

# CS6133 (Fall 2011)

## Software Specification and Verification

### Lecture 8: BDDs, $\mu$ -Calculus Model Checking

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.1/45

## Review: Model Checking

### Problem

Given a model  $\mathcal{M}$  (usually a Kripke structure) that represents the behaviour of a system, and a temporal logic formula  $f$  that represents a desired property of the system, determine whether the model satisfies the formula:

$$\mathcal{M} \models f$$

Model checking algorithms we're studying:

- Explicit CTL Model Checking: labelling a graph
- Symbolic CTL Model Checking: representing sets of states using Boolean functions

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.2/45

## Review: Symbolic Model Checking

1. Describe the model checking algorithm as computations over sets of states; abstract the operation of finding the set of states that can reach another set of states as a **pre-image** computation
2. Use Boolean characteristic functions to represent sets of states
  - Encode the set of states as Boolean vectors using the labelling function.
  - Represent the transition relation using a Boolean function.
  - Represent the atomic propositions of the temporal logic formula as Boolean functions.
3. Implement the set operations in the model checking algorithm as operations on Boolean functions.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.3/45

## Agenda

- Binary Decision Diagrams – a data structure for manipulating Boolean functions
- CTL model checking as fixed point operations in the  $\mu$ -calculus

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.4/45

## Representing Boolean functions

Both **truth tables** and **propositional formulae** are different ways of representing Boolean functions.

When implementing automated logical reasoning on a computer, we want to have **compact** representations of Boolean formulae with **efficient** operations on those formulae.

A representation for Boolean formulae plus a way of determining the validity of those formulae is a **proof procedure** for propositional logic.

Other relevant operations are:

- constructing formulae
- comparing two formulae for equality (a special case of validity)
- checking satisfiability (for counterexamples)

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.5/45

## Worst Case Limits

Most of the rest of the presentation is based on Kropf [Kro99]

Every representation of Boolean functions has the same problem complexity:

Determining the satisfiability of a formula is NP complete; determining the validity of a formula is co-NP complete.

This means in the worst case we have exponential run-time.

But can we do better for many cases?

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.6/45

## Binary Decision Trees

A Boolean function can be represented by an **decision tree** (a rooted, directed tree).

Non-terminal vertices are labelled by Boolean variables and have two branches corresponding to the cases when the value of the variable is T and F.

The terminal vertices are labelled with T or F.

The value of the function for its inputs is obtained by traversing the decision nodes from root to leaf.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.7/45

## Shannon's Expansion Theorem

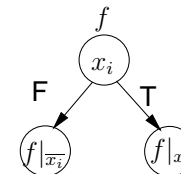
Binary decision trees are a graphical representation of the **Shannon expansion**.

The **cofactor** of  $f$  with regard to  $x_i$ , written as  $f|_{x_i}$  is  $f(x_1, \dots, x_{i-1}, \mathbf{T}, x_{i+1}, \dots, x_n)$

The **cofactor** of  $f$  with regard to  $\overline{x_i}$ , written as  $f|_{\overline{x_i}}$  is  $f(x_1, \dots, x_{i-1}, \mathbf{F}, x_{i+1}, \dots, x_n)$

**Shannon's Expansion:**

$$f(x_1, \dots, x_i, \dots, x_n) = (x_i \cdot f|_{x_i}) + (\overline{x_i} \cdot f|_{\overline{x_i}})$$



Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.8/45

## Reduced BDDs

The tree representation contains redundant information. Akers had the idea to **merge redundant information** to create reduced BDDs. Use a directed, acyclic graph (dag) instead of a tree to merge redundancies.

