THURSDAY, Session 2
Symbolic Model Checking

1 Introduction

Because of the state explosion problem, “explicit state” model checkers are limited in their application. One solution to this problem is to use an implicit “symbolic” representation of state transitions and state labelings. This technique is called symbolic model checking [3, 2].

Binary Decision Diagrams (Bryant, Akers) are a heuristically compact and easily manipulated representation for boolean functions.

This lecture will discuss how to

- Use BDD’s to represent the characteristic functions of sets and relations (*e.g.*, transition relation)
- Using fixed point characterizations of CTL operators, iteratively compute the set of states satisfying a CTL formula
- Avoid building and explicit state graph in model checking

We will note that BDD’s can exploit a certain kind of structure in the state space of the model (*i.e.*, limited mutual knowledge between components), allowing models with very large state spaces to be checked in some cases.

In addition, the techniques can be applied to

- language containment tests
- computing simulations, bisimulations, state machine equivalence
- CTL* model checking, using symbolic tableau procedure

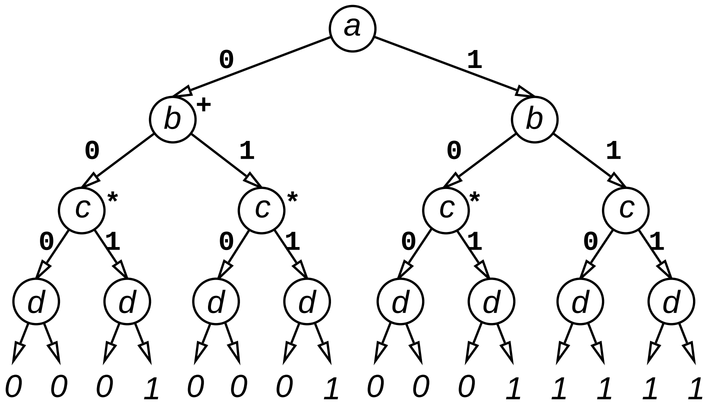
2 Binary Decision Diagrams

Binary decision diagrams [1] provide us with a representation of boolean functions with the following properties:

- Canonical form (constant time comparison)
- Polynomial algorithms for "and", "or", "not", *etc.*
- An exponential (but empiricially efficient) algorithm for boolean quantification

2.0.1 Ordered decision tree

Ordered decision tree for the function $ab + cd$:



Rules for obtaining the function value:

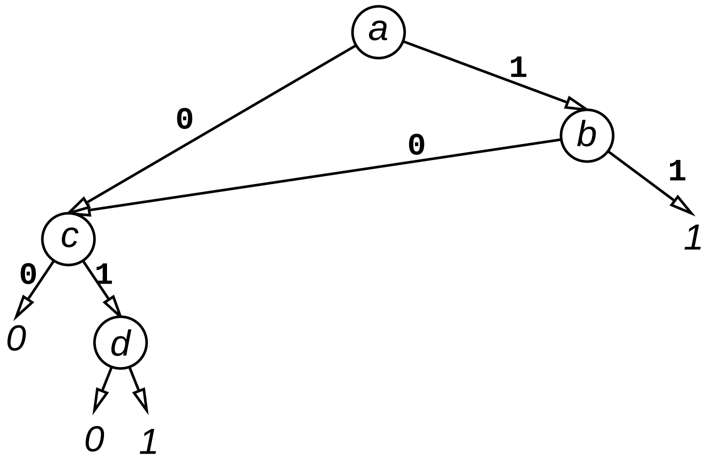
- Start at the root of the tree.
- At each node, take the "0" or "1" branch depending on the truth value of the given boolean variable.
- The leaf you reach gives the truth value of the function.

Requirement: the variables appear in the same order on any path from root to leaf. This requirement of a fixed order is very important for the results that follow.

Note: this tree is just an encoding of the truth table of the function.

2.0.2 Binary Decision Diagram

Binary Decision Diagram for the function $ab + cd$:



The BDD is obtained from the ordered decision tree by the following steps:

- Combine isomorphic subtrees.
- Eliminate redundant nodes (nodes with identical children).

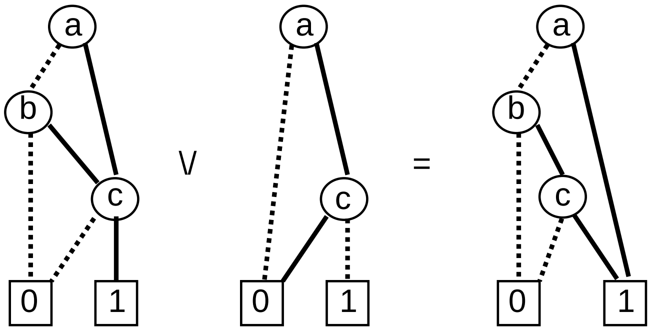
2.1 Apply algorithm

Given the OBDD for functions f and g, the *apply* algorithm builds the OBDD for $f \wedge g$, $f \vee g$, *etc.* In this way, we can build the OBDD for a complex function out of simpler functions.

The apply algorithm operates recursively on the structure of the two OBDD's:

1. Let v be the "top" variable in either f or g
2. Split problem into two subproblems for $v = 0$ and $v = 1$ and solve recursively
3. Subproblem results give 0 branch and 1 branch of result node. Do not create a new result node if:
   - Both branches equal (return common result)
   - Equivalent node already exists in "reduce" table (return it)

Example:



The terminating cases of the recursion are when both f and g are constants, in which case the result is a constant ($f \wedge g$, $f \vee g$, *etc.*).

