Model Checking

Clark Barrett

Stanford University

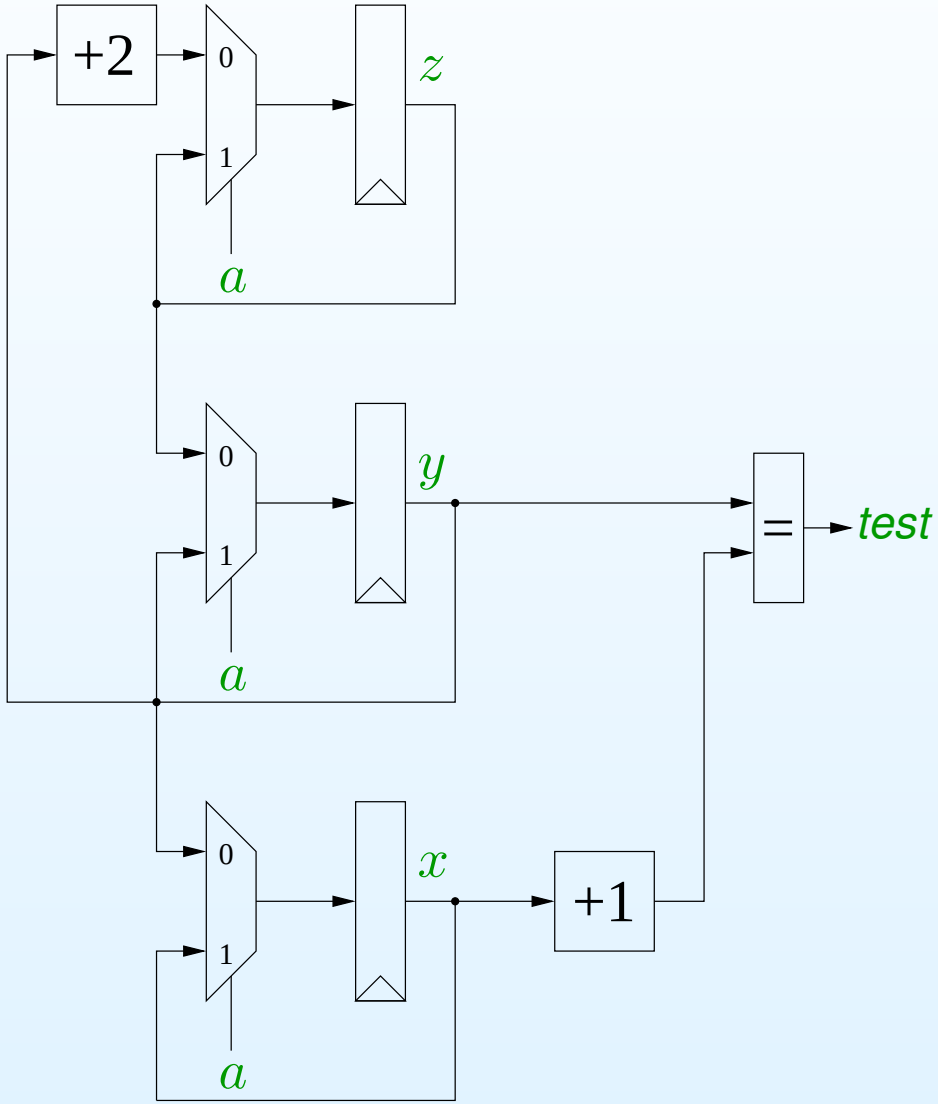
What is Formal Verification?

1. *Modeling*: Create a mathematical model of the system
   - An inaccurate model can introduce false bugs or mask real bugs
   - For many systems, this step can be done automatically
2. *Specification*: The properties which the system should satisfy must be stated in a formal language
   - *Challenge*: translate informal into formal specifications
   - *Challenge*: requires manual effort and expertise
3. *Proof*: Prove that the model satisfies the specification
   - Better than testing: covers *all* cases
   - ...when it succeeds:
   - *Challenge*: Can the proof be automated? Does it scale?

Modeling

Let us consider again the circuit example we saw before.

Circuit Example



Modeling

How do we model this circuit as a mathematical object?

Modeling

One formal model for systems is a *Kripke structure*, which is a specific kind of *transition system*.

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.

Modeling

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.

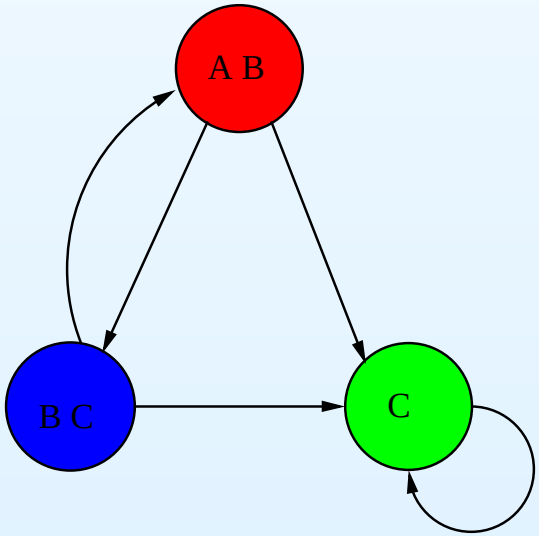
- The root of the tree is labeled by the state s.
- For each vertex v in the tree, if v is labeled by t, then there for each $t' \in \mathit{Image}(t)$, there is a child of t labeled by t'.

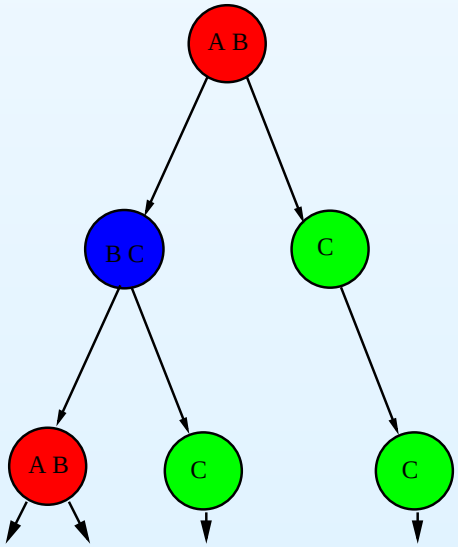
Example

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.





SAT encoding of Kripke structures

A Kripke structure contains a finite set S of states, a set S_0 of initial states, and a transition relation R.

Since S is finite, we can find an m such that $2^m \geq |Q|$. We can then use m variables: $\mathbf{x} = [x_1, \ldots, x_m]$ to represent the states. These are called *state variables*.

To represent R, we need m additional variables, $\mathbf{y} = [y_1, \ldots, y_m]$, which we call *next-state variables*.

We can write formulas $F_{S_0}(\mathbf{x})$ and $F_R(\mathbf{y})$ such that the solutions of $F_{S_0}(\mathbf{x})$ correspond to initial states in S_0 and the solutions of $F_R(\mathbf{x}, \mathbf{y})$ correspond to valid transitions in R.