- All isomorphic subtrees are combined.
- All nodes with isomorphic children are eliminated.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.9/45

## ROBDD reduction

Visit the nodes in a bottom-up order doing the combinations possible.

```
merge leaves with the same value;
for i := n downto 1 do
  for all nodes v with Index(v) = i do
    if left(v) = right(v)
    then replace v by left(v);
    if exists v'. (Index(v') = i
                  and left(v') = left(v)
                  and right(v') = right(v))
    then substitute v by v'
```

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.10/45

## Reduction Example [Kro99], p. 43

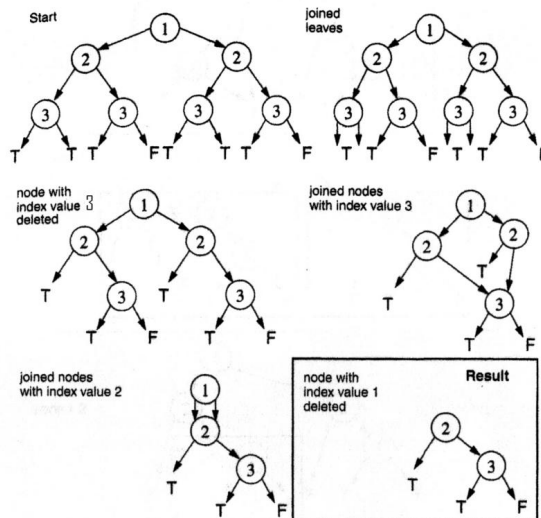


Fig. 2-5. ROBDD reduction example Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.11/45

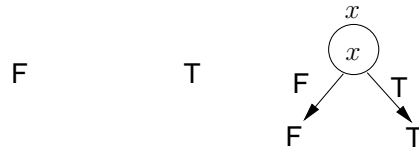
## Reduced Ordered BDDs

Bryant [Bry86, Bry92] had the idea to impose a variable ordering on the BDDs. On every path from the root to a terminal node, the variables appear in the same order with no repeated variables. All variables do not have to appear in all paths. Reduced Ordered BDDs are often just called BDDs.

The result is a **canonical** representation of a Boolean function for a given variable ordering.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.12/45

## Simplest ROBDDs



Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.13/45

## Comparison, Satisfiability, Validity

If we have a canonical representation for Boolean functions, comparison of two ROBDDs can be done by checking if their representations have the same structure. Usually this means checking if two pointers are equal (constant time).

**Validity:** check if the ROBDD is equal to the ROBDD for the constant true function.

**Satisfiability:** check that the ROBDD isn't equal to the ROBDD for the constant false.

ROBDDs can be considered a proof procedure for propositional logic.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.14/45

## Operations on ROBDDs

- equality – pointer equality
- negation – complement the leaf values
- conjunction, disjunction – apply a binary operation
- pre-image – exists and conjunction

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.15/45

## Apply

To determine the result of applying a Boolean operation to two ROBDDs use the Shannon expansion.

$$f = f_1 \star f_2$$

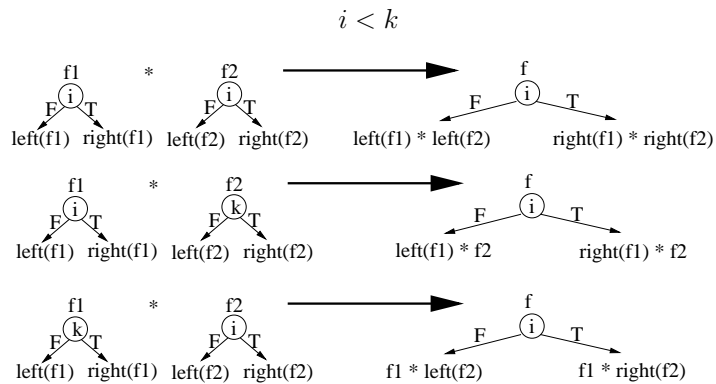
$$\begin{aligned} f &= x_i \cdot f|_{x_i} + \overline{x_i} \cdot f|_{\overline{x_i}} \\ &= x_i \cdot (f_1 \star f_2)|_{x_i} + \overline{x_i} \cdot (f_1 \star f_2)|_{\overline{x_i}} \\ &= x_i \cdot (f_1|_{x_i} \star f_2|_{x_i}) + \overline{x_i} \cdot (f_1|_{\overline{x_i}} \star f_2|_{\overline{x_i}}) \end{aligned}$$

(The cofactor is distributive with regard to Boolean operations.)