If we "memoize" the apply function using a hash table, the number of subproblems to be solved is it most the number of pairs of nodes in f and g respectively, *i.e.*, $O(|f| \times |g|)$.

2.2 The quantification algorithm

In addition to the usual boolean operations, we will also need to be able to quantify existentially over boolean variables. If v is a boolean variable, then

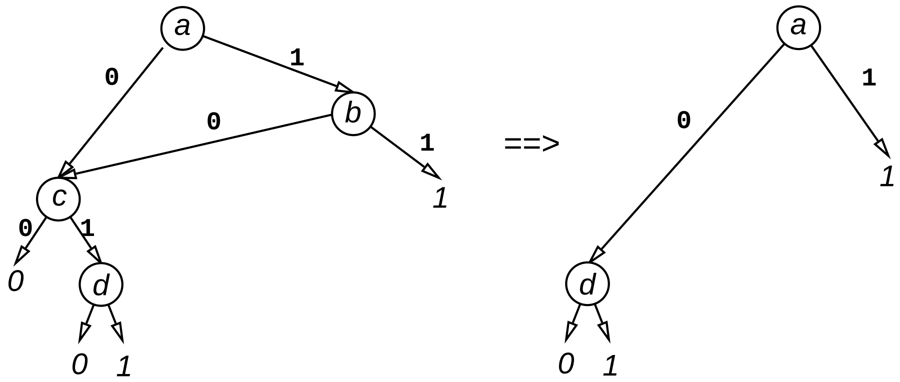
$$\exists v.f = f|_{v=0} \vee f|_{v=1}$$

The two cofactors $f|_{v=0}$ and $f|_{v=1}$ are easily obtained by simply replacing every node labeled v with its 0 or 1 branch respectively. However, we will require an efficient algorithm for quantifying over a set of variables, as in

$$\exists(w_1, w_2, \ldots, w_n).\ f$$

This is done by recursively replacing each OBDD node labeled with w_i with the logical or of its two branches.

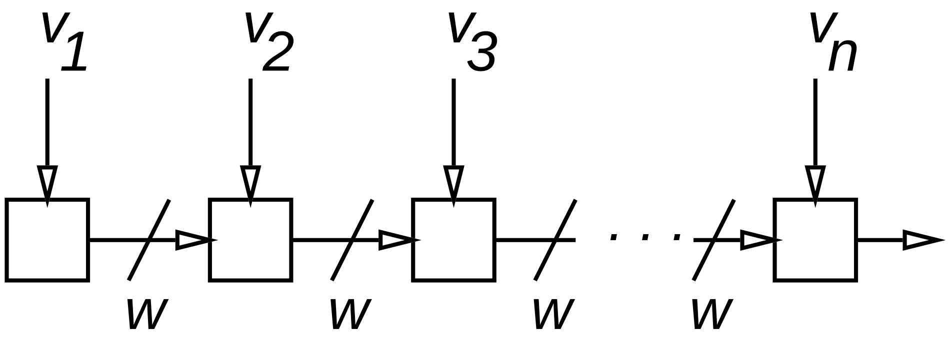
Example: $\exists(b, c).\ (ab \vee cd)\ =\ c \vee d$



In the worst case, multivariable quantification on OBDD's can result in an exponential blowup in size, but in practice, the OBDD size is most often reduced.

2.3 Circuit width and OBDD size

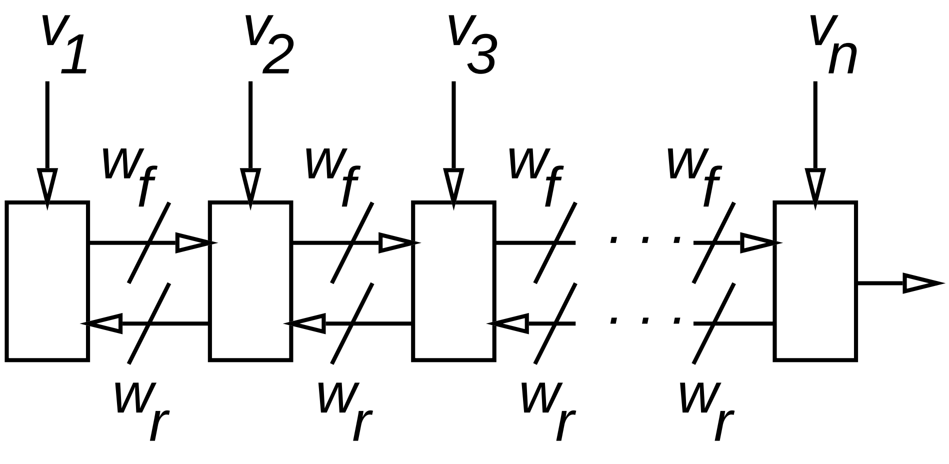
For a given topological sort of of the gates of a circuit, the *width* is the maximum number of wires crossing any cut:



For the corresponding order of the input variables, the OBDD size is bounded from above by $n2^w$. That is, as we increase circuit size, keeping the width constant, the OBDD grows at most linearly.

This is easy to see: having traversed the first i variables in the OBDD order, the function of the remaining variables is determined by the value of at most w wires. Since there are only 2^w values of w wires, there are at most 2^w inequivalent nodes at level i of the OBDD.

In case the order of the gates is not topological:



we have OBDD size $\leq n2^{w_f 2^{w_r}}$ (still linear in n).

