rangle$.
- Each predicate symbol $p \in \Sigma$ of arity $n$ has a sort which is an $n$-tuple $sort(p) = \langle s_{i_1}, \ldots, s_{i_n} \rangle$.

We define $sort(t)$, a function from terms to sorts as follows:
- If $t$ is a variable or constant term, then $sort(t) = sort(t)$.
- If $t = ft_1 \ldots t_n$, then $sort(t) = s_{i_{n+1}}$, where $sort(f) = \langle s_{i_1}, \ldots, s_{i_n}, s_{i_{n+1}} \rangle$.

Many-Sorted Logic

Example
Let $\Sigma = \{ 1, +, \text{car}, \text{cdr} \}$, $S = \{ \mathcal{N}, \mathcal{L} \}$, and suppose we have variables $v_1$ and $l_i$. Define $sort$ as follows:
- $sort(v_1) = \mathcal{N}$, $sort(l_i) = \mathcal{L}$
- $sort(1) = \mathcal{N}$
- $sort(+) = \langle \mathcal{N}, \mathcal{N}, \mathcal{N} \rangle$
- $sort(\text{car}) = \langle \mathcal{L}, \mathcal{N} \rangle$
- $sort(\text{cdr}) = \langle \mathcal{L}, \mathcal{L} \rangle$

Are the following well-sorted?
- $1 + v_5 \text{ yes}$
- $\text{car}(1 + v_5) \text{ no}$
- $\text{cdr}(l_5) + 1 \text{ no}$
- $\text{car}(l_5) + 1 \text{ yes}$
- $\text{car}(l_3) = 1 + \text{car}(\text{cdr}(l_3)) \text{ yes}$

Many-Sorted Logic: Syntax

Given a signature $\Sigma$, a set of sorts $S$, and a sort function $sort$, we can determine whether a $\Sigma$-formula is well-sorted as follows.

We define a function $well$ from expressions to $\{ \text{true}, \text{false} \}$.
We first define $well$ for terms.
- For every variable or constant term $t$, $well(t) = \text{true}$.
- If $t = ft_1 \ldots t_n$ and $sort(f) = \langle s_{i_1}, \ldots, s_{i_n}, s_{i_{n+1}} \rangle$, then $well(t) = well(t_1) \land \cdots \land well(t_n) \land \overline{sort}(f) = s_{i_1} \land \cdots \land \overline{sort}(t_n) = s_{i_n}$.

For atomic formulas,
- $well(t_1 = t_2) = well(t_1) \land well(t_2) \land \overline{sort}(t_1) = \overline{sort}(t_2)$.
- If $sort(p) = \langle s_{i_1}, \ldots, s_{i_n} \rangle$, then $well(pt_1 \ldots t_n) = well(t_1) \land \cdots \land well(t_n) \land \overline{sort}(t_1) = s_{i_1} \land \cdots \land \overline{sort}(t_n) = s_{i_n}$.

A formula is well-sorted if each of its atomic formulas is well-sorted.

Many-Sorted Logic: Semantics

Given a signature $\Sigma$, a set of sorts $S$, and a sorting function $sort$, a many-sorted model consists of the following:

1. For each $s_i \in S$, a nonempty set called the domain of $s_i$, written $\text{dom}(s_i)$.
2. A mapping from each constant $c$ in $\Sigma$ of sort $sort(c) = s_i$ to an element $c^\mathcal{M}$ of $\text{dom}(s_i)$.
3. A mapping from each $n$-ary function symbol $f$ in $\Sigma$ of sort $sort(f) = \langle s_{i_1}, \ldots, s_{i_n}, s_{i_{n+1}} \rangle$ to $f^\mathcal{M}$, an $n$-ary function from $\text{dom}(s_{i_1}) \times \cdots \times \text{dom}(s_{i_n})$ to $\text{dom}(s_{i_{n+1}})$.
4. A mapping from each $n$-ary predicate symbol $p$ in $\Sigma$ of sort $sort(p) = \langle s_{i_1}, \ldots, s_{i_n} \rangle$ to an $n$-ary relation $p^\mathcal{M} \subseteq \text{dom}(s_{i_1}) \times \cdots \times \text{dom}(s_{i_n})$.

The definitions of truth and satisfaction are analogous to the standard ones. The only one that is slightly different is that $\forall v_i \phi$ is interpreted as taking $v_i$ to range over all the elements of $\text{dom}(sort(v_i))$. 
Many-Sorted Logic

Though convenient, many-sorted logic does not give us any more power than regular first-order logic.

Let $\Sigma$ be a signature, $S$ a set of sorts, and $\text{sort}$ a sorting function.

Let $\Sigma^*$ be a new signature which includes $\Sigma$ as well as a new predicate $Q_{s_i}$ for each $s_i \in S$.

Given a $\Sigma$-formula $\phi$, define $\phi^*$ to be $\phi$ with every instance of $\forall v_i \psi$ replaced with $\forall v_i (Q_{\text{sort}(v_i)} (v_i) \rightarrow \psi)$.