$\star$  is any binary operation.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.16/45

## Apply [Kro99], p44



Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.17/45

## Apply Optimizations

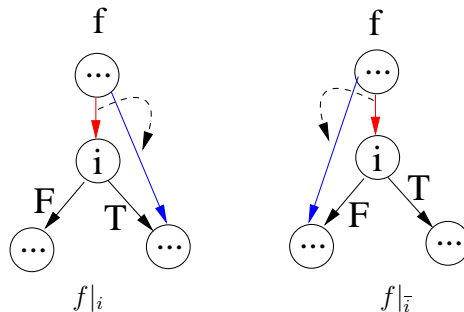
Simple optimizations:

- If one argument is T or F, it can often immediately be reduced.
- Use a hash table to cache results.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.18/45

## Cofactor (Restriction)

Cofactoring (restriction):  $f|_i$   
Eliminate all nodes labelled by  $i$ .



Composition of two ROBDDs  $f_1|_{x_i=f_2}$  can be done in terms of existing operations:  $f_1|_{x_i=f_2} = (f_2 \cdot f_1|_{x_i}) + (\bar{f}_2 \cdot f_1|_{x_i})$

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.19/45

## Pre-image

$\chi_{\text{pre}_\exists} Y(v_0, \dots, v_n) =$

$$\exists v'_0, \dots, v'_n \bullet \text{replace } Y(v'_0 | v_0, \dots, v'_n | v_n) \cdot \chi_R(v_0, \dots, v_n, v'_0, \dots, v'_n)$$

In ROBDDs, **existential** quantification can be implemented using conjunction and restriction:  $\exists x. f = f|_{x=F} \vee f|_{x=T}$

We can compute the pre-image using  $\exists$  and  $\wedge$  but this is such a frequent operation that we do better by implementing this operation directly.

The **relational product** is:  $\exists x. R \wedge Y$ , where  $R$  is a function of the Boolean vectors  $x$  and  $x'$ , and  $Y$  is a function of  $x$ . ROBDD packages include implementations of this operation.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.20/45

## Heuristics for Var Order

Different variable orders can produce quite different ROBDDs.

**Rough heuristic:** group related variables together.

For example, in a ripple-carry adder, interleave the variables from each operand.

ROBDDs do not work well for representing multiplication because all variables are related to each other.

Lots of packages do **dynamic variable reordering** (on-the-fly attempts at reducing the size of the ROBDD through local optimizations).

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.21/45

## ROBDD packages

- usually implemented using pointers
- multi-rooted ROBDD to represent many ROBDDs all using same variable ordering
- negative pointers to represent negation of a ROBDD

Existing packages:

- VIS <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/> (does model checking also)
- CMU <http://www-2.cs.cmu.edu/modelcheck/bdd.html>
- CUDD <http://vlsi.colorado.edu/fabio/CUDD/cuddIntro.html>

Note: there are many variety of decision diagrams such as multi-way decision graphs [CZS<sup>+</sup>94], and binary moment diagrams [BC94].

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.22/45

## Symbolic Model Checking

The first paper to describe symbolic model checking used ROBDDs to represent sets of states. This paper is by Burch, Clarke, McMillan, Dill, and Hwang [BCM<sup>+</sup>90].

Explicit state model checking could handle systems of size  $10^3 - 10^6$  states.

Symbolic model checking could handle models of size  $10^{20}$ .

(These numbers are from the 1990 paper – they are higher now.)

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.23/45

## Agenda

- Binary Decision Diagrams – a data structure for manipulating Boolean functions
- CTL model checking as fixed point operations in the  $\mu$ -calculus
  - Symbolic when we represent the sets of states by Boolean functions

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.24/45

## $\mu$ -Calculus

- Notation for describing properties of transition systems
- Uses fixed point operators in addition to logical connectives (no temporal operators)
- Many temporal logics can be expressed as  $\mu$ -calculus formulae

We will only cover enough of the  $\mu$ -calculus to show how to encode CTL formulae.