A good example of a bounded width circuit is an adder, where only a single wire (the carry bit) passes from one adder cell to the next. Thus, the OBDD representation for any output bit of an adder is $O(n)$. The same argument applies to a parity chain.

A multiplier, on the other hand, has no constant width impementation, and consequently no efficient OBDD representation for any variable order.

2.4 Variable ordering

The OBDD representing a given function can vary exponentially in size for different variable orderings. A suitable variable order is usually chosen:

- By heuristics based on the structure of a given circuit
- By "dynamic reordering" which adjusts the variable ordering during computation using steepest descent heuristics

3 Representing sets and relations

Now we come to the question of model checking using OBDD's.

3.1 Characteristic functions of sets

Suppose our set of states S consists of all the boolean vectors

$$(v_1, v_2, \ldots, v_n) \in \{0,1\}^n$$

That is, we have *encoded* the entire state of our finite state program in terms of n boolean variables.

Any subset P of S can be represented implicitly by a characteristic formula χ_P, where

$$P = \{(v_1, v_2, \ldots, v_n) : \ \chi_P\}$$

This is, of course, the usual way in which mathematicians define sets –by "comprehension" expressions such as the above.

For example, the set $P = \{(0,1,0),(1,0,1)\}$ has characteristic:

$$\chi_P = (\neg v_1 \wedge v_2 \wedge \neg v_3) \vee (v_1 \wedge \neg v_2 \wedge v_3)$$

The boolean function χ_P might be represented, for example, by an OBDD.

We note that

- $\chi_\emptyset =$ false, $\chi_S =$ true
- $\chi_{P\cup Q} = \chi_P \vee \chi_Q$
- $\chi_{P\cap Q} = \chi_P \wedge \chi_Q$
- $\chi_{S\setminus P} = \neg P$

3.2 Characteristic functions of relations

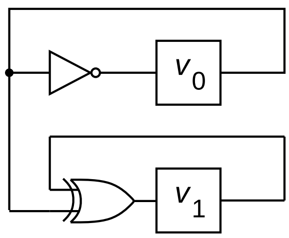
A relation R on states (*e.g.* the transition relation of a Kripke model) is a set of state pairs. A relation can be characterized by a boolean function over two symbolic vectors:

$$R = \{((v_1, v_2, \ldots, v_n), (v'_1, v'_2, \ldots, v'_n)) : \ \chi_R\}$$

If R is the transition relation of a Kripke model, then $(v_1, v_2, \ldots, v_n)$ represents the "pre-state" of a transitions and $(v'_1, v'_2, \ldots, v'_n)$ the "post-state".

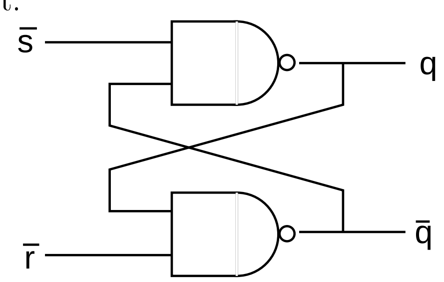
Some examples of characteristic functions for simple models:

- A sequential circuit:



$$\chi_R = (v'_0 = \neg v_0) \wedge (v'_1 = v_0 \oplus v_1)$$

- An asynchronous circuit:



– "interleaving" concurrency model:

$$\begin{aligned}\chi_R &= (q' = \neg(\bar{s} \wedge \bar{q})) \wedge (\bar{q}' = \bar{q}) \\ &\quad \vee (\bar{q}' = \neg(\bar{r} \wedge q)) \wedge (q' = q)\end{aligned}$$

In this model, only one gate can change its output at a time, and the other gate must keep its output unchanged.

– "simultaneous" concurrency model:

$$\begin{aligned}\chi_R &= (q' = \neg(\bar{s} \wedge \bar{q})) \vee (q' = q) \\ &\quad \wedge (\bar{q}' = \neg(\bar{r} \wedge q)) \vee (\bar{q}' = \bar{q})\end{aligned}$$

In this model, both gate may change simultaneously and independently.

- A shared-variables program

```
       ...
A::    x := ¬x;
B::    ...
```

$$\begin{aligned}\chi_R &= \ldots \\ &\quad \vee A \wedge B' \wedge (x' = \neg x) \wedge \text{other vars unchanged}\ldots \\ &\quad \vee \ldots\end{aligned}$$

- A message-passing program

```
A:: P!0; B:: skip ||
C:: Q?x; D:: skip
```

$$\begin{aligned}\chi_R &= \ldots \\ &\quad \vee A \wedge B' \wedge C \wedge D' \wedge (x' = 0) \wedge \text{other vars unchanged}\ldots \\ &\quad \vee \ldots\end{aligned}$$

3.3 Forward and reverse image

forward image

If P is a set of states and R a transition relation, the (forward) image of P with respect to R is the set of post-states related to pre-states in P (that is, the set of states that can be reached from P in one transition):

$$Image(P, R) = \{\mathbf{v}' : \text{ for some } \mathbf{v}, \, \mathbf{v} \in P \text{ and } (\mathbf{v}, \mathbf{v}') \in R\}$$

The forward image can be computed quite simply in terms of characteristic functions, using logical and, and boolean quantification:

$$\chi_{Image(P,R)}(\mathbf{v}') = \exists \mathbf{v}. \, (\chi_P(\mathbf{v}) \wedge \chi_R(\mathbf{v}, \mathbf{v}'))$$