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.

Many-Sorted Logic

Given a many-sorted model $M$, define a standard model $M^*$ as follows.

- $\text{dom}(M^*)$ is the union of $\text{dom}(s_i)$ for $s_i \in S$.
- For constant symbols $c, c^{M^*} = c^M$.
- For function symbols $f, f^{M^*}$ is an arbitrary extension of $f^M$ to a total function on $\text{dom}(M^*)$.
- For predicate symbols $p, p^{M^*} = p^M$.
- For the new predicate symbols $Q_{s_i}, Q_{s_i}^{M^*} = \text{dom}(s_i)$

Theorem

A many-sorted sentence $\sigma$ is true in $M$ iff $\sigma^*$ is true in $M^*$.

Proof

Use induction to show that $\models_M \phi[s]$ iff $\models_{M^*} \phi^*[s]$.

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. $L : S \rightarrow \mathcal{P}(a^*)$ 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 $\pi = s_0 s_1 s_2 \ldots$ 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 $\text{Image}(X)$ of a set $X \subseteq S$ is the set $\{y | \exists x \in X, R(x, y)\}$. For a single state $x$, $\text{Image}(x)$ denotes $\{y | R(x, y)\}$.

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 $t' \in \text{Image}(t)$, there is a child of $t$ labeled by $t'$.

Specifying Properties

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

There are two types of formulas in $\text{CTL}^*$: state formulas and path formulas. Let $a^*$ be a set of atomic propositions. The syntax of $\text{CTL}^*$ 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 $\neg f$, $f \lor g$, and $f \land g$ are state formulas.
- If $f$ is a state formula, then $f$ is also a path formula.
- If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $f \land g$, $\text{X}f$, $\text{F}f$, $\text{G}f$, $f \text{U}g$, and $f \text{R}g$ are path formulas.
- If $f$ is a path formula, then $\text{EF} f$ and $\text{AF} f$ are state formulas.

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

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.
**Semantics of CTL**

*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 \( \pi \) 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 applying this operator, we can specify that a property holds in the \( n \text{th} \) 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.

- \( A \land B \) true
- \( X(A \land B) \) false
- \( X(C) \) true
- \( XX(C) \) 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?

- \( A U \neg B \) false
- \( B U C \) true
- \( X(\neg A U \neg B) \) true
- \( X(C U A) \) false

**Other Temporal Operators**

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

- \( F f = true U f \) ("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.
- \( f R g = \neg (\neg f U g) \) ("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 $\forall$ (“for all paths”) and $\exists$ (“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 $s$.

- $\forall f$ specifies that the path formula $f$ is true for every path through the tree starting at $s$.
- $\exists f$ specifies that the path formula $f$ is true for some path through the tree starting $s$.

**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 $\alpha: p \in a^+ \mid -\alpha \mid \alpha \lor \alpha \mid \alpha \land \alpha \mid E(\beta) \mid A(\beta)$
- Path formula $\beta: \alpha \mid -\beta \mid \beta \lor \beta \mid \beta \land \beta \mid X(\beta) \mid F(\beta) \mid G(\beta) \mid U(\beta) \mid R(\beta)$

Syntax of $CTL$:

- State formula $\alpha: p \in a^+ \mid -\alpha \mid \alpha \lor \alpha \mid \alpha \land \alpha \mid E(\beta) \mid A(\beta)$
- Path formula $\beta: X(\alpha) \mid F(\alpha) \mid G(\alpha) \mid U(\alpha) \mid R(\alpha)$

Syntax of $LTL$:

- State formula $\alpha: A(\beta)$
- Path formula $\beta: p \in a^+ \mid -\beta \mid \beta \lor \beta \mid \beta \land \beta \mid X(\beta) \mid F(\beta) \mid G(\beta) \mid U(\beta) \mid R(\beta)$

**Path Quantifiers Example**

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

- $EG(C)$ false
- $AF(C)$ true
- $AG(C \lor X(C))$ true
- $EX(AG(C))$ true

**CTL and LTL**

The three logics have different expressive powers.

- There is no $CTL$ formula that is equivalent to the $LTL$ formula $A(FGp)$.
- There is no $LTL$ formula that is equivalent to the $CTL$ formula $AG(EFp)$.
- The disjunction of these two formulas $A(FGp) \lor AG(EFp)$ 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 path 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 arise in specifying properties of an actual system.

- **EF**(Start \( \land \neg Ready \))\): It is possible to get to a state where Start holds but Ready does not hold.
- **AG**(Req \( \rightarrow \) AF Ack)\): If a request occurs, then it will eventually be acknowledged.
- **AG**(AF DeviceEnabled)\): The device is enabled (DeviceEnabled is true) infinitely often on every computation path.
- **AG**(EF 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 \( \pi \) is fair if for every \( P \in F \), some element of \( P \) occurs infinitely often on \( \pi \).

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 \( \neg, \lor, \land, \forall, \exists \), and \( E \).