## Fixpoints: Intuition

SAT\_EU and SAT\_EG are iterative until they reach a point where “no new relevant states are being considered”. This corresponds to the notion of a fixed point.

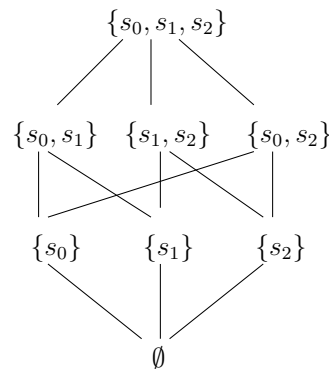
function SAT_EG( $f_1$ )	function SAT_EU ( $f_1, f_2$ )
$K := SAT(f_1);$	$K = SAT(f_2); W = SAT(f_1);$
do	do
oldK := K;	oldK := K;
$K := oldK \cap pre_{\exists}(oldK);$	$K := oldK \cup (W \cap pre_{\exists}(oldK))$
until oldK = K;	until oldK = K
return K;	return K;

## Power Set of a Set

For the Kripke structure  $\mathcal{M} = (S, R, L)$ , the set  $\mathbb{P}(S)$  is the set of all subsets of  $S$ .

The set of all subsets forms a lattice with the ordering of set inclusion.

The least element is the empty set ( $\emptyset$ ). The greatest element is the set  $S$ .



$$S = \{s_0, s_1, s_2\}$$

## Fixed Points

A **fixed point** of a function  $f$  is an element  $x$  such that  $f(x) = x$ .

The **least fixed point** in a lattice for a function  $f$  is the least element that is a fixed point.  $y$  is the lfp of  $f$  in  $S$  iff

$$(f(y) = y) \wedge (\forall x \in S \bullet (f(x) = x) \Rightarrow (y \subseteq x))$$

The least fixed point of the function  $f$  is denoted as  $\mu Z.f(Z)$

The **greatest fixed point** in a lattice for a function  $f$  is the greatest element that is a fixed point.  $y$  is the gfp of  $f$  in  $S$  iff

$$(f(y) = y) \wedge (\forall x \in S \bullet (f(x) = x) \Rightarrow (x \subseteq y))$$

The greatest fixed point of the function  $f$  is denoted as  $\nu Z.f(Z)$

## Fixed Points Example

Example:  $f_1(Y) = Y \cup \{s_0\}$

$Y$	$f_1(Y)$
$\{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}$
$\{s_0, s_1\}$	$\{s_0, s_1\}$
$\{s_1, s_2\}$	$\{s_0, s_1, s_2\}$
$\{s_0, s_2\}$	$\{s_0, s_2\}$
$\{s_0\}$	$\{s_0\}$
$\{s_1\}$	$\{s_0, s_1\}$
$\{s_2\}$	$\{s_0, s_2\}$
$\emptyset$	$\{s_0\}$

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.29/45

## Monotonic Functions

For a special kind of function, there is a way to compute the least and greatest fixed points.

A function  $f : \mathbb{P}(S) \rightarrow \mathbb{P}(S)$  is **monotonic** iff for all  $P \subseteq S$  and  $Q \subseteq S$

$$P \subseteq Q \Rightarrow f(P) \subseteq f(Q)$$

(Also called monotone functions.)

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.30/45

## Useful Theorems

**Thm 1:** A monotonic function always has a least fixed point and a greatest fixed point. These are:

$$\mu Z.f(Z) = \cap \{Z \mid f(Z) \subseteq Z\}$$

$$\nu Z.f(Z) = \cup \{Z \mid Z \subseteq f(Z)\}$$

**Thm 2:** If  $S$  is finite and  $f$  is a monotonic function, then for some  $i$ :

$$\mu Z.f(Z) = \cup_i f^i(\emptyset)$$

and

$$\nu Z.f(Z) = \cap_i f^i(S)$$

This gives us a way to **compute** the lfp and gfp.