Note, we have explicitly shown the parameters of the characteristic functions above. To get $\chi_{Image(P,R)}$ in our standard representation in terms of $\mathbf{v}$, we can just substitute v_i for v_i':

$$\chi_{Image(P,R)}(\mathbf{v}) = (\exists \mathbf{v}. \, (\chi_P(\mathbf{v}) \wedge \chi_R(\mathbf{v}, \mathbf{v}')))\langle \mathbf{v}/\mathbf{v}' \rangle$$

We already know how to perform all of these operations on OBDD's. In fact, they can be performed in a single OBDD traversal if v_i and v_i' are always kept together in the variable order (called an "interleaved order").

reverse image

The reverse image of P w.r.t. R is the set of pre-states related to some post-state in P. That is, the set of states that can reach P in one transition:

$$Image(P, R^{-1}) = \{\mathbf{v} : \text{ for some } \mathbf{v}', \, \mathbf{v}' \in P \text{ and } (\mathbf{v}, \mathbf{v}') \in R\}$$

The reverse image can also be computed in terms of characteristic functions:

$$\chi_{Image(P,R^{-1})}(\mathbf{v}) = \exists \mathbf{v}. \, (\chi_P(\mathbf{v}') \wedge \chi_R(\mathbf{v}, \mathbf{v}')$$

To put all characteristic functions into our stanard representation in terms of $\mathbf{v}$, we get $\chi_P(\mathbf{v}')$ by substituting v_i' for v_i:

$$\chi_{Image(P,R^{-1})}(\mathbf{v}) = \exists \mathbf{v}.\ (\chi_P(\mathbf{v})\langle \mathbf{v}'/\mathbf{v}\rangle \wedge \chi_R(\mathbf{v}, \mathbf{v}')$$

Thus, we can compute both forward and reverse images using OBDD's.

3.4 Reachability analysis using OBDD's

By iteratively applying the forward image, we can compute the (characteristic function of) the set of reachable states.

Let P_0 stand for the set of initial states, and P_i stand for the states reachable after i transitions or fewer. Then

$$P_{i+1} = P_i \vee \text{Image}(P_i, R)$$

(Note, the χ's have been dropped...)

When $P_i = P_{i+1}$, we have reached a fixed point of this iteration, and have obtained the set of reachable states P_∞.

4 Fixed point characterization of CTL

Every formula of CTL is a state formula (is either true or false in every state). Thus, we can think of a CTL formula f as characterizing a set of states:

$$f = \{\mathbf{v} \in S :\ \mathbf{v} \models f\}$$

Given characteristic functions for p and q, we already know how to compute the following using OBDD's:

- $\neg p$, $p \vee q$, $p \wedge q$ (use corresponding operations on BDD's)
- EXp (the reverse image of p w.r.t. R)
- $AXp \equiv \neg EX \neg p$

The remaining operators of CTL can be characterized as fixed points of certain monotonic functions from sets to sets. As a result, we can compute them using OBDD's and an iterative algorithm.

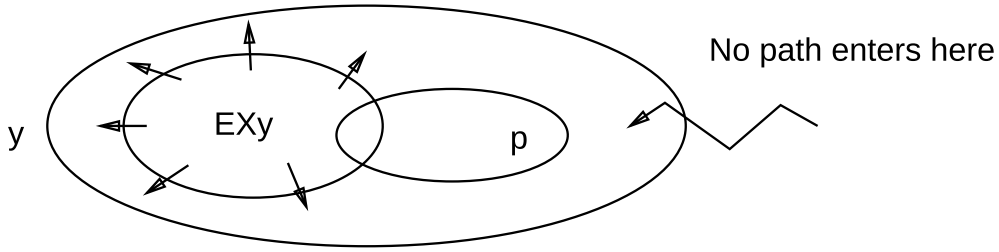
For example, we have the identity

$$EFp \equiv p \vee EX\ EFp$$

That is, there is a path leading from state s to p iff s satisfies p, or s has some successor leading to p.

In fact, EFp characterizes the least set of states y such that $y = p \vee EXy$.

The proof is simple: Suppose $y = p \vee EXy$. Then $p \subseteq y$ and $EXy \subseteq y$. Thus, there are no transitions from outide y to inside y, hence no paths from outise y to p. Thus $EFp \subseteq y$.



4.1 Fixed points of monotonic functions

Suppose $\tau : S \to S$ is any function from sets of states to sets of states. We say τ is *monotonic* when

$$x \subseteq y \text{ implies } \tau(x) \subseteq \tau(y)$$

If τ is monotonic, then it has a *least fixed point* that we denote $\mu y.\tau(y)$, and a *greatest fixed point*, that we denote $\nu y.\tau(y)$.

For example, we just proved that $EFp = \mu y.(p \vee EXy)$.

If S is finite, then the least fixed point is obtained by computing the series:

$$\text{false} \subseteq \tau(\text{false}) \subseteq \tau(\tau(\text{false})), \cdots$$

until a fixed point is reached (*i.e.*, applying τ yields no change). Note that we must reach a fixed point after at most n steps, since there is no chain of subsets of S that increases more than n times.

For example, we can compute EFp from p by the series

$$\text{false} \subseteq p \vee EX\text{false} \subseteq p \vee EX(p \vee EX\text{false}) \subseteq \cdots$$

until a fixed point is reached. We note that this is equivalent to the series

$$EF_{\leq 0}p \subseteq EF_{\leq 1}p \subseteq EF_{\leq 2}p \subseteq \cdots$$

where $EF_{\leq n}p$ is the set of states that can reach p in n transitions. This fixed point algorithm for EF can be carried out using only OBDD operations.