Proof by Induction

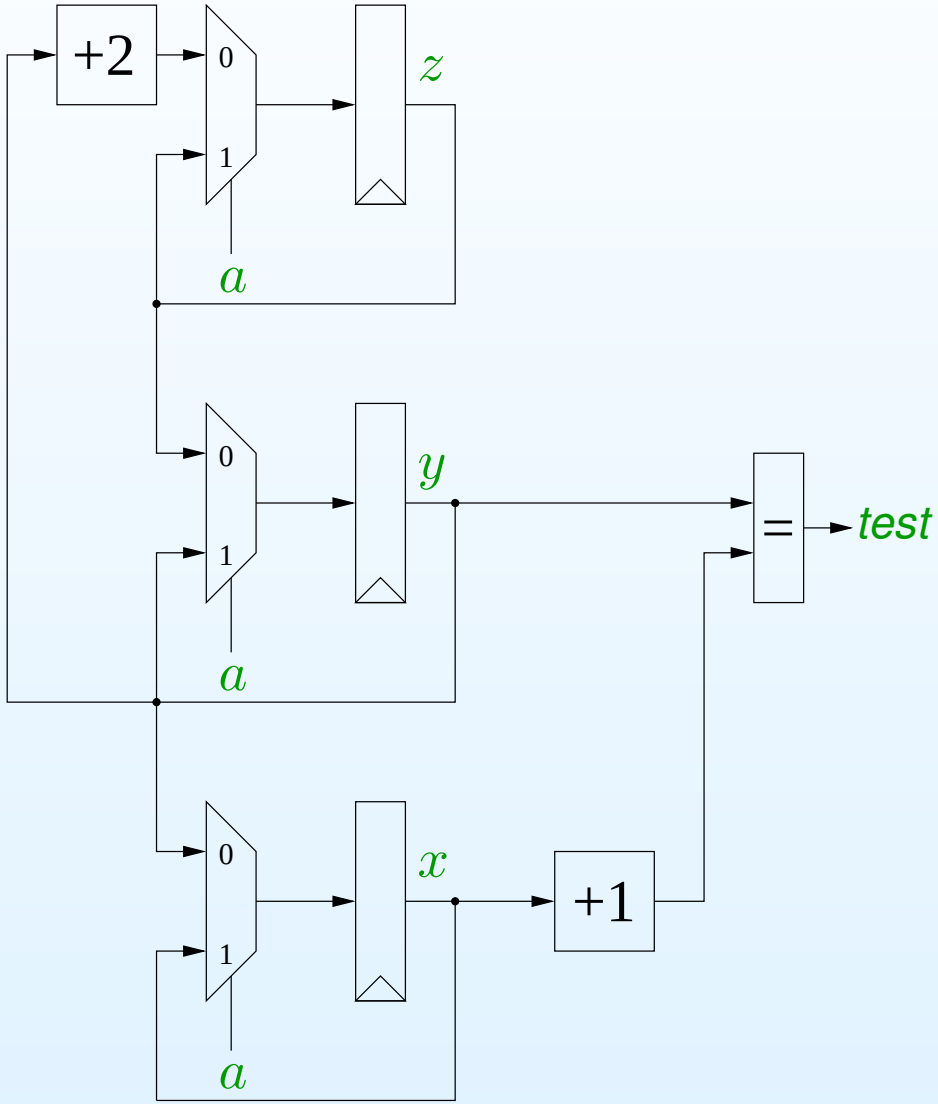
Induction can be used to show that a property P holds for all states:

- *Base case*: $F_{S_0}(\mathbf{x_0}) \rightarrow F_{S_P}(\mathbf{x_0})$
- *Induction case*: $(F_{S_P}(\mathbf{x}) \wedge F_R(\mathbf{x}, \mathbf{y})) \rightarrow F_{S_P}(\mathbf{y})$

k-induction is a more general version that often works better:

- *Base case*: $(F_{S_0}(\mathbf{x_0}) \wedge F_R(\mathbf{x_0}, \mathbf{x_1}) \wedge \cdots \wedge F_R(\mathbf{x_{k-1}}, \mathbf{x_k})) \rightarrow (F_{S_P}(\mathbf{x_0}) \wedge F_{S_P}(\mathbf{x_1}) \wedge \cdots \wedge F_{S_P}(\mathbf{x_k}))$
- *Induction case*: $(F_{S_P}(\mathbf{x_0}) \wedge F_R(\mathbf{x_0}, \mathbf{x_1}) \wedge \cdots \wedge F_{S_P}(\mathbf{x_k}) \wedge F_R(\mathbf{x_k}, \mathbf{x_{k+1}})) \rightarrow F_{S_P}(\mathbf{x_{k+1}})$

Circuit Example



Circuit Example

Assume every register is 1-bit.

What is the property?

Circuit Example

Assume every register is 1-bit.

What is the property?

$F_{S_P}(x, y, z) = (z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x)$

Circuit Example

Assume every register is 1-bit.

What is the property?

$F_{S_P}(x, y, z) = (z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x)$

What is the base case?

Circuit Example

Assume every register is 1-bit.

What is the property?

$F_{S_P}(x, y, z) = (z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x)$

What is the base case?

$F_{S_0}(x, y, z) \rightarrow F_{S_P}(x, y, z) =$
$((x \leftrightarrow x_0) \wedge (y \leftrightarrow y_0) \wedge (z \leftrightarrow z_0)) \rightarrow ((z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x))$

Circuit Example

Assume every register is 1-bit.

What is the property?

$F_{S_P}(x, y, z) = (z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x)$

What is the base case?

$F_{S_0}(x, y, z) \rightarrow F_{S_P}(x, y, z) =$
$((x \leftrightarrow x_0) \wedge (y \leftrightarrow y_0) \wedge (z \leftrightarrow z_0)) \rightarrow ((z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x))$

What is the transition relation?

Circuit Example

Assume every register is 1-bit.

What is the property?

$F_{S_P}(x, y, z) = (z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x)$

What is the base case?

$F_{S_0}(x, y, z) \rightarrow F_{S_P}(x, y, z) =$
$((x \leftrightarrow x_0) \wedge (y \leftrightarrow y_0) \wedge (z \leftrightarrow z_0)) \rightarrow ((z \leftrightarrow \neg y) \wedge (y \leftrightarrow \neg x))$

What is the transition relation?

$F_R(x, y, z, x', y', z') =$
$((z' \leftrightarrow z) \wedge (y' \leftrightarrow y) \wedge (x' \leftrightarrow x)) \vee ((x' \leftrightarrow y) \wedge (y' \leftrightarrow z) \wedge (z' \leftrightarrow y))$

SMT solvers: Motivation

SAT solvers are automatic and efficient.

As a result, they are frequently used as the “engine” behind verification applications.

However, systems are usually designed and modeled at a higher level than the Boolean level and the translation to Boolean logic can be expensive and confusing.

