Formal Verification

Yoav Rodeh

Jerusalem College of Engineering

1 Introduction

1.1 Testing vs. verification

Imagine you are building: A chip with 100K transistors, a flight controller with many processes running in parallel. Every few days the systems crashes. What to do?

Rare unexpected coincidences can happen. Like two processes running at the same time. one does $x = x + 1$, one does $x = x - 1$. With right timing in the end, $x = 1$ and not $x = 0$.

For checking correctness of a program:

1. Human testing.
2. human written automatic tests.
3. More clever - add randomness.
4. Most clever - prove correctness of your program.

This all happens because we really have unexpected bugs.

1.2 What do we want to show?

1. In a program for playing chess, we want to say that every move it makes is legal. This is very complex so we break it down to small rules:
   (a) Exactly one piece moves.
   (b) If the piece that moves is a pawn then...
   (c) if its a horse then ...
   (d) The king is not threatened after the move (of course this is again more complex)
   (e) etc.
2. Circuit for calculating sqrt, we want to say that for every input x, the output y satisfies $y * y = x$.
3. In a printer serving many computers, we want that every request made will eventually be served.

Last thing is more interesting than it seems, why? because we have a time thing going on. This is very important for computers.

First thing we have to do, is find a logic that will talk about things that change through time.

2 Linear Temporal Logic - LTL

We want to write formulas of systems that change through time. We have boolean variables like every other logic, but they can change through time. For that we consider infinite sequences of states. Let π be such a sequence : $\pi(0), \pi(1), ...$

Our formulas contain:

- variable names.
- boolean operators ($\vee, \wedge, \neg$, etc.)
- temporal operators G, F, U, X.

Each formula is evaluated over a specific point in time. so we write for example

$$\pi, 6 \models p \wedge q$$

if both p and q are true at the 6-th state of π.

What are the temporal operators (draw an example of a sequence and see where the formulas are true).

1. $\pi, i \models G(p)$ if p is true from time i on.
2. $\pi, i \models F(p)$ if p is true at some point $j \geq i$.
3. $\pi, i \models X(p)$ if p is true at $i + 1$.
4. $\pi, i \models pUq$ if p is true if there is some $j \geq i$, where q is true, and for all $i \leq k < j$, p is true.

We say that $\pi \models \phi$, if $\pi, 0 \models \phi$.

2.1 Example formulas

This was written for a variable p, but actually we can combine formulas: Show this on some specific sequence.

1. $G(Xp \vee p)$ - first see where $Xp \vee p$ is true, and then check G.
2. $F(p \wedge Xp \wedge XXp)$
3. $F(p \vee Xp)$, is actually $F(p)$.
4. $G(p) = \neg F(\neg p)$.
5. $GF(p)$ - p appears infinitely often.
6. $FG(p)$ - p is true from some point on.
7. $F(p) = trueUp$
8. $G(p) = \neg(trueU\neg p)$ - so we actually only need U and X.
9. $G(a \wedge b) = G(a) \wedge G(b)$.
10. $G(a \vee b) \neq G(a) \vee G(b)$.
11. $F(p) = p \vee XF(p)$

2.2 Mutual Exclusion

Two processes, P_1 and P_2, each loops through:

1. non-critical
2. wait
3. critical

properties:

1. safety: $G(\neg(crit_1 \wedge crit_2))$
2. liveness: $GF(crit_1) \wedge GF(crit_2)$
3. but what if they dont want to? so weaker form: $(GF(wait_1) \to GF(crit_1)) \wedge$ ...

2.3 Traffic light

It has red green and yellow.

1. $GF(green)$
2. has to be yellow in the middle $G(red \to \neg X green)$
3. order of colors can be stated in a similar way.
4. $G(red \to red U(yellow \wedge yellow U green))$ - when you are red, you'll be red until you're yellow, and then yellow until green.

2.4 elevator example

The goal of this exercise is to specify some properties of an elevator system. Assume that there is an elevator door at each door of the building with an up and a down button, and one button for each door in the elevator cabin.

1. at_i : The elevator is at the i-th floor.
2. *open* : The elevator door is open.
3. $open_i$: The door at the i-th floor is open
4. $press_i$: Someone is pressing the button for the i-th floor inside the elevator.
5. $pressUp_i$: Someone is pressing the "up" button on the i-th floor.
6. $pressDown_i$: Someone is pressing the "down" button on the i-th floor.

Describe the following properties by LTL formulae:

1. The elevator is never at the first and second floor at the same time.
2. If a button is pushed on some floor, the elevator will serve that floor.
3. A floor door is only open if the elevator is at that floor.
4. Again and again the elevator returns to the i-th floor.
5. If no button is pushed and the elevator is at the i-th floor, it will wait at that floor until a button is pushed.

2.5 More operators

1. *Weak Until*, marked pWq, means that p is true until q is true, or it just goes on forever. So is actually equivalent to:

$$pWq = (pUq) \vee Gp)$$

2. *Release*, marked pRq or sometimes pVq, means that q is true until p releases it. That is, when p becomes true, q still has to be true but right after it can be whatever it wants. This is actually:

$$pRq = \neg(\neg pU\neg q)$$

To understand the last one, lets see what is the negation of pUq:

1. Either q never occurs.
2. Or, q occurs, but before its first occurrence p fails. That is, the trace will be something like: $(p, \neg q), (p, \neg q), (\neg p, \neg q)$

To conclude:

$$\neg(pUq) = G(\neg q) \vee (\neg qU(\neg q \wedge \neg p))$$

2.6 Exercises

Formalize in LTL:

1. If p occurs at least twice, then p occurs infinitely often.

$$F(p \wedge X(F(p) \to GF(p)$$

2. p holds at all even states - this we cannot express:

$$p \wedge G(p \to XX(p))$$

   Does not hold.
3. p holds at all even states - and does not hold on all odd places:

$$p \wedge X(\neg p) \wedge G(p \leftrightarrow XX(p))$$

4. If p holds at a state s_i, then q must holds at at least one of the two states just before s_i, that is $s_{i?1}$ and $s_{i?2}$.

$$G(\neg q \vee X(\neg q) \to XX(\neg p)$$

   And we should probably think of the initial state here and fix the formula.
5. p never holds at less than two consecutive states (that is, if p holds at a state s_i, it also holds either at the state s_{i+1} or at the state s_{i-1}.

$$G(\neg p \wedge X(p) \to XX(p)) \wedge (p \to X(p))$$

3 Modeling Systems

What is our system, how can we talk about it mathematically?