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.31/45

## Calculating Least Fixed Points

$$\begin{aligned} \mu Z.f(Z) &= \cup_i f^i(\emptyset) \\ &= f(\emptyset) \cup f(f(\emptyset)) \cup f(f(f(\emptyset))) \dots \end{aligned}$$

**Thm 3:** If  $f$  is monotonic,  $\forall i. f^i(\emptyset) \subseteq f^{i+1}(\emptyset)$

Thus,  $\forall i. f^{i+1}(\emptyset) = f^i(\emptyset) \cup f^{i+1}(\emptyset)$ .

$$\begin{aligned} \mu Z.f(Z) &= \cup_i f^i(\emptyset) \\ &= f(\emptyset) \cup f(f(\emptyset)) \cup f(f(f(\emptyset))) \dots \\ &= f^i(\emptyset) \text{ for some } i \end{aligned}$$

Because  $f$  is monotonic and  $S$  is finite,  $\exists i. f^i(\emptyset) = f^{i+1}(\emptyset)$ . This is a fixed point and is the least fixed point of  $f$ !

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.32/45

## Calculating Least Fixed Points

$\text{fp } f \ e = \text{if } (e = f \ e) \text{ then } e \text{ else fp } f \ (f \ e)$   
 $\text{lfp } f = \text{fp } f \ \emptyset$

## Calculating Greatest Fixed Points

$\nu Z. f(Z) = \bigcap_i f^i(S)$   
 $= f(S) \cap f(f(S)) \cap f(f(f(S))) \dots$

**Thm 4:** If  $f$  is monotonic,  $\forall i. f^{i+1}(S) \subseteq f^i(S)$

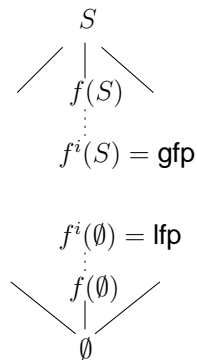
Thus,  $\forall i. f^{i+1}(S) = f^i(S) \cap f^{i+1}(S)$ .

$\nu Z. f(Z) = \bigcap_i f^i(S)$   
 $= f(S) \cap f(f(S)) \cap f(f(f(S))) \dots$   
 $= f^i(S) \text{ for some } i$

Because  $f$  is monotonic and  $S$  is finite,  $\exists i. f^i(S) = f^{i+1}(S)$ .  
This is a fixed point and is the greatest fixed point of  $f$ !

## Calculating Fixed Points

$\text{fp } f \ e = \text{if } (e = f \ e)$   
 $\quad \text{then } e$   
 $\quad \text{else fp } f \ (f \ e)$   
 $\text{lfp } f = \text{fp } f \ \emptyset$   
 $\text{gfp } f = \text{fp } f \ S$



This calculation looks familiar ...

## Checking EG

```

function SAT_EG( $f_1$ )
  K := SAT( $f_1$ );
  do
    oldK := K;
    K := oldK  $\cap$  pre $_{\exists}$ (oldK);
  until oldK = K;
  return K;

```

Unfolding the loop:

$K = \text{SAT}(f_1)$   
 $K = \text{SAT}(f_1) \cap \text{pre}_{\exists}(\text{SAT}(f_1))$   
 $K = \text{SAT}(f_1) \cap \text{pre}_{\exists}(\text{SAT}(f_1)) \cap \text{pre}_{\exists}(\text{SAT}(f_1) \cap \text{pre}_{\exists}(\text{SAT}(f_1)))$   
 ... until it reaches a fixed point.

## Checking EG

$$\text{SAT\_EG } f_1 = \nu Z. \text{SAT}(f_1) \cap \text{pre}_\exists Z$$

which is the same as

$$\text{SAT\_EG } f_1 = \nu Z. ((\lambda x. \text{SAT}(f_1) \cap \text{pre}_\exists x) Z)$$

Expanding the definition of gfp, we get:

$$\begin{aligned} \text{SAT\_EG } f_1 &= \text{gfp } (\lambda x. \text{SAT}(f_1) \cap \text{pre}_\exists x) \\ &= \text{fp } (\lambda x. \text{SAT}(f_1) \cap \text{pre}_\exists x) S \end{aligned}$$

Unfolding fp 3 times, we get:

$$\begin{aligned} \text{SAT\_EG } f_1 &= \\ &\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists S)) \end{aligned}$$

We continue this until we reach a fixed point.