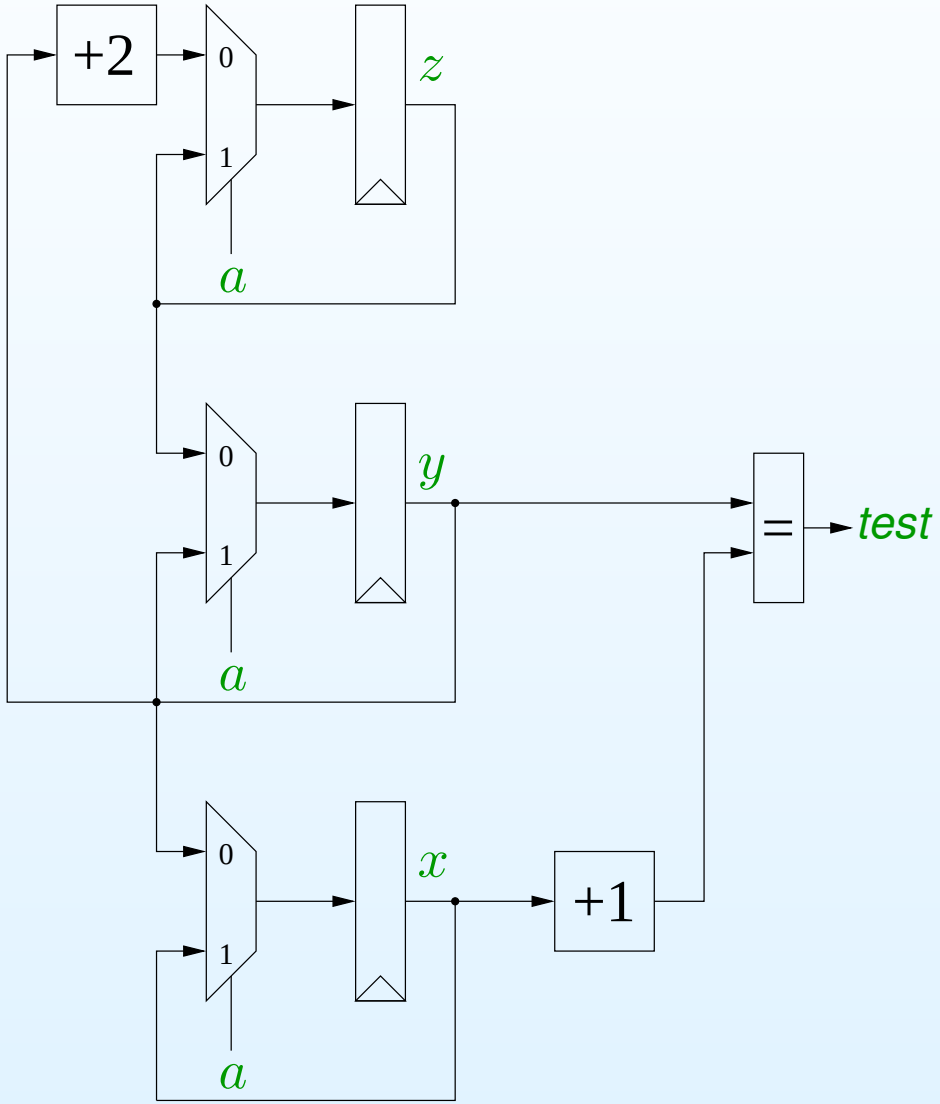
A primary goal of research in *Satisfiability Modulo Theories (SMT)* [BT18] is to create verification engines that can reason natively at a higher level of abstraction, while still retaining the speed and automation of today’s Boolean engines.

Modeling

The language of SMT allows us to model at a higher level of abstraction.

Consider again the circuit example.

Running Example



Circuit Example

Suppose now that every register is modeled as an integer and we are allowed to use arithmetic operators and equality.

What is the property?

Circuit Example

Suppose now that every register is modeled as an integer and we are allowed to use arithmetic operators and equality.

What is the property?

$F_{S_P}(x, y, z) = (z = y + 1) \wedge (y = x + 1)$

Circuit Example

Suppose now that every register is modeled as an integer and we are allowed to use arithmetic operators and equality.

What is the property?

$F_{S_P}(x, y, z) = (z = y + 1) \wedge (y = x + 1)$

What is the base case?

Circuit Example

Suppose now that every register is modeled as an integer and we are allowed to use arithmetic operators and equality.

What is the property?

$$F_{S_P}(x, y, z) = (z = y + 1) \wedge (y = x + 1)$$

What is the base case?

$$F_{S_0}(x, y, z) \rightarrow F_{S_P}(x, y, z) =$$
$$((x = x_0) \wedge (y = y_0) \wedge (z = z_0)) \rightarrow ((z = y + 1) \wedge (y = x + 1))$$

Circuit Example

Suppose now that every register is modeled as an integer and we are allowed to use arithmetic operators and equality.

What is the property?

$$F_{S_P}(x, y, z) = (z = y + 1) \wedge (y = x + 1)$$

What is the base case?

$$F_{S_0}(x, y, z) \rightarrow F_{S_P}(x, y, z) =$$
$$((x = x_0) \wedge (y = y_0) \wedge (z = z_0)) \rightarrow ((z = y + 1) \wedge (y = x + 1))$$

What is the transition relation?

Circuit Example

Suppose now that every register is modeled as an integer and we are allowed to use arithmetic operators and equality.

What is the property?

$$F_{S_P}(x, y, z) = (z = y + 1) \wedge (y = x + 1)$$

What is the base case?

$$F_{S_0}(x, y, z) \rightarrow F_{S_P}(x, y, z) =$$
$$((x = x_0) \wedge (y = y_0) \wedge (z = z_0)) \rightarrow ((z = y + 1) \wedge (y = x + 1))$$

What is the transition relation?

$$F_R(x, y, z, x', y', z') =$$
$$((z' = z) \wedge (y' = y) \wedge (x' = x)) \vee ((x' = y) \wedge (y' = z) \wedge (z' = y + 2))$$

Modeling

Notice that at this level of abstraction, we can prove the formula is true for arbitrary integers, eliminating the need to consider the size of the registers

Alternatively, if we are concerned about overflow, we can use the theory of bitvectors, which still has the advantage that the size of the formula does not increase with increasing bit-width.

Model Checking

Model Checking is a verification technique which automatically checks whether a *model* satisfies a given property [CGP02].

This is done by enumerating (either explicitly or symbolically) a set of *states* of the model, and checking that each state satisfies the property.

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:

Syntax of *CTL**

- 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**.

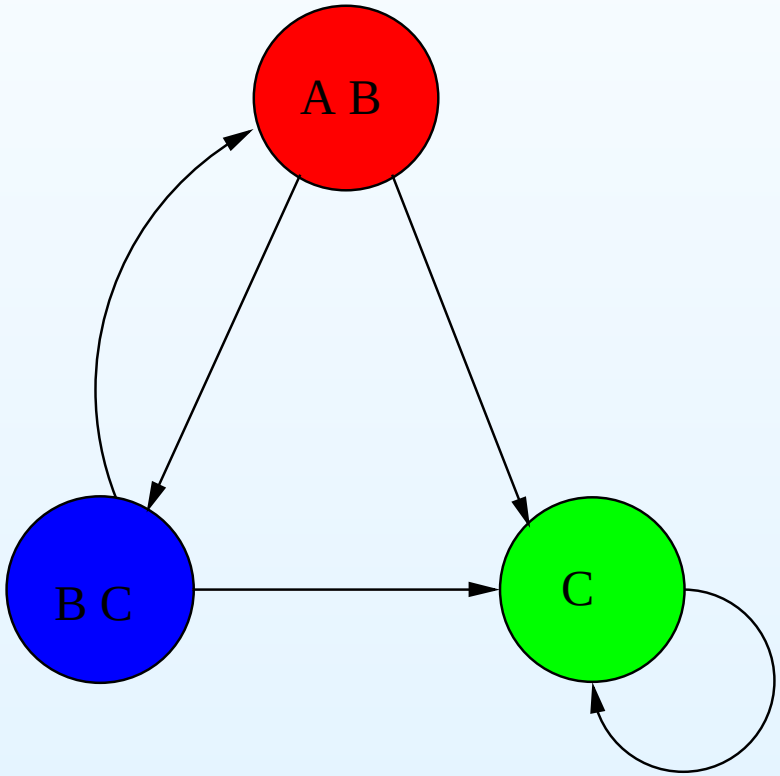
Semantics of CTL*

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.