Assume all our variables are boolean and examine a simple 3 bit counter:

```
init(bit0) := 0;
init(bit1) := 0;
init(bit2) := 0;

next(bit0) := !bit0;
next(bit1) := bit1 xor bit0;
next(bit2) := bit2 xor (bit0 & bit1);
```

We can write all sorts of formulas:

```
G(bit0 & bit1 & bit2 -> X(!bit0 & !bit1 & !bit2))
GF(bit0 & bit1 & bit2)
GF(!bit0 & bit1 & !bit2)
```

Thinking of calculations, this model is not very interesting - if we draw its states, its just a circle. There is one computation path only.

We add a variable that if its true, the counter jumps by 2 and not one.

```
init(bit0) := 0;
init(bit1) := 0;
init(bit2) := 0;
init(jump) := {0,1};

next(bit0) := jump ? bit0 : !bit0;
next(bit1) := jump ? !bit1 : bit1 xor bit0;
next(bit0) := bit2 xor ((jump & bit1) | (!jump & bit1 & bit2));
next(jump) := {0,1};
```

What is this variable jump? it can take whatever value it wants. Each state will have 4 values. And we actually have two initial states. Notice that graph now. Now we have many possible calculations. What formulas are possible now? the previous ones are not.

Remark: If our counter was again simple, but with a variable saying - count or don't count. Now we can get stuck constantly in the same number. A standard thing here is to add to the formulas the antecedent $GF(count)$. It is called fairness and we will discuss more of it.

3.1 Kripke Structure

Given a set of atomic propositions AP. A Kripke structure is a tuple $\langle S, I, R, L\rangle$, where

– S is the set of states.

- $I \subseteq S$ is the initial state.
- $R \subseteq S \times S$ is the transition relation. It is required that for every $s \in S$ there is some $s' \in S$ s.t. $(s, s') \in R$.
- $L : S \to 2^{AP}$ is the labeling function.

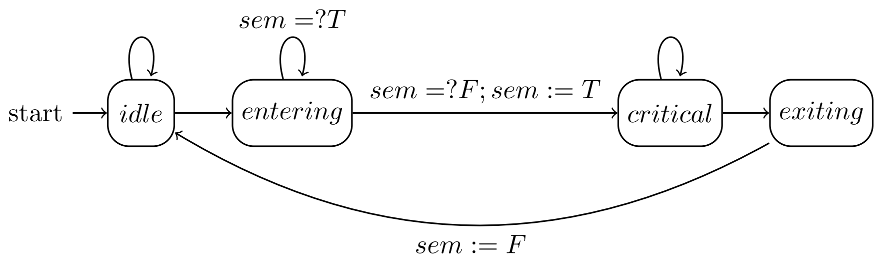
A path π in the Kripke structure is a sequence of states $s_0, s_1, ...$ where $s_0 \in I$, and for each i, $(s_i, s_{i+1}) \in R$.

We will write $K \models \phi$ if ϕ is true on all paths of K.

3.2 modeling asynchronous behaviour

Semaphore of two processes. General idea:

1. Each process is a four state variable : $\{idle, entering, critical, exiting\}$.
2. There is a boolean *semaphore*. when a process moves from *entering* to *critical* it turns it true.
3. A process cannot move to *critical* if the semaphore is true.
4. When a process leaves *exiting* it turns *semaphore* to false.
5. In *idle* and *critical* we can stay as long as we want. From *entering* and *exiting* we move when we can.



When modelling asynchronous system there is the assumption that at each moment only one of them can move. We can therefore draw the Kripke structure when we have such two or even more processes.

We can now specify LTL formulas and check them. What would be true on such a model.

3.3 enhancements

We can have non-determinism in slightly more complex ways than just having totally free variables.

```
init(x) := 1;
next(x) := x & {0,1};
```

We can also have variables that are just temporary - they don't need to appear in the state - just defines.

```
init(x) := 0;
init(y) := 0;

DEF d := x | y;
next(x) := d;
next(y) := (not d) \wedge {0,1}
```

There are slight problems here if d contains non-determinism itself:

```
DEF d := {0,1};
next(x) := d;
next(y) := d;
```

Will x and y get the same value? We would assume that yes but this has to be checked.

3.4 Exercise

lets look at the SMV tutorial.

4 Checking LTL formulas

1. Easy to check simple $G(p)$ - reachability analysis.
2. Also $F(p)$ - just see that removing p-states, there is no cycle in the graph reachable from an initial state.
3. How about $GF(p)$? same, but no cycle at all in the original reachable graph
4. $X(p)$? easy.
5. How about $G(req \rightarrow F(ack))$? Same, but no cycle reachable from a req state. That is remove all ack. now do reachability from surviving req. In this part of the graph, check there are no cycles.
6. how about $FG(p)$? see that any state in a strongly connected components is p.
7. How about $GF(p) \rightarrow GF(q)$? We're looking for a cycle with p but no q (counter example). First remove all q. and then if there is any p inside an SCC we're bad.
8. pUq? remove all q's. If there's a cycle reachable from initial state we're false. If there is not $!p$ reachable from initial state we are false.
9. $GF(p) \rightarrow G(q)$. The negation is $GF(p) \wedge F(\neg q)$. Calculate SCC's, Mark each SCC with a p and each one with a $\neg q$. We want a path to a p one that goes through a $\neg q$ one. If one has both we're done. go back from the p's and if we reach a $\neg q$ we have a counter example.
10. $GF(p) \wedge GF(q)$ We can just check both, but in a different way: For this we need a trick. We want both to run infinitely. If they do then there is also a sequence of $...p...q...p...q...p...q....$ This we can check. Add a variable $x \in \{0, 1, 2\}$:

```
next(x) := case
              x = 0 & p : 1;
              x = 1 & q : 2;
              x = 2     : 0;
            esac;
```

Now, all we want is: $GF(x = 2)$ (why wont 0 or 1 work?)

11. $GF(p) \vee GF(q)$ Just check $GF(p \vee q)$.
12. How about X? we can remember stuff and then check later. $G(p \to Xq)$ remember p with a variable p' and then just check $G(p' \to q)$

Real LTL model checking is more difficult, but this is a start.

5 Bounded Model Checking

Two links for the stuff we learn here:

1. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.6391&rep=rep1&type=pdf`
2. Another one with more details about all formulas is: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.9074&rep=rep1&type=pdf`

We have too many states to really play with the graph as we did before. We want a way to deal with these systems without actually building them.

5.1 The idea

We are given an LTL formula ϕ and a description of a structure - just the *next* and *init* statements.

Lets start writing boolean formulas and see what they mean... what satisfying assignments they have. Keep working here with an example.

```
init(x) = 0
init(y) = 0
init(z) = 0
init(e) = {0,1}

next(x) = e ? x : !z
next(y) = e ? y : x
next(z) = e ? z : y
next(e) = {0,1}
```

5.2 Init and Trans

What is this formula:

$$(x = 0) \wedge (y = 0) \wedge (z = 0) \wedge (e = 0 \vee e = 1)$$

Its easily derived from the init statements, and an assignment satisfies if it describes an initial state.