4.2 Characterization of EG

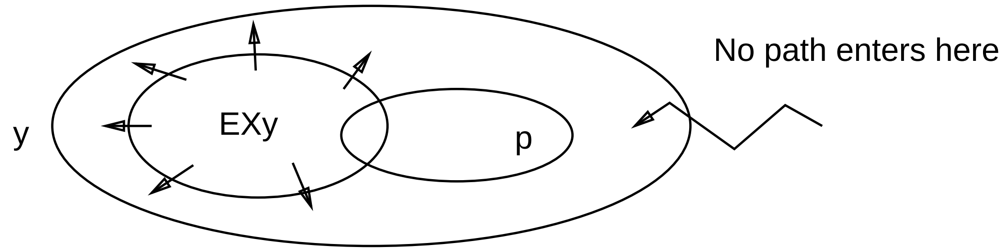
To characterize EG, we can use the equivalence

$$EGp \equiv p \wedge EX\ EGp$$

In this case, however, we have a greatest fixed point:

$$EGp = \nu y.(p \wedge EXy)$$

The proof of this is also simple. If $y = p \wedge EXy$, then $y \subseteq EXy$ and $y \subseteq p$. Thus every state in y has a successor in y, so we can build an infinite path within y from any state in y. Every state on such a path satisfies p, so $y \subseteq EGp$.



The least fixed point of a monotonic function is found by computing:

$$\text{true} \supseteq \tau(\text{true}) \supseteq \tau(\tau(\text{true})) \supseteq \cdots$$

until a fixed point is reached.

Thus, we can compute EGp from p by the (decreasing) series

$$\text{true} \supseteq p \wedge EX\text{true} \supseteq p \wedge EX(p \wedge EX\text{true}) \supseteq \cdots$$

until a fixed point is reached.

The fixed point characterizations of the remaining operators are

$$\begin{aligned}
AFp &= \mu y.(p \vee AXy) \\
AGp &= \nu y.(p \wedge AXy) \\
E(p\ U\ q) &= \mu y.(q \vee (p \wedge EXy)) \\
A(p\ U\ q) &= \mu y.(q \vee (p \wedge AXy))
\end{aligned}$$

4.3 Complexity of OBDD based model checking

- number of iterations is potentially exponential in number of state variables
- complexity of image operation is worst-case exponential
- general problem of model checking CTL when transition relation expresses as a boolean formula is PSPACE-complete

However, in practice, the number of iterations required to reach a fixed point is often small, and OBDD-based methods can be much more efficient than "explicit-state" methods.

5 The SMV system

- Very simple "Mealy machine" language
- Finite enumerated and scalar types (encoded as booleans)

- Nondetermistic choice operator for describing abstract models
- A simple mechanism for interleaving processes
- Checks CTL specs using OBDD-based algorithms

5.1 SMV language

assignments

The basic statement in SMV is the assignment. This comes in two forms:

- Zero-delay assignments

  x := expr;

  transition relation: $\chi_R :\quad (\mathrm{x} = \mathrm{expr})$

- Unit-delay assignments

  next(x) := expr;

  transition relation: $\chi_R :\quad (\mathrm{x}' = \mathrm{expr})$

All assignments in a process are *simultaneous*. Thus, the transition relation of the process is the conjunction of the relations for each component assignment.

Zero-delay dependency cycles and multiple assignments are not allowed. This (along with type checking) prevents writing systems of equations that have no solutions, *e.g.*

```
x := y;
y := !x;
```

nondeterministic choice

A list of values in set brackets is treated as a nondeterministic choice, *e.g.*,

```
send := full ? {0,1} : 0;
```

transition relation:

$$\begin{aligned} Chi_R: \quad & (\text{full} \Rightarrow (\text{send} = 0 \vee \text{send} = 1) \\ & \wedge (\neg\text{full} \Rightarrow \text{send} = 0) \end{aligned}$$

"If full, then send can be either 0 or 1, otherwise it is 0"

Deterministic assignments are treated as macros to avoid unnecessary state variables.

modules

A module encapsulates a set of program statements in a reusable, parameterized way, *e.g.*

```
MODULE bar(x,y){
    q : boolean;
    next(q) := x;
    y := !q;
}
```

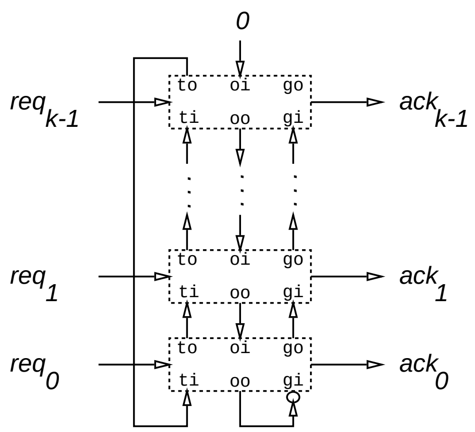
Given the above definition, the instantiation

```
foo : bar(a,b);
```