Semantics of CTL*

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.



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 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.

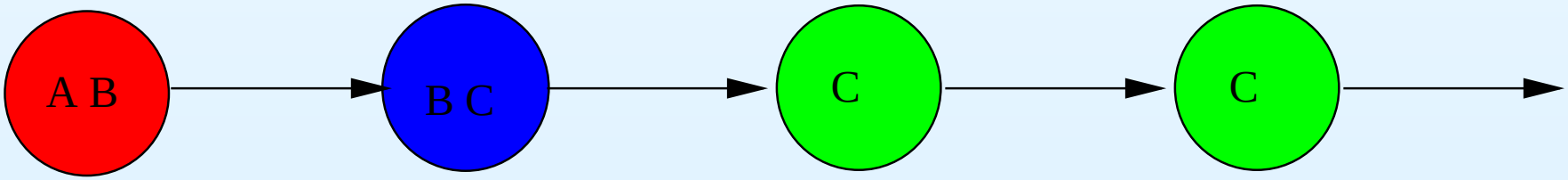
By repeatedly appplying this operator, we can specify that a property holds in the n^{th} state of the path.

X *operator*

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.

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

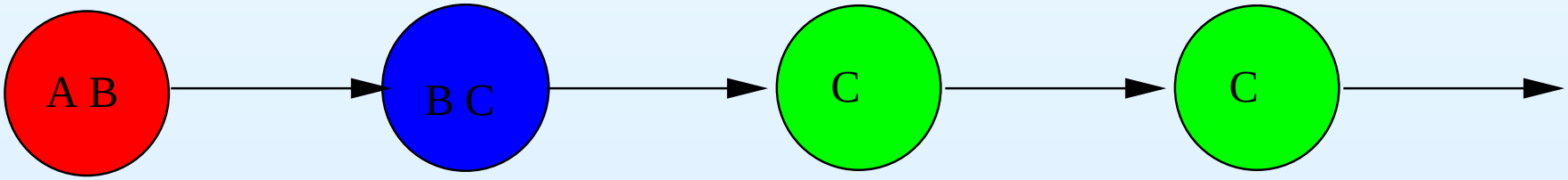


X *operator*

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

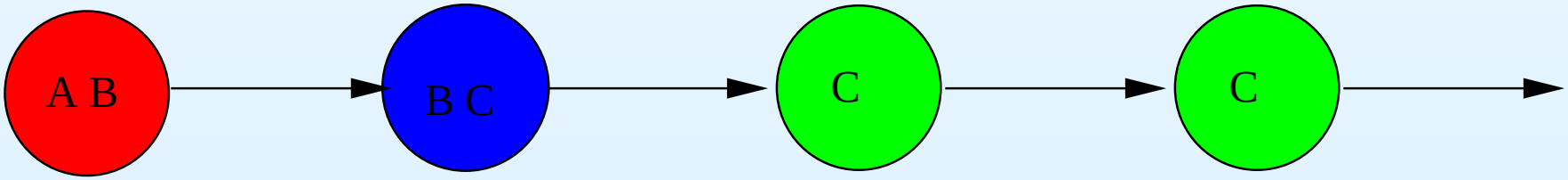


X *operator*

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*
- $\mathbf{X}(C)$
- $\mathbf{X}\,\mathbf{X}(C)$

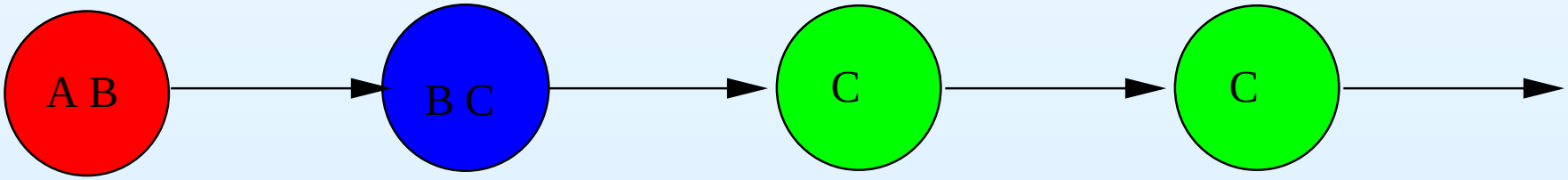


X *operator*

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*
- **X** $(A \wedge B)$ *False*
- **X** (C) *True*
- **X X** (C)

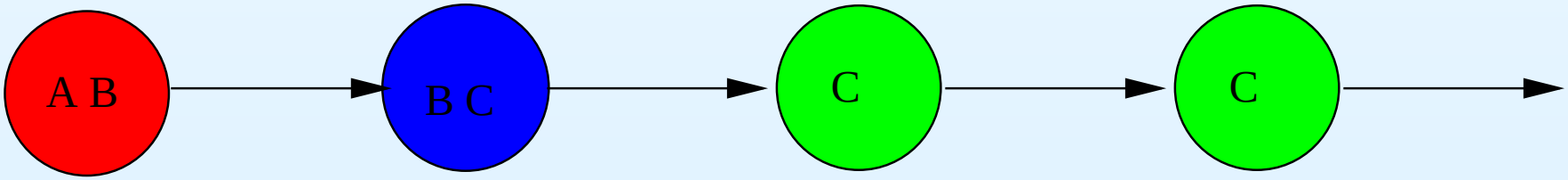


X *operator*

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*
- **X** $(A \wedge B)$ *False*
- **X** (C) *True*
- **X X** (C) *True*



U *operator*

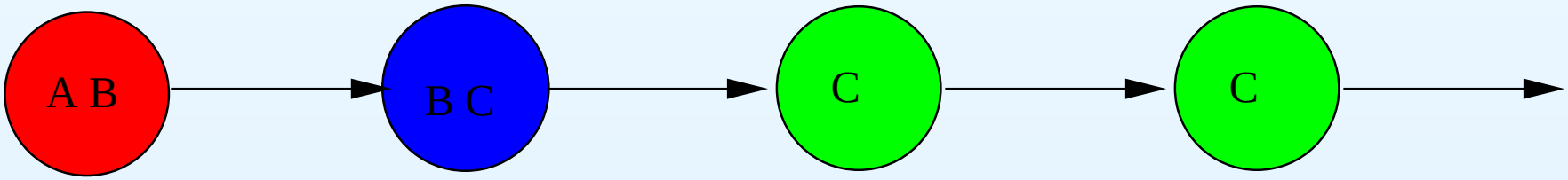
U (“until”) is a binary operator which asserts that the first property holds for every state on a path up to but not necessarily including a state in which the second property holds.