Notice how we describe a set of states here without actually writing the list down.

Now make new variables x', y', z', e' and look at the following formula:

$$(x' = e?x : \neg z) \wedge (y' = e?y : x) \wedge (z' = e?z : y) \wedge (e' = 0 \vee e' = 1)$$

An assignment requires 8 values. Which assignments make this true?

Call the first formula $Init(x, y, z, e)$, or for short $Init(V)$. Call the second one $Trans(x, y, z, e, x', y', z', e')$ or for short $Trans(V, V')$

5.3 Many possible formulas

Now how about these formulas, what assignments satisfy them?

1. $(x = 0) \wedge Trans(V, V')$
2. $Trans(V, V') \wedge (x' = 1)$
3. $Init(V) \wedge Trans(V, V')$
4. $Init(V) \wedge Trans(V, V') \wedge Trans(V', V'')$
5. $Init(V_0) \wedge Trans(V_0, V_1) \wedge ... \wedge Trans(V_{k-1}, V_k)$ Mark this one by $[M]_k$. Assignments that satisfy it mark paths of length k in the model.
6. $[M]_3 \wedge x_3 = 1$
7. $[M]_3 \wedge (x_0 = x_3) \wedge (y_0 = y_3) \wedge (z_0 = z_3) \wedge (e_0 = e_3)$ For short, mark the second part $V_3 = V_0$.
8. $$(x' = y') \wedge \bigvee_{i=3}^{i=7} ([M]_i \wedge (V_i = V'))$$

   This one is tricky - note we added extra variables : V', just for our own use.

The important point here is the size of the formulas. It is small relative to the size of the Kripke structure.

5.4 Checking simple LTL formulas

Lets say we want to check $G(p)$. A counter example would be $\neg G(p) = F(\neg p)$, which is a path that has $\neg p$ somewhere. If we have some sort of upper bound k on the length of such a path, we can write the formula:

$$[M]_k \wedge \bigvee_{i=0}^{k} \neg p_i$$

This formula is satisfiable iff there is a counter example of length at most k. The formula itself is of size $O(n \cdot k)$, where n is the size of the description of the model. Usually k is pretty small, and n is much smaller than the size of the Kripke structure.

This formula we send to a SAT-solver. That can take a long time, but today they are many times pretty good. We will hopefully discuss them later in the course.

Note we can never really prove a formula, but by increasing k we can be more and more convinced it is true.

Now how about checking $F(p)$, its negation is $G(\neg p)$. We want a path that is always $\neg p$. For this we need an infinite path. a path with a loop. So our formula will therefore be based on this one:

$$[M]_k \wedge \bigvee_{l=0}^{k-1} (V_k = V_l)$$

An assignment to it describes a path with a loop. To it we add:

$$\wedge \bigwedge_{i=0}^{k-1} \neg p_i$$

Satisfying all this mean we have an infinite path with all $\neg p$ as we wanted.

5.5 ad hoc LTL formula checking

We can try this trick for many formulas. First we negate and then write a formula that will be satisfiable if there is a path satisfying the negation.

1. checking $FG(p)$. The negation is $GF(\neg p)$. The formula is :

$$[M]_k \wedge \bigvee_{l=0}^{k-1} \left((V_k = V_l) \wedge \bigvee_{i=l}^{k-1} \neg p_i \right)$$

2. Checking $GF(p)$. The negation is $FG(\neg p)$. The formula is :

$$[M]_k \wedge \bigvee_{l=0}^{k-1} \left((V_k = V_l) \wedge \bigwedge_{i=l}^{k-1} \neg p_i \right)$$

3. Find a path satisfying pUq. no need to negate.

$$[M]_k \wedge \bigwedge_{i=0}^{k} \left(q_i \wedge \bigwedge_{j=0}^{i-1} p_i \right)$$

4. Checking pUq. The negation is $G(\neg q) \vee (\neg qU\neg p \wedge \neg q)$. We can check these separately. We just saw two formulas for this.
5. Find counter example for $G(p \to F(q))$. Negation: $\neg G(\neg p \vee F(q)) = F(p \wedge G(\neg q))$ We are looking for some p that has after it infinitely many $\neg q$'s.

$$[M]_k \wedge \bigvee_{l=0}^{k-1} \left((V_k = V_l) \wedge \bigvee_{i=0}^{k-1} \left(p_i \wedge \bigwedge_{t=i}^{l} \neg q_t \right) \wedge \bigwedge_{i=l}^{k-1} \neg q_i \right)$$

   Note that this formula is already $O(k^3)$.

6 General LTL - Bounded model checking

6.1 Prelude for the general thing

For this we need a little trick. Just for the trick, here is an example. Lets say we want to find a path satisfying $F(p \wedge q)$. We can do as usual and write:

$$[M]_k \wedge \bigvee_{i=0}^{k} p_i \wedge q_i$$

That's fine, but if this $p \wedge q$ appears many times in the formula we want to create shorthand for it and make the formula smaller. We would like a new variable satisfying $x = p \wedge q$. That is actually easy to achieve:

$$[M]_k \wedge \bigwedge_{i=0}^{k} (x_i = p_i \wedge q_i) \wedge \bigvee_{i=0}^{k} x_i$$

Now an assignment satisfying this formula will necessarily give x the correct values. if x is wrong the formula is false.

We have now actually added a line in our table:

$time$	0	1	2	3	4	5	6
p	$+$	$-$	$-$	$+$	$+$	$+$	$+$
q	$-$	$+$	$-$	$-$	$-$	$+$	$-$
x	$-$	$-$	$-$	$-$	$-$	$+$	$-$

Extending this idea, we want x to be actually $X(p)$.

$$[M]_k \wedge \bigvee_{l=0}^{k-1} \left((V_k = V_l) \wedge \bigwedge_{i=0}^{k-1} (x_i = p_{i+1}) \wedge (x_k = x_l) \right)$$

And if we want to find an example for the formula $G(p \vee X(p))$:

$$[M]_k \wedge \bigvee_{l=0}^{k-1} \left((V_k = V_l) \wedge \bigwedge_{i=0}^{k-1} (x_i = p_{i+1}) \wedge (x_k = x_l) \right) \wedge \bigwedge_{i=0}^{k-1} (p_i \vee x_i)$$