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

For atomic propositions \( p \), \( \text{Check}(p) \) just labels each state \( s \) such that \( p \in L(s) \).

For nontrivial formulas, there are five possible operators to consider: \( \neg, \lor, \exists X, E U, \) and \( E G \).

- \( \text{Check}(\neg g) \) simply calls \( \text{Check}(g) \) and then labels with \( \neg \) every state not labeled with \( g \).
- \( \text{Check}(g_1 \lor g_2) \) calls \( \text{Check}(g_1) \) and \( \text{Check}(g_2) \) and then labels with \( g_1 \lor g_2 \) every state labeled with either \( g_1 \) or \( g_2 \).
- \( \text{Check}(E X g) \) calls \( \text{Check}(g) \) and then labels with \( E X g \) every state that has some successor labeled by \( g \).
- \( \text{Check}(E(g_1 U g_2)) = \text{CheckEU}(g_1, g_2) \)
- \( \text{Check}(E G(g)) = \text{CheckEG}(g) \)
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 $\phi$ 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 $f_R$:

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

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 $p \in L(s)$.

Explicit State Model Checking: $\mathbf{E \ U}$

procedure CheckEU($f_1, f_2$)
    Check($f_1$); Check($f_2$);
    $T := \{s | f_2 \in label(s)\}$;
    for each $s \in T$ do $label(s) := label(s) \cup \{E(f_1 \cup f_2)\}$;
    while $T \neq \emptyset$ do
        choose $s \in T$; $T := T \setminus \{s\}$;
        for each $t$ such that $R(t,s)$ do
            if $E(f_1 \cup f_2) \notin label(t)$ and $f_1 \in label(t)$ then
                $label(t) := label(t) \cup \{E(f_1 \cup f_2)\}$;
                $T := T \cup \{t\}$;
            end if
        end for
    end while

Explicit State Model Checking: $\mathbf{E \ G}$

procedure CheckEG($f_1$)
    Check($f_1$);
    $S' := \{s | f_1 \in label(s)\}$;
    $SCC := \{C | C$ is a nontrivial SCC of $S'\}$;
    $T := \bigcup_{C \in SCC} \{s | s \in C\}$;
    for each $s \in T$ do $label(s) := label(s) \cup \{EG(f_1)\}$;
    while $T \neq \emptyset$ do
        choose $s \in T$; $T := T \setminus \{s\}$;
        for each $t$ such that $t \in S'$ and $R(t,s)$ do
            if $EG(f_1) \notin label(t)$ then
                $label(t) := label(t) \cup \{EG(f_1)\}$;
                $T := T \cup \{t\}$;
            end if
        end for
    end while

Symbolic Model Checking

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 $\neg$, $\lor$, $EX$, $EU$, and $EG$.

- $Check(\neg g) = \overline{Check(g)}$
- $Check(g_1 \lor g_2) = Check(g_1) + Check(g_2)$
- $Check(EG) = CheckEG(Check(g))$
- $Check(EG) = CheckEG(Check(g))$
- $Check(EU) = CheckEU(Check(g_1), Check(g_2))$
Predicate Transformers and Fixpoints

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

\[
\text{CheckEX}(g) = \exists \tilde{y}. [(\exists x. ((\tilde{x} = \tilde{y}) \cdot f_R)]
\]

The functions CheckEG and CheckEU depend on fixpoint calculations.

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

A predicate transformer \(\tau\) 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 \(\tau\) 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 \(\mu\) of \(\tau\) is defined to be \(\tau^k(\emptyset)\).

The greatest fixpoint \(\nu\) of \(\tau\) is defined to be \(\tau^k(S)\).

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

Fairness in Symbolic Model Checking

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

The formula is true for a state \(s\) if 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 \(\text{EG}(f)\) as follows:

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

CheckEG and CheckEU

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

Let \(\tau_f(Z) = f \cdot \text{CheckEX}(Z)\), 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:

- \(\text{CheckEG}(g) = \nu, \tau_g\)
- \(\text{CheckEU}(g_1, g_2) = \mu, (\sigma_{g_2} \circ \tau_{g_1})\), 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 unchanged. This is easy to detect since equivalence of BDD's can be determined in constant time.

CheckFair

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

- \(\text{CheckFair}(p) = L_p \cdot \text{CheckFairEG}(S)\) for \(p \in a^*\)
- \(\text{CheckFair}(-g) = \overline{\text{CheckFair}(g)}\)
- \(\text{CheckFair}(g_1 \lor g_2) = \text{CheckFair}(g_1) + \text{CheckFair}(g_2)\)
- \(\text{CheckFair}(\text{EX}g) = \text{CheckEX}(\text{Check}(g) \cdot \text{CheckFairEG}(S))\)
- \(\text{CheckFair}(\text{EG}(g)) = \text{CheckFairEG}(\text{Check}(g))\)
- \(\text{CheckFair}(\text{E}(g_1 U g_2)) = \text{CheckEU}(g_1, g_2 \cdot \text{CheckFairEG}(S))\)
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.