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

U *operator*

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

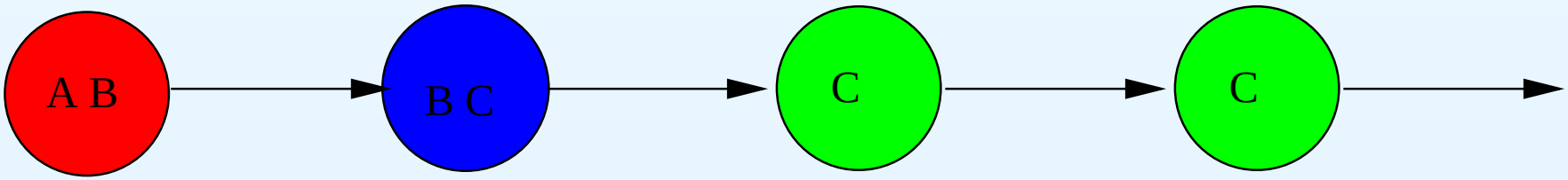
- $A \,\mathbf{U}\, \neg B$
- $B \,\mathbf{U}\, C$
- $\mathbf{X}\,(\neg A \,\mathbf{U}\, \neg B)$
- $\mathbf{X}\,(C \,\mathbf{U}\, A)$



U *operator*

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

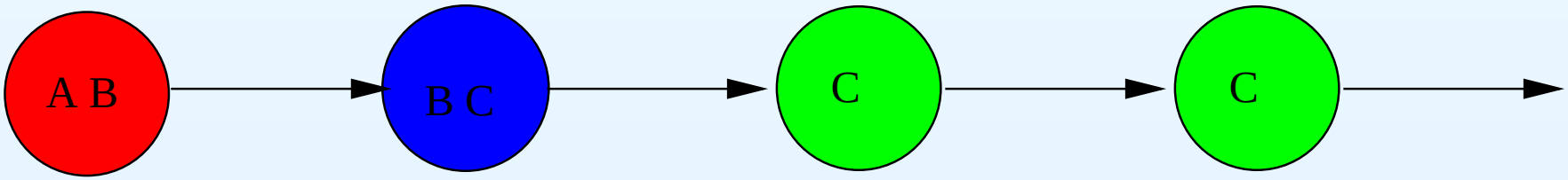
- $A \mathbf{U} \neg B$ *False*
- $B \mathbf{U} C$
- $\mathbf{X}(\neg A \mathbf{U} \neg B)$
- $\mathbf{X}(C \mathbf{U} A)$



U *operator*

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

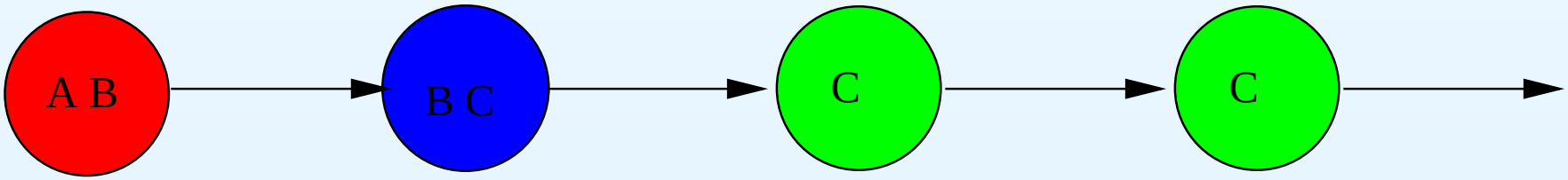
- $A \mathbf{U} \neg B$ *False*
- $B \mathbf{U} C$ *True*
- $\mathbf{X}(\neg A \mathbf{U} \neg B)$
- $\mathbf{X}(C \mathbf{U} A)$



U *operator*

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

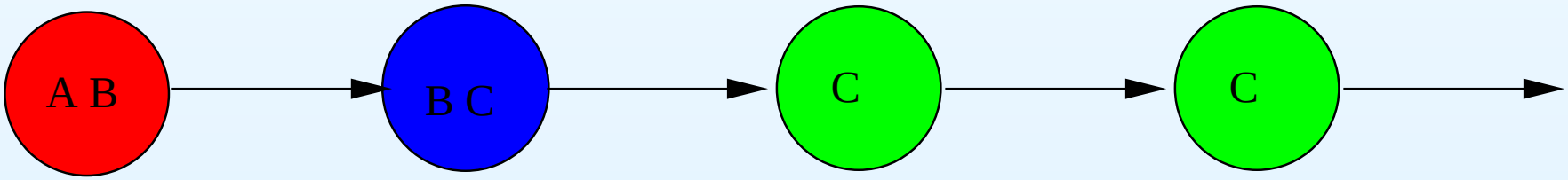
- $A \mathbf{U} \neg B$ *False*
- $B \mathbf{U} C$ *True*
- $\mathbf{X}(\neg A \mathbf{U} \neg B)$ *True*
- $\mathbf{X}(C \mathbf{U} A)$



U *operator*

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:

- $\mathbf{F}\, f = \mathit{True}\, \mathbf{U}\, f$ ("eventually" or "in the future") asserts that f holds at some state on the path.
- $\mathbf{G}\, f = \neg \mathbf{F} \neg f$ ("always" or "globally") specifies that f holds at every state on the path.
- $f\, \mathbf{R}\, g = \neg(\neg f\, \mathbf{U}\, \neg g)$ ("release") requires that g holds up to and including the first state where the f holds. Unlike $\mathbf{U}$, 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.