6.2 Checking all LTL formulas

1. Given an LTL formula, we first negate it. Now we write a formula that will be satisfiable iff there is a path satisfying the negation. Run with an example such as $F(pUX(p \to G(q))$
2. Our formula will always start with:

$$[M]_k \wedge \bigvee_{l=0}^{k-1} ((V_k = V_l) \wedge ...)$$

3. For each temporal operator we create a new variable.
4. We now create a formula binding the new variable to actually hold the value of its sub-formula.
   (a) for $G(q)$, variable is a. we write

$$\begin{array}{ll}\left(a_l = \bigwedge_{i=l}^{k-1} q_i\right) & \wedge \\ (a_0 = q_0 \wedge a_1) & \wedge \\ (a_1 = q_1 \wedge a_2) & \wedge ... \\ (a_{l-1} = q_{l-1} \wedge a_l) & \wedge \\ (a_{l+1} = a_l) \wedge ... \wedge (a_k = a_l) & \end{array}$$

   (b) for $X(p \to G(q))$, variable is b. we write

$$\begin{array}{ll}(b_0 = p_1 \to a_1) & \wedge \\ (b_1 = p_2 \to a_2) & \wedge \\ ... & \\ (b_{k-1} = p_k \to a_k) & \wedge \\ (b_k = b_l) & \end{array}$$

   (c) for $pUX(q \to G(q))$, variable is c. we write

$$\begin{array}{ll}(c_0 = b_0 \vee (p_0 \wedge c_1)) & \wedge \\ (c_1 = b_1 \vee (p_1 \wedge c_2)) & \wedge \\ ... & \\ (c_{k-1} = b_{k-1} \vee (p_{k-1} \wedge c_k)) & \wedge \\ (c_k = c_l) \wedge & \\ (\neg b_l \wedge \neg b_{l+1} \wedge ... \wedge \neg b_{k-1} \to \neg c_l) & \end{array}$$

   (d) And for F, the variable is d.

$$\begin{array}{ll}\left(d_l = \bigvee_{i=l}^{k-1} c_i\right) & \wedge \\ (d_0 = c_0 \vee d_1) & \wedge \\ (d_1 = c_1 \vee d_2) & \wedge ... \\ (d_{l-1} = c_{l-1} \vee d_l) & \wedge \\ (d_{l+1} = d_l) \wedge ... \wedge (d_k = d_l) & \end{array}$$

5. Now we write the final formula:

$$[M]_k \wedge \bigvee_{l=0}^{k-1} ((V_k = V_l) \wedge [a] \wedge [b] \wedge [c] \wedge [d]) \wedge d_0$$

6.3 We are missing a SAT solver

SAT solvers have improved greatly over the past years. Still, in worse case they are exponential in the number of variables. What is a trivial SAT solver? just check all assignments.

The complexity we get is related to the size of the formula and not the size of the graph. That is an important improvement.

7 Induction in bounded model checking

So far we were not able to prove any LTL formula correct. Lets try and do so by using induction. This we will only do for safety properties: $G(p)$. We want to show all reachable states satisfy $G(p)$, the induction here is clear:

1. Show initial states satisfy $G(p)$.
2. Show that if a state satisfies p then so does it successor states.

Remember at our disposal is a SAT solver, if we want to check that a boolean formula is always true, we should negate it, and give it to the SAT-solver. If it finds a satisfying assignment we are not true. From now on I just write checking a formula, but actually it needs to be negated first.

1. $Init(V_0) \to \neg p_0$
2. $p_1 \wedge Trans(V_1, V_2) \to p_2$

Example 1. A model for a 3-bit counter counting up to 5.

```
init(x) = init(y) = init(z) = 0
DEF five := x & !y & z
next(z) = five? 0 : !z
next(y) = five? 0 : z XOR y
next(x) = five? 0 : (z & y) XOR x
```

We want to prove $G(\neg(x \wedge y))$. Induction works here. Both formulas will be tautologies.

$$(x = 0) \wedge (y = 0) \wedge (z = 0) \to \neg(x \wedge y)$$
$$\neg(x_1 \wedge y_1) \wedge Trans(V_1, V_2) \to \neg(x_2 \wedge y_2)$$

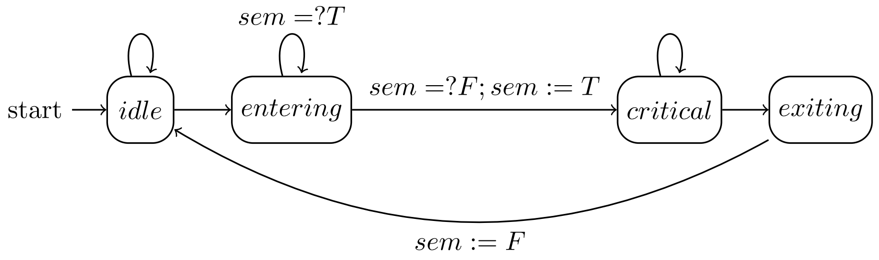
7.1 Strengthening the invariant

Sometimes what we try to prove cannot be proved by induction. Think of the case $G(x \wedge y \wedge z)$ This would not work in the second formula. Show the satisfying assignment.

For this we need to strengthen the inductive invariant. Instead of proving $G(p)$ we will prove $G(q)$, where $q \to p$. The we need to check three formulas. The two as above plus another one: $q \to p$.

In this example, q would be $x \wedge y$

Example 2. Our critical section example:



Lets write down Trans. Notice we have two state variables v, u, each can take one of 4 values, and one semaphore variable sem. At each step either one process moves or the other.

$$Trans(V,V') = \begin{pmatrix} (v = idle) \to (v' = idle \lor v' = entering) & \land \\ (v = entering \land sem) \to (v' = critical \land \neg sem') & \land \\ (v = entering \land \neg sem) \to (v' = entering) & \land \\ (v = critical) \to (v' = critical \lor v' = exiting) & \land \\ (v = exiting) \to (v' = idle \land sem') & \land \\ u = u' & \end{pmatrix} \lor (same(u \leftrightarrow v))$$

Here we would like to prove $G(\neg(u = critical \land v = critical))$ Base case works, but step doesn't. For example the state where $u = critical, v = entering, sem = 1$. We strengthen it, to

$$\neg(u = critical \land v = critical)) \land \neg((u = critical \lor v = critical) \land sem)$$

This seems right, but also won't work! Notice the state:

$$v = critical, u = exiting, sem = 1$$

It satisfies the invariant, but on the next step will violate it. I think adding:

$$\neg(v = critical \land u = exiting) \land \neg(u = critical \land v = exiting)$$

Will work.

7.2 Longer induction

For proving $G(p)$, instead of just saying p held one step ago, we can assume p a few steps back. To prove this way we check the formulas:

$$\begin{gathered} Init(V_0) \land Trans(V_0, V_1) \land ... \land Trans(V_{k-1}, V_k) \to (p_0 \land p_1 \land ... \land p_k) \\ Trans(V_0, V_1) \land ... \land Trans(V_{k-1}, V_k) \land Trans(V_k, V_{k+1}) \land p_0 \land ... \land p_k \to p_{k+1} \end{gathered}$$

On our previous example this should work on the once strengthened formula:

$$\neg(u = critical \land v = critical)) \land \neg((u = critical \lor v = critical) \land sem)$$

Using $k = 1$: looking two steps back - notice how it rules out our bad example. It will however not work on the original formula since we can always just remain in critical, so the original problem remains.