## Checking EG

After 3 iterations:

From the algorithm we got:

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1)) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1)))$$

From the fixed point computation we got:

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists S))$$

These look very similar!

## Checking EG

In a Kripke structure  $\text{pre}_\exists S = S$  because all states are required to have a next state. Therefore, the fixed point computation of:

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists S))$$

can be reduced to:

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap S))$$

And  $X \cap S = X$ , so

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1)))$$

But we're still not done yet.

## Checking EG

The algorithm produced:

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1)) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1)))$$

A property of  $\text{pre}_\exists$  is that  $\text{pre}_\exists(A \cap B) \subseteq \text{pre}_\exists A$  (and similarly  $\text{pre}_\exists(A \cap B) \subseteq \text{pre}_\exists B$ ). This means

$$\text{pre}_\exists A \cap \text{pre}_\exists(A \cap B) = \text{pre}_\exists(A \cap B).$$

Therefore, we can reduce what the algorithm computes to:

$$\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1) \cap \text{pre}_\exists(\text{SAT}(f_1)))$$

Which is the same as what we derived from the fixed point calculation! (note: this is not a proof just an illustration!)

## Checking EG

Thus, a very succinct description of the algorithm is:

$$\text{SAT\_EG } f_1 = \nu Z. \text{SAT}(f_1) \cap \text{pre}_{\exists} Z$$

or

$$\text{SAT\_EG } f_1 = \text{gfp } (\lambda x. \text{SAT}(f_1) \cap \text{pre}_{\exists} x)$$

## Fixed Points for CTL

Identify each CTL formula  $f$  with the set of states satisfying that formula (thus  $\text{EX } f = \text{pre}_{\exists} f$ ):

- **AF**  $f = \mu Z. f \vee \text{AX } Z$
- **EF**  $f = \mu Z. f \vee \text{EX } Z$
- **AG**  $f = \nu Z. f \wedge \text{AX } Z$
- **EG**  $f = \nu Z. f \wedge \text{EX } Z$
- **A** $[f_1 \text{ U } f_2] = \mu Z. f_2 \vee (f_1 \wedge \text{AX } Z)$
- **E** $[f_1 \text{ U } f_2] = \mu Z. f_2 \vee (f_1 \wedge \text{EX } Z)$

Intuitively least fixed points correspond to eventualities, and greatest fixed points correspond to properties that should hold forever.

## Symbolic Fixed Point Calculations

As before, if we represent sets of states using Boolean functions then this is **symbolic** model checking.

The empty set is the predicate that always returns F. The set  $S$  (the top of the lattice) is the predicate that always returns T.

The set operations are implemented as operations on Boolean functions.

BDDs provide a constant time operation for checking to see if two Boolean functions are equal. This is a frequent operation in this calculation.

## Summary

- Binary Decision Diagrams – a data structure for manipulating Boolean functions
- CTL model checking as fixed point operations in the  $\mu$ -calculus
  - Symbolic when we represent the sets of states by Boolean functions

# Model Checking

We've studied:

- explicit CTL model checking
- symbolic CTL model checking
- symbolic fixpoint CTL model checking

There are also LTL, CTL\*, and  $\mu$ -calculus model checkers.

Next class: Symbolic Trajectory Evaluation (STE)

Copyright ©Nancy Day, 2002-2004; Permission is granted to copy without modification. – p.45/45

## References

- [BC94] Randal E. Bryant and Yimng-An Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, June 1994.
- [BCM<sup>+</sup>90] J. R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10<sup>20</sup> states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992. Preprint version published as CMU Technical Report CMU-CS-92-160.
- [CZS<sup>+</sup>94] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. Technical Report RC19676,

45-1

- [Kro99] IBM, 1994. Also *Formal Methods in Systems Design*, 10(1), pages 7-46, 1997.
- Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999.

45-2