is equivalent to

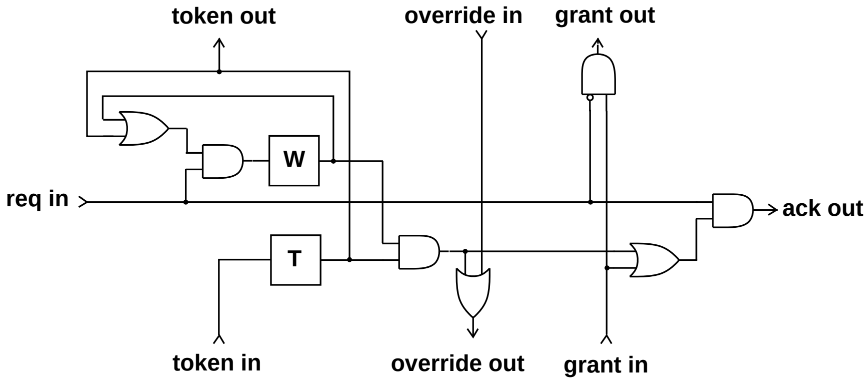
```
foo.q : boolean;
next(foo.q) := a;
b := !foo.q;
```

5.2 Example – a synchronous arbiter circuit



- At most one requester acknowledged on each clock cycle
- Priority implemented by "grant" signals
- Token moves one cell per clock cycle
- Each cell sets "waiting" bit when requesting and token
- If still requesting when token returns, asserts "override" signal

Cell implementation:




```
MODULE arbiter-element(req-in,ack-out,...)
{
  W,T : boolean;

  init(T) := init-token;
  next(T) := token-in;
  init(W) := 0;
  next(W) := req-in & (W | T);

  token-out := T;
  override-out := override-in | (W & T);
  grant-out := ~req-in & grant-in;
  ack-out := req-in & (W & T | grant-in);
}
```

The desired properties of the arbiter circuit are:

1. No two acknowledge outputs are asserted simultaneously
2. Every persistent request is eventually acknowledged

3. Acknowledge is not asserted without request

Expressed in CTL, they are:

1. $\bigwedge_{i \neq j} AG \neg(\text{ack}_i \wedge \text{ack}_j)$
2. $\bigwedge_i AGAF(\text{req}_i \Rightarrow \text{ack}_i)$
3. $\bigwedge_i AG(\text{ack}_i \Rightarrow \text{req}_i)$

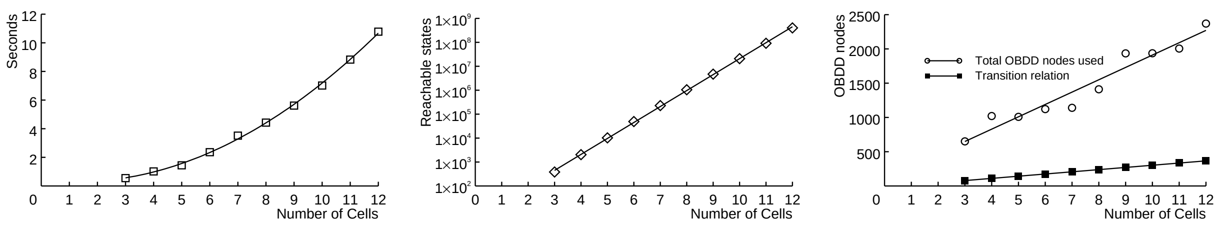
In the SMV code...

```
for(i=0;i<CELLS;i=i+1)
  SPEC
    AG((e[i].ack-out -> e[i].req-in) & AF (~e[i].req-in | e[i].ack-out));

SPEC
  AG
    for(i=0;i<CELLS;i=i+1)
      for(j=i+1;j<CELLS;j=j+1)
        ~(e[i].ack-out & e[j].ack-out);

SPEC AG ~(e[2].T  & e[2].W);
```

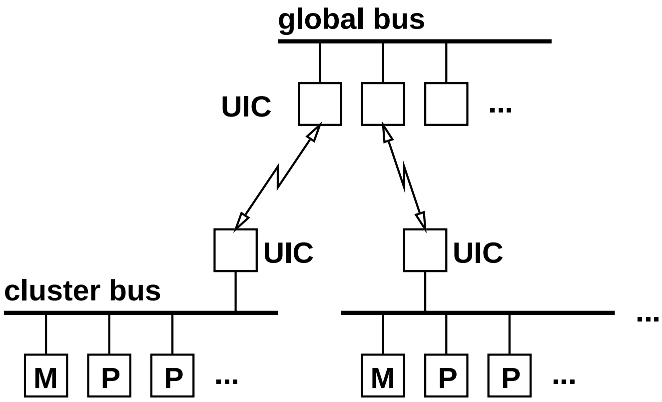
Model checker performance:



Points to note:

- Bounded width property of circuit implies linear size OBBD representation of transition relation
- Fixed point approximations are only well behaved when restricted to reachable states

5.3 Example – distributed cache coherence protocol



- distributed, shared memory multiprocessor
- processors grouped in clusters (synchronous)
- bus snooping within cluster
- consistency between clusters maintained by message passing protocol (asynchronous)
- UIC provides link between buses

Because of the complexity of the protocol, it is modeled at a very high level of abstraction:

- Only state of one memory address modeled
- Message queues abstracted to represent only messages pertaining to the given memory address
- Cache line replacement policy nondeterministic
- Bus operations considered instantaneous (not pipelined), *etc.*

Properties checked

- Absense of deadlock

  For each cache...

  ```
  SPEC
    AG(EF readable & EF writable)
  ```

Note, this property is only a heuristic check, in the sense that it requires that it is always *possible* to eventually reach a good state, but does not imply that state is inevitable reached.