This will also help the original counter counting to 5. Wanting to show we don't reach 7, lengthening the induction will require a path of length 3 leading to 7, but such a path does not exist.

In a variant of our model where the counter goes in circles between 6 and 7 will word as well. We assume we are not in seven for the two last steps, so we cannot be in 6 either, and then it would work.

7.3 distinct states

We can force our states to be different. This does not harm the induction, and will solve our original problem. The change to the formula is simple.

Why does it not harm the induction? Draw the graph in a doughnut fashion - first one is initial states, next one is those reachable but different and so on. Induction base checks first $k-1$, next one is provable through this new induction by definition, and so on.

7.4 Last comment

Induction proves, while what we did before can only disprove. Another way of proving $G(p)$ formulas is our normal method without induction if we just knew a good k. What would be a good k? the maximal number of states needed to reach any reachable state. Can we calculate it?

We can write a formula:

$$[M]_k \wedge \bigwedge_{0 \leq i < j \leq k} V_i \neq V_j$$

If this formula is satisfiable then there is a state that needs k steps to be reached. If it is not, then all states are reachable within k steps and we can use that k to check any $G(p)$ formula.

How about formulas that need a loop? This is much more difficult, because these might actually need to repeat a state for the counter example. Think of a counter example needed an infinite number of p's and q's - a model where there a central state leading to a loop with a p and a loop with a q. We will need a counter example that repeats a state more than once.

So it depends on the formula. If we just want a counter example showing $GF(p)$, then all we need is exactly one repetition at the end (imagine a much more complex counter example and simplify it). a simple loop. For this we can just take k that is larger just by 1 than the one we found with the formula before.

For the example just before it is not so simple, because we may need to repeat many states, but we can write a formula calculating the max distance between any two reachable states and then I something like twice that distance plus k should work.

8 Circuit-SAT to SAT

For checking our formulas we needed something that takes as input a boolean formula. It would many times be helpful if it could actually get as input a boolean-circuit and say if it is satisfiable. What is a circuit? its a formula that can share sub-formulas. Quite useful for making them short.

What is usually provided is something else. Its a SAT checker. As input it gets a CNF formula: made up of clauses, each consists of literals.

8.1 Every boolean function has a CNF equivalent

This you should all know. Write the truth table. For every 0 in it, write a clause saying: we are not in this row. All these clauses give us the CNF form. Show example of $x?y : \neg z$.

This algorithm is of course exponential, and actually it must be.

8.1.1 Sometimes this must be exponential

Any CNF equivalent of $x_1 \oplus x_2 \oplus ... \oplus x_n$ will have 2^{n-1} clauses:

1. Each clause must contain all variables. If a variable is missing, then we just dropped at least one legal assignment
2. So each clause drops exactly one assignment.
3. since we have 2^{n-1} false assignments, we must have this number of clauses

So small formulas we can find exact equivalents, but are we really lost for big ones?

8.2 Transformation

Instead of finding a formula that is equivalent, we find a formula that is satisfiably equivalent. That is ϕ' is satisfiable if ϕ is.

The idea is to break down a big formula into a conjunction of very small ones - each small one can be turned into CNF, and altogether we get CNF. We do this by adding new variables.

Show an example with a small circuit: each gate gets a new variables, and we force it to be equal to the value of the gate. We also add the final clause containing the output variable.

This transformation gives a formula that is linear in the size of the original one, but adds new variables.

8.2.1 big gates

In real life we may sometimes get into small optimizations that can actually change our results a lot. This is an example of such a case.

We can be a little more efficient with the big gates. Think of $\bigvee_i x_i$. We can try adding many variables - taking it apart into binary operators, we would get $n-1$ new variables.

buy we just want one new variable a and to encode the clause:

$$\begin{aligned}
&(a = x_1 \vee x_2 \vee ... \vee x_n) &&= \\
&(\neg a \vee (x_1 \vee x_2 \vee ... \vee x_n)) \wedge (a \vee \neg(x_1 \vee x_2 \vee ... \vee x_n)) &&= \\
&(\neg a \vee x_1 \vee x_2 \vee ... \vee x_n) \wedge (a \vee (\neg x_1 \wedge \neg x_2 \wedge ... \wedge \neg x_n)) &&= \\
&(\neg a \vee x_1 \vee x_2 \vee ... \vee x_n) \wedge (a \vee \neg x_1) \wedge (a \vee \neg x_2) \wedge ... \wedge (a \vee \neg x_n)
\end{aligned}$$

and so we get just one new variable for the whole gate, and $n+1$ clauses.

9 SAT solving

Exponentially its easy. Most modern SAT-solvers rely on the DPLL algorithm.

```
boolean SAT(Clauses S)
{
  simplify(S);
  if (S is empty)
    return TRUE;
  if (S has an empty clause)
    return FALSE;
  x = selectVariable(S)
  return SAT(assign(S, x = 0) || SAT(assign(S, x = 1)))
}
```

What is assign? in every clause x appears as is remove the clause. every where it appears opposite. remove it only.

Two things are missing *simplify* and *selectLiteral*.

Show an example.

$$(p \vee q) \wedge (\neg p \vee \neg q \vee r) \wedge (r \vee q)$$

9.1 simplification

The original DPLL algorithm had two simplifications:

1. Unit clause propagation. If we have a clause with just one literal, then it must be true. then assign it, just as we've seen before. This is very simple and very powerful.
2. If a variable appears everywhere in the same polarity, remove all clauses containing it.

Example 3.

$$\begin{gathered} p \vee q \\ p \vee \neg q \\ \neg p \vee t \vee s \\ \neg p \vee \neg t \vee s \\ \neg p \vee \neg s \\ \neg p \vee s \vee \neg a \end{gathered}$$

Start by noticing that a is pure. Then split on s - and then on one side, on p. don't forget to propagate.

At each stage we have a variable we decide on, and then there are those that result from it. Show how the stack of variables looks.

9.2 How this works for 2-CNF

Let us examine how this algorithm behaves where all clauses are of size 2 (or less). We decide on a variable, and then we have the resulting assignments. This can either cause a direct contradiction, or result in a smaller formula.

Claim. In 2-CNF, If original formula is satisfiable, then if decision does not cause a direct contradiction then resulting formula is satisfiable.

Proof. If we did not reach a contradiction, then what we get now is a subset of the original clauses. This is important - it does not necessarily happen in non 2-SAT instances.

So we have two sets of clauses - those we have left, and those eliminated by our partial assignment.