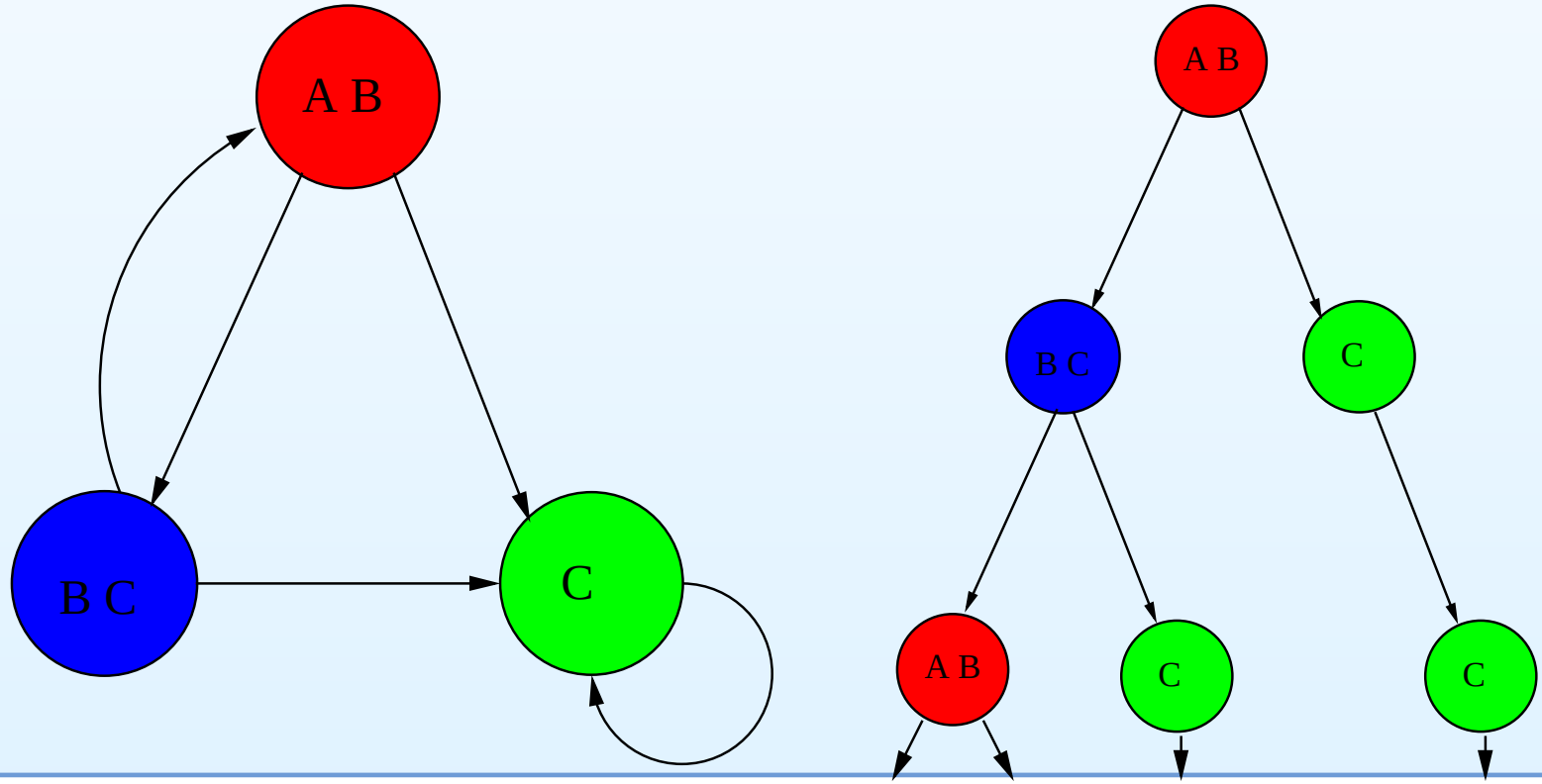
To interpret these formulas relative to a given state s, we consider the computation tree rooted at s.

- **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?

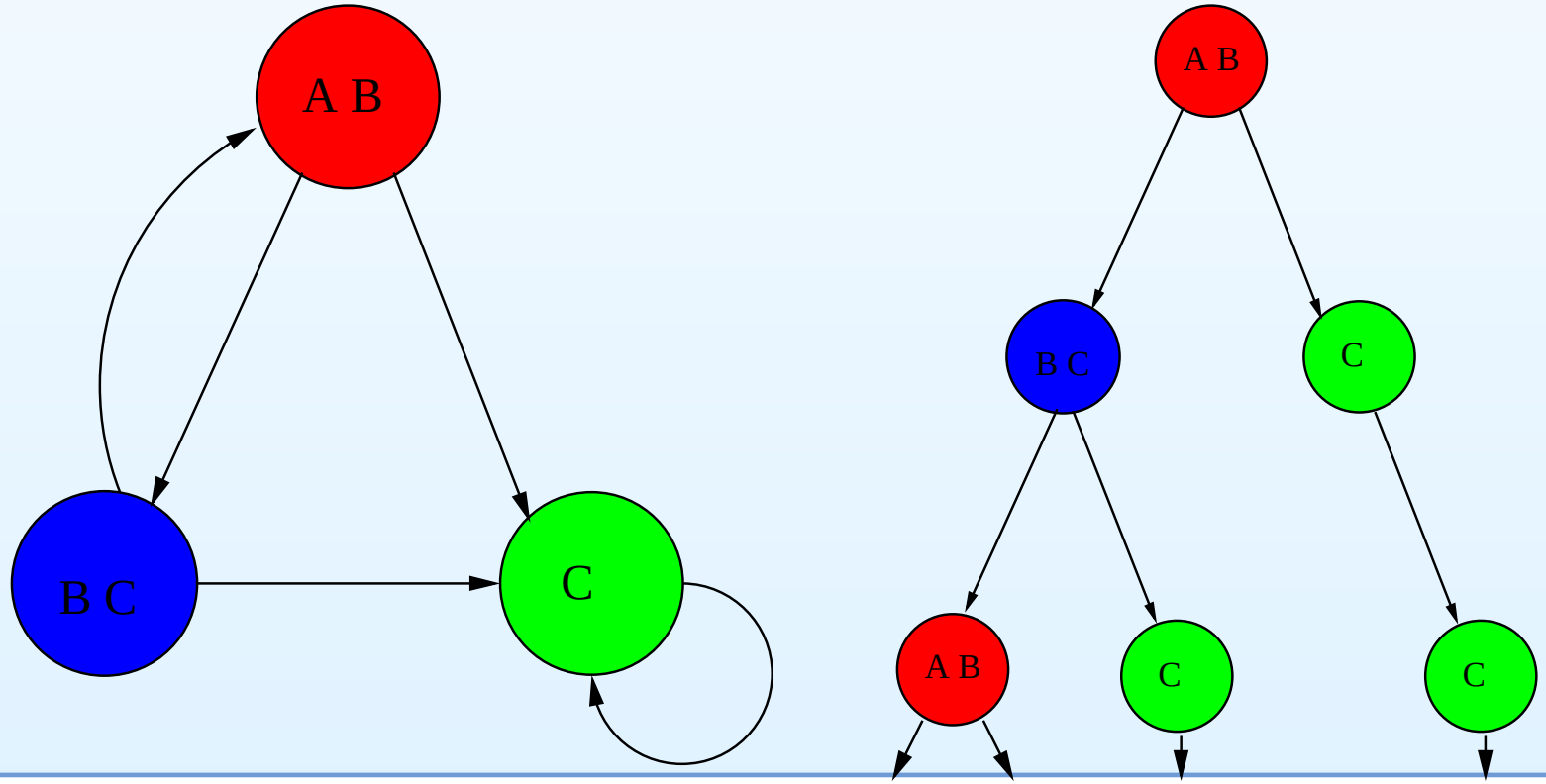
- $\mathbf{EG}(C)$
- $\mathbf{AF}(C)$
- $\mathbf{AG}(C \vee \mathbf{X}(C))$
- $\mathbf{EX}(\mathbf{AG}(C))$