However, checking this property yielded a 13-step deadlock scenario, at the end of which two "read request" messages had crossed in opposite directions on a link, and each was blocking the other from being issued because of locks being held on the two buses.

- Data coherence

  Here we use a variation on the data independence principle. We want to show that data values are always read in the same order they are written to the given memory address (note, this is not the complete requirement for the protocol, but is as much as we can express about a single cache line).

  – Assume that date values are written in increasing order.

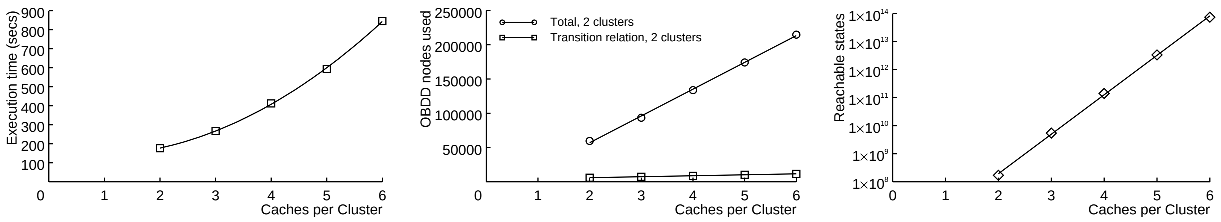
  – Use one bit to encode whether the data value is $\geq n$ for some arbitrary n.

  – Show that no value $< n$ is read by a given processor after a value $\geq n$:

$$AG[(\text{readable} \wedge \text{data} \geq n) \Rightarrow AG\neg(\text{readable} \wedge \text{data} < n)]$$

Bounded knowledge and OBDD's

Model checker performance (for checking deadlock property):



- Bounded width property of bus communication means that transition realtion scales linearly (note, separate transition relations are used for each cluster because clusters they are interleaving processes).

- Representation of reachable state set scales well because of "bounded knowledge" property of protocol – knowing state of one half of system provides bounded information about the other half of the system.

Points to remember–

- Checking "existential" properties is a useful heuristic for finding concurrency bugs
- Effectiveness of OBDD's strongly depends on communication/information structure of system
- Abstract models are a must for complex systems

6 The Mu-calculus and symbolic model checking

If we think of a formula as representing a set of states of our model, then a formula with a free variable x, can be thought of as a function from sets of states to sets of states. For example

$$p \vee EXx$$

In the propositional mu-calculus, we can express fixed points of such functions. For example

$$\mu x.(p \vee EXx)$$

6.1 propositional mu-calculus

A formula in the propositional mu-calculus is either

- An atomic proposition or propositional variable
- One of

  $$\neg p,\ p \vee q,\ p \wedge q,\ EXp,\ AXp$$

  where p and q are formulas.
- One of

  $$\mu x.p,\ \nu x.p$$

  where x is a propositional variable and p is a formula.

The formula $\mu x.p$ stands for the least fixed point of $p(x)$, while $\nu x.p$ stands for the greatest fixed point. The variable x must occur *positively* in p (under an even number of negations) to insure monotonicity.

For example, the mu-calculus formula

$$\mu x.(p \vee EXx)$$

is equivalent to EFp.

6.1.1 Simple model checking algorithm

A very simple model checking algorithm for the propositional mu-calculus derives from our previous observation that the least fixed point $\mu y.\tau(y)$ (for the finite state case) is the limit of the increasing series

$$\tau^0(\text{false}) \subseteq \tau^1(\text{false}) \subseteq \tau^1(\text{false}) \subseteq \cdots$$

That is, to compute $\mu y.p$, we first evaluate p with false for y, then iteratively plug the result back in for y in p, until a fixed point is reached.

The number of iterations of this process is at most $|S|$.

The greatest fixed point is similarly obtained from the decreasing series

$$\tau^0(\text{true}) \supseteq \tau^1(\text{true}) \supseteq \tau^1(\text{true}) \supseteq \cdots$$

Note, though, that the formula p might contain another fixed point formula $\mu y.q$ or $\nu y.q$. In this case, the inner fixed point formula will be re-evaluated up to $|S|$ times. Each re-evaluation will require up to $|S|$ evaluations of q. Thus the formula q will evaluated up to $|S|^2$ times.

In general, a subformula nested under k fixed point operators will be evaluated $|S|^k$ times. Thus, the innermost fixed point operator will require $O(|f| \times |S|^k)$ time to evaluate (since its argument is a purely propositional formula requiring at worst $O(|f| \times |S|)$ to evaluate).

6.2 Mu-calculus and CTL

Clearly, by dint of their fixed point characterizations, we can view the CTL operators as abrevations for the corresponding formulas in the propositional mu-calculus. Examples:

$$E(p\ U\ q) = \mu y.(q \vee (p \wedge EXy))$$

$$AGp = \nu y.p(\wedge AXp)$$

From now on, we will mix CTL operators and fixed point operators without comment.

6.2.1 Simple fairness constraints

Recall that for CTL fairness constraints, the only additional algorithm we need is for EGp. The rest of the operators can be converted to EU and EX, which we compute as usual, restricting to the set of "fair states" (those satisfying EGtrue). To compute EGp, we simply find the set of fair states when restricting the state space to p.

Recall also that a "simple" fairness constraint is of the form

$$\bigwedge_i GFp_i$$

For "explicit state" model checking, we find the fair states by finding all the strongly connected components. An SCC is fair if it contains, for every p_i, a state in p_i. Any state on a path leading to a fair SCC is fair.