Now take the values of variables

We claim that

Assume now that the original formula has □

9.3 A different algorithm

each $(a \vee b)$ can be interpreted as an implication. If we think of progogation, it is actually two implications: $\neg a \to b$ and $\neg b \to a$. We draw there as a graph with a vertex for each a and each $\neg a$.

We try to assign $\{0, 1\}$ values to vertices. Three rules suffice here for it to be a valid assignment to the formula:

1. x and $\neg x$ should always be assigned opposing values.
2. if some vertex is given 1 and there is an edge coming out of it then the vertex it goes into should also be given 1
3. If a vertex is given 0, then the vertices before it (who have edges coming into it) should also get 0.

Actually notice that we don't really need rule 3 because rule 1 and 2 give it.

So what we need is an assignment to the graph following the rules. Split the graph to SCC's (with the linear algorithm). Note that each SCC must be all 1 or all 0, because even if one vertex is 1 then all become 1 by the second rule.

This means that if some x and some $\neg x$ are in the same SCC the formula is not satisfiable. This turns out to be an "iff" condition.

Why? look at the SCC graph - it is a DAG. we want to give it values so that each SCC gets a 0 or 1 value, SCC's that hold opposing literals should get opposing values, and no 1 leads to a 0.

Notice that this graph has a following symmetry. If x leads to y, then $\neg y$ leads to $\neg x$. This means that each SCC has the dual SCC with all the opposite literals.

topologically order the graph. take the first vertex and give it 0. its dual must get 1, but if this SCC is first then it has no ion edges, and therefore the dual has no out edges. Go on like this and you'll always be fine.

9.4 selecting the split variable

This turns out to be quite important. Selecting the right one can make an exponential difference.

We would generally like to choose a variable that has the most effect.

1. Choose one randomly.
2. choose one that appears the most.
3. Better. Choose one that appears the most in small clauses.

Of course, it is very important that this heuristic does not take too much time because then it becomes the overhead itself. For example, this heuristic can be dynamic, but the bookkeeping is usually not worth it.

10 Home brewed SAT solver

We build a small and simple SAT solver. We would really like to optimize this basic algorithm. Most of the time is constant propagation.

10.1 Keep assignment only

Split steps are very expensive. because we have to remember the situation before the split and then come back to it. So in each split we should copy the whole formula. that is very bad.

A better idea is to actually not change the formula at all. but to just keep the current assignment. Then checking for unit clauses is a little different.

10.2 Literal counting

So how do we keep track of what is going on.

1. each clause will have counters, saying how many literals in it are free, how many are positive and how many negative.
2. each variable has a list of clauses where it appears positively and a list where it appears negatively.
3. when a variable is assigned we go over its list of clauses and change their counters.
4. if a clause has no positive literals, and only one free literal left - it should be propagated.

show all our basic data-structures. Note that clauses hold pointers to literals which hold pointers to singular variables.

10.3 assignment queue

how do we handle all these propagation work? we keep a queue saving all our assigned variables so far (actually literals). when we discover a new constant we put it in the end. we handle each variable at a time until the queue is completed.

10.4 back tracking

if we try a variable both ways and both eventually lead to failure we have to backtrack. but notice we have to free set variables and fix clause counts. we therefore have a special backtrack method.

10.5 improvements

1. (Exercise) change code to first handle all 1-clauses
2. (Exercise) change code so that it does not put new constants in the end if they are already in the queue. this can improve running time a lot and will cause our queue to be maximum of length n.
3. Note that the clauses are always in the list of a variable even if they become true. we would sure want to remove them so we don't have to check them every time. However, this can be a big overhead, since when backtracking we will have to return them.

11 Improvements and problems

Last chapter's solver was very basic. In the last years a few major improvements were introduced to this basic scheme.

11.1 watched literals

In our solver, note that we have to handle clauses for every variable touched, even if they are far from being interesting. We want to catch clauses when they become of size 1 or 0.

For each clause we choose two literals.

1. In the beginning they are both free. And we don't care about anything while they remain so.
2. If one of them turns true. Then clause is satisfied.
3. If one turns false, we search the clause and try to find another free literal to watch.
   (a) If we find someone who is true we move there, understanding the clause is actually satisfied.
   (b) If we find a free literal we move our watch there.
   (c) If we find no such literal - then we realise the other watched literal should now be a constant and we propagate it
4. We will find contradictions in the usual manner - in the last case, trying to set a variable, we may discover it is already set.

Example: `http://www.cs.cmu.edu/~emc/spring06/home_files/zchaff4ed.new.ppt` basically, example shown there is :

$$\begin{array}{l} v2 + v3 + v1 + v4 + v5 \\ v1 + v2 + v3 \\ v1 + v2 \\ v1 + v4 \\ v1 \end{array}$$

where in the beginning watched literals are the first two.

Here you have to show what happens with back-tracking. Setting the variables back to *free* on our queue is enough. There is no need to move watched literals back.

This saves us the complex backtracking we are used to to.

11.2 back jumping

Consider this:

$$x1, -x2, -x3$$
$$x1, -x2, x3$$
$$x1, x2, -x3$$
$$x1, x2, x3$$
$$y1, y2, ..., y100$$
$$-y1, -y2, ..., -y100$$

Its easy to see that $x1 = 0$ cannot be. But lets say variable order is $x1, y1, y2, ..., y100, x2, x3$, and we start by assigning $x1 = 0$, so we get:

$$-x2, -x3$$
$$-x2, x3$$
$$x2, -x3$$
$$x2, x3$$
$$y1, y2, ..., y100$$
$$-y1, -y2, ..., -y100$$

Now we assign $y1..y100$ and then only $x2$ which gives the problem.

How do we know where to back-jump to? we will see this next week.

11.3 Choosing decision variable

There are many heuristics for choosing a variable order. Here is a relatively simple one. It is important to notice that our running time should be fast for this one.

Each variable gets a score according to how many times it appears in clauses. Higher numbers means the variable is more important and we want to decide on it.

We keep all variables in a heap and can have fast access to the highest one easy: $O(log(n))$.

There is a slight complication we will see next week - new clauses are added at certain points. For this we need to update the counters of variables, perhaps changing the heap a bit.

A variation to make new clauses more important, is to once in a while cut scores by half.

Note this is not really a dynamic heuristic, it does not consider the present state of clauses. But it is not totally static.