Path Quantifiers Example

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

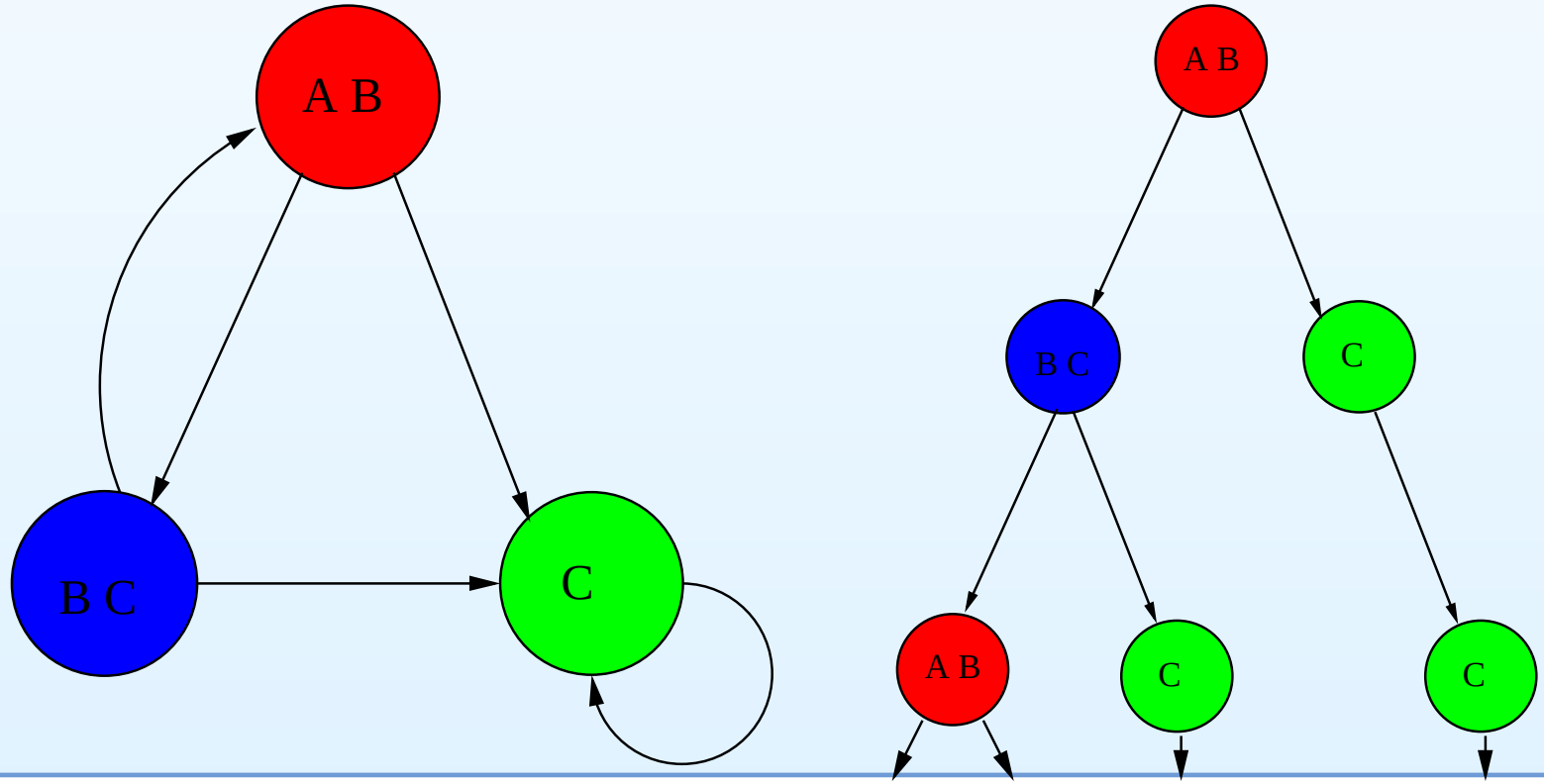
- $\mathbf{EG}(C)$ *False*
- $\mathbf{AF}(C)$
- $\mathbf{AG}(C \vee \mathbf{X}(C))$
- $\mathbf{EX}(\mathbf{AG}(C))$



Path Quantifiers Example

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

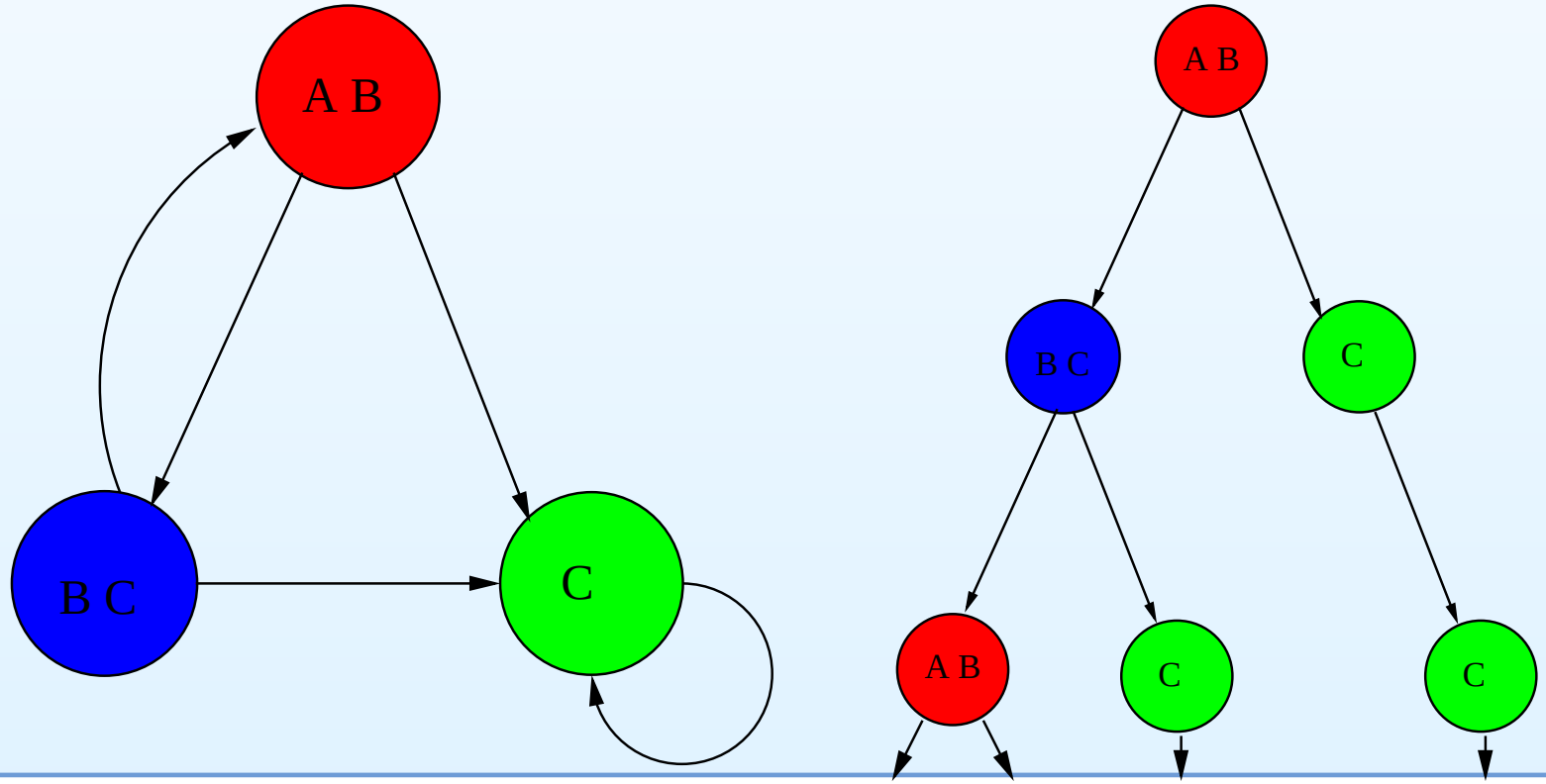
- $\mathbf{EG}(C)$ *False*
- $\mathbf{AF}(C)$ *True*
- $\mathbf{AG}(C \vee \mathbf{X}(C))$
- $\mathbf{EX}(\mathbf{AG}(C))$