Now, suppose y is the set of fair states. Clearly, since every state in y has a fair path, every state in y has a path through y leading to every fairness constraint in one or more steps. Thus, for all i,

$$y \subseteq y \wedge EX\ E(y\ U\ (y \wedge p_i))$$

It follows that

$$y = \bigwedge_i y \wedge EX\ E(y\ U\ (y \wedge p_i))$$

Hence the fair states are a fixed point of the above equation.

Now suppose that y is any set satisfying the above equation. We will show that y is contained in the fair states, thus the fair states are the greatest fixed point. Suppose s is a state in y. We can construct a path (of length greater than one) in y from s that visits every p_i (why?). By repeating this procedures, we obtain an infinite fair path. (Why wouldn't this work if we dropped the EX in the above equation?) Hence every state in y is fair.

As a result, the set of fair states is

$$Fair = \nu y. \bigwedge_i y \wedge EX\ E(y\ U\ (y \wedge p_i))$$

This is an abbreviation for a mu-calculus formula of alternation depth 2, since the EU subformula is a least fixed point nested within νy, and it contains a reference to y.

CTL under Streett fairness can also be expressed with mu-calculus formulas of alternation depth 2. Thus, using the fixed point algorithm, we can check CTL formulas under fairness using OBDD's.

6.3 Relational mu-calculus and symbolic model checking

In the relational (Park's) mu-calculus, we have terms that represent not just sets of states, but relations of any arity.

The notation used for introducing relations is

$$\lambda(x_1, \ldots, x_n).f$$

where $x_1, \ldots, x_n$ are variables (or in our case, vectors or boolean variables representing states), and f is the characteristic function of the relation, over the given variables.

Any term representing a relation can be *applied* to a tuple, to determine if that tuple is in the relation. For example, if R is a relation of arity 3, then

$$R(x, y, z)$$

is true when the triple (x, y, z) is in R.

Finally, the variables bound by μ and ν can now stand for relations as well as sets, and we allow existential and universal quantification over individual (in our case boolean) variables.

Note that EXp can be viewed as an abbreviation in this calculus:

$$EXp = \lambda(\mathbf{v}).\ \exists\mathbf{v}'.\ (p(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}'))$$

This extended calculus can also be model checked using characteristic functions to represent relations, and fixed point iteration. It allows us to express a number of useful properties and relations and evaluate them using OBDD's.

6.3.1 Simulation relations

Two states x and y of a Kripke structure are said to be *bisimular* if:

1. x and y agree on the atomic propositions $(a_1, \ldots, a_k)$,
2. every successor of x is bisimular to a successor of y and
3. every successor of y is bisimular to a successor of x.

The greatest bisimulation of a Kripke model (or between two Kripke models) can be computed as follows:

$$\begin{aligned}\text{Bisim} \quad = \quad & \nu Y.\ \lambda(x, y).\ \bigwedge_{1 \le i \le k} (a_i(x) \iff a_i(y)) \\ & \wedge \forall x'.\ (R(x, x') \Rightarrow \exists y'.(R(y, y') \wedge Y(x', y'))) \\ & \wedge \forall y'.\ (R(y, y') \Rightarrow \exists x'.(R(x, x') \wedge Y(x', y'))))\end{aligned}$$

The asymmetric relation of "simulation" is similar. Simulations were computed in this way for the modular CTL verification work described earlier. Moore machine equivalence can also be defined in this way.

6.3.2 Language containment

We can use mu-calculus model checking to check language containment between deterministic automata. Suppose, for example, that we want to check $\mathcal{L}M_1 \subseteq \mathcal{L}M_2$, where $M_i = (S_i, I_i, R_i, \Phi_i)$. Suppose further that the transition relation of these automata are characterized symbolically:

$$R_i = \lambda(s, \sigma, s').R'_i$$

where $s = (s_1, \ldots, s_k)$, $\sigma = (\sigma_1, \ldots, \sigma_l)$ and $s' = (s'_1, \ldots, s'_k)$. The formula R'_i is true when M_i has a transition from state s to state s' on symbol σ.

Now let $M = M_1 \times M'_2$, where M'_2 is M_2 with complemented acceptance condition. The transition relation of M is

$$R = \lambda((s, r), \sigma, (s', r')).(R_1(s, \sigma, s') \wedge R_2(r, \sigma, r'))$$

The initial states of the product are

$$I = \lambda(s, r).(I_1(s) \wedge I_2(r))$$

We now define the transition relation of a corresponding Kripke model:

$$R' = \lambda((s, r), (s', r')).\exists\sigma.R((s, r), \sigma, (s', r'))$$

We have $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$ when there is no fair path in this model under the contraint $\Phi_1 \wedge \neg\Phi_2$.

7 Summary

- OBDD's
  - canonical form for boolean functions, given variable order
  - efficient algorithms for $\wedge$, $\vee$, $\neg$, $\exists$
  - bounded circuit width properties
- Symbolic CTL model checking
  - Use OBDD's to represent characteristic functions
  - Use fixed point characterizations of CTL operators
  - Yields iterative MC algorithm that exploits
    * bounded communication width
    * bounded knowledge

- Mu-calculus
  - Expresses many useful properties and relations:
    - CTL uder fairness constraints
    - Simulation and various equivalence relations
    - Language containment
    - CTL* tableaux (not discussed here)
  - Allows these computations to be done using OBDD's

References

[1] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-91-162, Carnegie Mellon, July 1991.

[2] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite state systems.

[3] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.