11.4 random restarts

12 Learning new Clauses

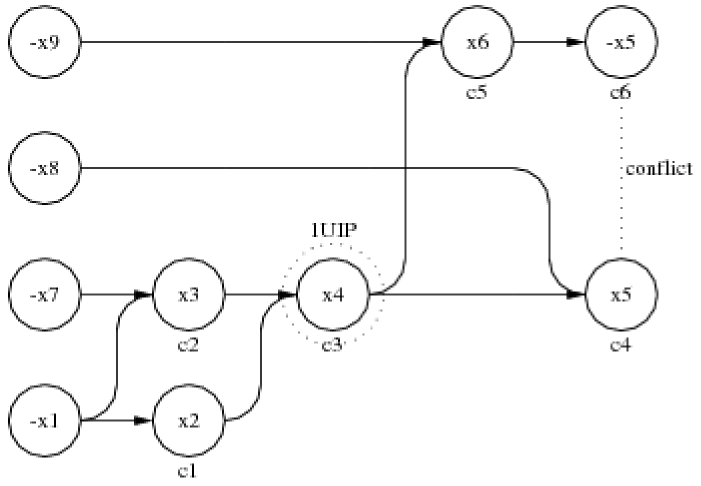
Idea is to try and learn from our mistakes. Seeing we reached a conflict, let's understand why and not repeat it. We will try and put our understanding of our conflict into a new clause, add it to our clauses, and therefore avoid searching the same area again.

12.1 Example 1

Consider the following constraints:

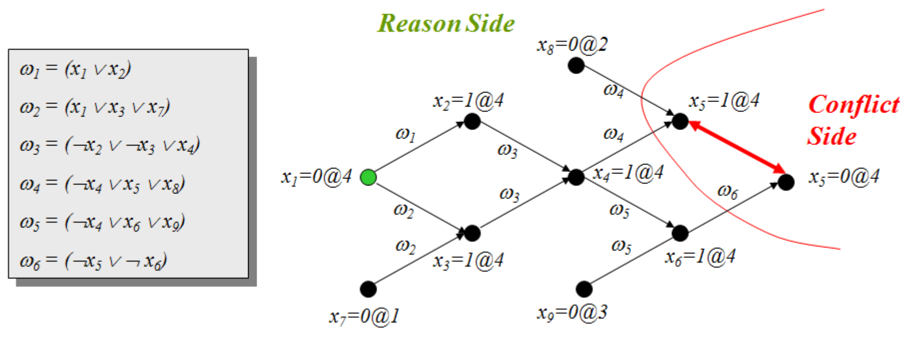
$$\begin{aligned}
c_1 &= \{x_1, x_2\} \\
c_2 &= \{x_1, x_3, x_7\} \\
c_3 &= \{-x_2, -x_3, x_4\} \\
c_4 &= \{-x_4, x_5, x_8\} \\
c_5 &= \{-x_4, x_6, x_9\} \\
c_6 &= \{-x_5, -x_6\}
\end{aligned}$$

and the following labelling choices: $-x_9$ then $-x_8$ then $-x_7$ then $-x_1$. The following implication graph leading to a conflict is constructed:



12.2 Example 2

Learning Schemes (1 UIP)


$\omega_1 = (x_1 \vee x_2)$
$\omega_2 = (x_1 \vee x_3 \vee x_7)$
$\omega_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$
$\omega_4 = (\neg x_4 \vee x_5 \vee x_8)$
$\omega_5 = (\neg x_4 \vee x_6 \vee x_9)$
$\omega_6 = (\neg x_5 \vee \neg x_6)$
Reason Side
Conflict Side
$x_8=0@2$
$x_2=1@4$
$x_5=1@4$
$x_1=0@4$
$x_4=1@4$
$x_5=0@4$
$x_3=1@4$
$x_6=1@4$
$x_7=0@1$
$x_9=0@3$
ω_1
ω_2
ω_3
ω_4
ω_5
ω_6

12.3 1-UIP

We can choose any cut to learn a new clause, as long as the conflict is on one side and the all decisions on the other.

A common heuristic is 1-UIP. We go back until the first time there is only one variable at the current decision level.

We start with the conflict and expand backwards, whenever there is a variable of higher older level we just put it in the clause. When our frontier is just with 1 variable - that's the last one.

There is always guaranteed to be such a clause - because go all the way back, and just one variable started the current level.

12.4 Back Tracking

Where would we backtrack now. We should go all the way until this clause can actually be satisfied.

Imagine we would have had our new clause from the start. When would it start propagating? We have there only one variable from this decision level (this is 1-UIP), so it would propagate at some point. This point is when all others were set. so we take the latest level of all other variables. We go back to that point. Get our new propagation and continue to see what will happen.

Examine our back jumping example and see what happens.

13 BDDs

Reduced ordered binary decision diagram - ROBDD, or for short - BDD. They are an instrument to represent binary formulas over a set of given variables. It is simply a different representation than CNF. Lets start with something simpler:

13.1 Binary Decision Trees

Lets look at a tree for $(a \wedge c) \vee \neg b$. Draw it in the natural order of variables (a, b, c).

1. Each node is labelled by a variable name.
2. Each leaf is labelled 0 or 1.
3. The order of variables is fixed (ordered).

This is actually a small automata. How can we improve it?

13.2 Minimization

identify isomorphic sub-trees and identify them. This reduces our tree considerably. Show it on the example. We see two minimization rules:

1. Have only one copy of node.
2. If a node has left and right that are the same, then don't create it, just use its children.

How can this be achieved? just take care while creating. First create the two leaves: $trueBDD$ and $falseBDD$. Keep a BDD pool, where all nodes created so far sit.

13.3 unique table

```
Class BDD {
  int level; // variable
  BDD left,right;

  static Set pool;

  public BDD(int level, BDD left, BDD right) {
    this.level = level;
    this.left = left;
    this.right = right;
    pool.add(this);
  }

  public makeNode(int level, BDD left, BDD right) {
    if (left == right)
```

```
      return left;
    BDD p = FindInPool(level, left, right);
    if (p != null)
      return p;
    return new BDD(level, left, right);
  }
}
```

How does the pool work? its a hash table, which uses the three values $level, left, right$ to calculate the hash. For example $(level+137*left+253*right)$ $mod\ tableSize$.

13.4 Creating BDD's

How would we create a BDD from formula in some other format?

```
BDD create(phi, i) {
  if (phi == 1) return oneBDD;
  if (phi == 0) return zeroBDD;
  return makeNode(i,
           create(assign(phi, 0), i+1),
           create(assign(phi, 1), i+1))
}
```

Show how this works on a simple example like $a \oplus b \oplus c$. This is basically exponential, but if we remember our results we can save time (in a temporary hash table for example).

13.5 Uniqueness

We can actually think of each node of the BDD as a formula (or boolean function). We see it as it is represented by a very specific boolean circuit, with ITE gates.

Claim. Given a variable order, for every boolean function there is only one BDD that implements it.

IT is not hard to see this by induction. for zero and one it is clear. so for one variable BDDs, and then for two by induction. and so on.

If we are working using BDDs it is very easy to see if two functions are the same. Also checking for satisfiability is trivial.