Path Quantifiers Example

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

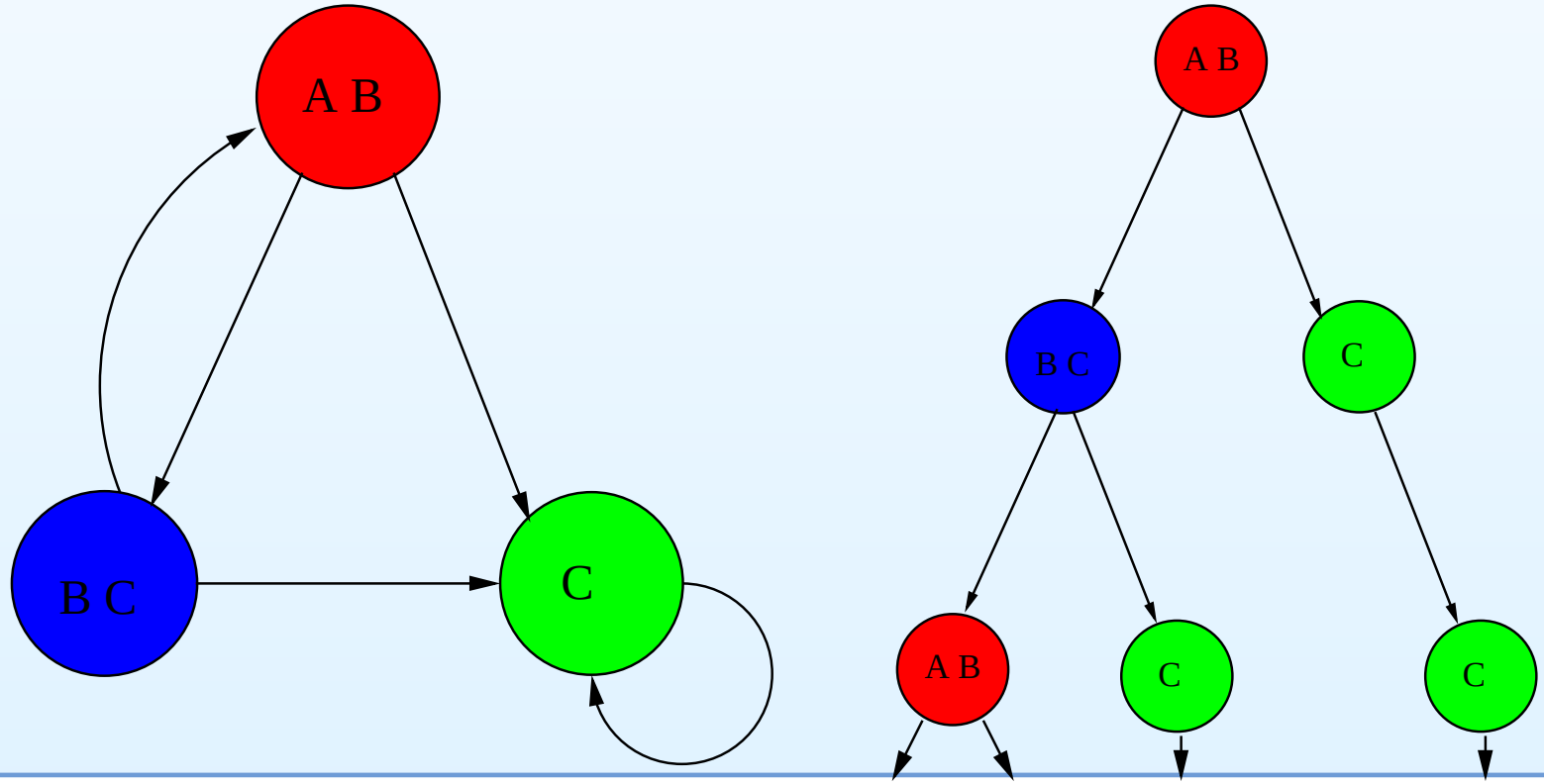
- $\mathbf{EG}(C)$ *False*
- $\mathbf{AF}(C)$ *True*
- $\mathbf{AG}(C \vee \mathbf{X}(C))$ *True*
- $\mathbf{EX}(\mathbf{AG}(C))$



Path Quantifiers Example

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

- $\mathbf{EG}(C)$ *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.

CTL *and* LTL

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: $p \in a^* \mid \neg\beta \mid \beta \vee \beta \mid \beta \wedge \beta \mid \mathbf{X}(\beta) \mid \mathbf{F}(\beta) \mid \mathbf{G}(\beta) \mid \beta\,\mathbf{U}\,\beta \mid \beta\,\mathbf{R}\,\beta$

Typical 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.

Model Checking

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

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**.

Explicit State CTL *Model Checking*

- *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* $(\mathbf{E}\,(g_1\, \mathbf{U}\, g_2)) =$ *CheckEU* (g_1, g_2)
- *Check* $(\mathbf{EG}\,(g)) =$ *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
```

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 $\mathbf{x}$ 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* $\mathbf{y}$. 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 $p \in L(s)$.

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.

Symbolic model checking can scale many orders of magnitude beyond explicit state model checking.

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.

Bounded Model Checking

Suppose we are checking $\mathbf{AG}(P)$. P is called a *safety property*.

Bounded Model Checking [BCCZ99, CBRZ01] can be used to determine whether P holds after some bounded number of transitions.

To perform bounded model checking to a depth of k using SAT or SMT, we need k extra copies of the state variables.

Let $\mathbf{x_0}, \ldots, \mathbf{x_k}$ be $k+1$ copies of the state variables. And let $F_{S_P}(\mathbf{x})$ be a formula that is true iff the property P holds.

Then P holds after k steps iff the following formula is valid:

$$(F_{S_0}(\mathbf{x_0}) \wedge F_R(\mathbf{x_0}, \mathbf{x_1}) \wedge \cdots \wedge F_R(\mathbf{x_{n-1}}, \mathbf{x})) \rightarrow F_{S_P}(\mathbf{x_n}).$$

CoSA Model Checker

The *CoSA* model checker [MMB+18] is an open-source model-checker for hardware

It reads in verilog with properties and can then use a variety of model checking techniques to try to prove the properties.

In particular, CoSA can do bounded and k-induction based model checking using SMT formulas and solvers

References

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1573 of *LNCS*, pages 193–207. Springer-Verlag, 1999

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Formal Methods in System Design*, 19(1):7–34, 2001

[CGP02] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2002

[MMB+18] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan. CoSA: Integrated verification for agile hardware design. In Nikolaj Bjørner and Arie Gurfinkel, editors, *Proceedings of the* 18^{th} *International Conference on Formal Methods In Computer-Aided Design (FMCAD '18)*, pages 7–11. FMCAD Inc., October 2018. Austin, Texas