13.6 basic BDD manipulations

The real strength of BDDs is in the ability to do boolean operations on them.

A BDD representing the formula v is easy to build: *makeNode(level, trueBDD, falseBDD)* What is the BDD for $\neg v$?

How do we negate a BDD? we go through the vertices and reverse all leaves. We cant really reverse without destroying the original BDD, so we need to be slightly more careful. Easiest way:

```
BDD negateBDD(BDD n)
{
  if (n == trueBDD)
    return falseBDD;
  if (n == falseBDD)
    return trueBDD;
  return makeNode(n.level, negateBDD(n.left), negateBDD(n.right));
}
```

This works well except it is really inefficient. Think of a BDD that is not a tree. So we use dynamic programming.

```
BDD negateBDD(BDD n)
{
  if (n == trueBDD)
    return falseBDD;
  if (n == falseBDD)
    return trueBDD;
  BDD old = results.find(negateBDD, n);
  if (old != null)
    return old;
  BDD res = makeNode(n.level, negateBDD(n.left), negateBDD(n.right));
  results.add(negateBDD, n, res);
  return res;
}
```

How about assigning a value to a variable?

```
BDD assignBDD(BDD n, int level, boolean value)
{
  if (n is leaf)
    return n;
  BDD res;
  if (n.level == level)
    res = value ? n.right : n.left);
  else if (n.level < level)
    res = makeNode(n.level, assignBDD(n.left,level, value),
                            assignBDD(n.right,level, value));
  else
    res = n;
  return res;
}
```

And of course we should remember our previous results. **Exercise:** Calculate the support of a BDD. Given a BDD, print out a satisfying assignment from the BDD. Print out all.

13.7 BDD conjunction

How do we do it? assume x is the topmost variable.

$$\begin{aligned}
f \wedge g &= ite(x, f_0, f_1) \wedge ite(x, g_0, g_1) \\
&= ((\neg x \wedge f_0) \vee (x \wedge f_1)) \wedge ((\neg x \wedge g_0) \vee (x \wedge g_1)) \\
&= (\neg x \wedge f_0 \wedge g_0) \vee (x \wedge f_1 \wedge g_1) \\
&= ite(x, f_0 \wedge g_0, f_1 \wedge g_1
\end{aligned}$$

This is actually not so surprising...

```
BDD andBDD(BDD f, BDD g)
{
  if (f == falseBDD || g == falseBDD)
    return falseBDD;
  if (g == trueBDD)
    return f;
  if (f == trueBDD)
    return g;
  if (f.level < g.level)
    return makeNode(f.level, andBDD(f.left, g), andBDD(f.right, g));
  else if (g.level < f.level)
    return makeNode(g.level, andBDD(f, g.left), andBDD(f, g.right));
  else
    return makeNode(f.level, andBDD(f.left, g.left), andBDD(f.right, g.right));
}
```

We have a serious problem with running time, but it can be solved with dynamic programming as before. For each pair of nodes we save the result of their *and*. This promises us a complexity at most $O(|f| \cdot |g|)$.

13.8 General binary operators

This whole thing can work for any binary operator. Examine how the Shannon expression behaves for $\oplus$ for example.

So now, given a formula (or circuit) we can bottom up construct BDDs for it. Constants and lone variables are easy BDDs. We join them using our operations.

This is an algorithm for satisfiability - but it is generally not that good.

13.9 Quantification

This is very interesting. Examine the formula:

$$\phi(y) = \exists x(x \vee y)$$

We can do many more of these. What is really this existential quantification?

$$\exists x.\phi = \phi|_{x=0} \vee \phi|_{x=1}$$

```
BDD existBDD(BDD f, ints levels)
{
  if (f is leaf)
    return f;
  if (f.level is in levels)
    return orBDD(existBDD(f.left, levels), existBDD(f.right, levels));
  else
    return makeNode(f.level, existBDD(f.left, levels), existBDD(f.right, levels));
}
```

Of course we need a results table here. What is the complexity? not so easy this time, because we create nodes on the way, and then *or* them. If we were doing only one level, then it would be quadratic in the BDD size. So in worst case it goes very bad. Practically it is usually pretty ok, and will generally decrease BDD size.

Exercise: Write the function:

```
BDD andExists(BDD f, BDD g, int levels);
```

13.10 Using BDDs for verification

Think of the reachability procedure...

13.11 Bad and Good BDD order

Examine the function:

$$f(x_1, .., x_n, y_1, ..., y_n) = \wedge_{i=1}^{n}(x_i = y_i)$$

Think of it in the ordering $x_1, ..., x_n, y_1, ..., y_n$ where it is exponential because after you've seen all the x, you must have a separate node for each assignment. Otherwise you cannot differentiate and you don't know what y must be. So this gives a BDD with at least 2^n nodes.

What is a good ordering? the interleaving: $x_1, y_1, x_2, y_2, ..., x_n, y_n$. Draw it and see we get a BDD with something like $3n$ nodes.

There are functions that have no good variable ordering. They are always exponential. Like the middle bit of a multiplier. And many more.

13.12 How to change the BDD order of an existing BDD

You can easily change the order of two consecutive levels. Think of the general thing:

$$x?(y?f_1 : f_2) : (y?f_3 : f_4)$$

Can be written as:

$$y?(x?f_1 : f_3) : (y?f_2 : f_4)$$

The way it is done is to stop everything and change the whole BDD pool at once. This can be leveraged to all sorts of greedy algorithms. Most famous is Ruddel's which takes a variable moves it around all its options and then moves it to the minimum. Do this for all variables.

Needless to say, this takes a lot of time but is worth it.

13.13 Discussion of BDD usage and sizes

So now we have a formula handler as we wanted. We can calculate any boolean operation. Check for equality of formulas, calculate the support variables of a formula and so on.

We can for example check simple boolean formulas - are they satisfiable? Take such a formula, build a BDD for every variable (a 3 node BDD). and then calculate operations until you get the BDD for the whole thing. Then check if it is equal to *falseBDD*.

How much would this cost us? If we have n operators in the formula then it would be 3^{n+1}. Thats not so good but we really cant expect something better.

Still BDDs are usually better than this. BDD sizes are extremely important and the ordering of the variables is the best way we can try and control it, since it has a huge effect of the size.

14 Symbolic Model Checking

We see very we have too many states to start to really play with the graph. We want a way to deal with these huge graphs but handling many states at once. We we to be able to intersect them, complement, and much more.

If each state is represented by an assignment to our variables V,

So lets examine sets of assignments. How can we describe them? For our examples here $V = \{x, y, z\}$. First option is list them one by one: $\{(0,0,1),(0,0,0)\}$. This is a set of two assignments. Second option is to write a formula

1. $x \wedge (y \vee z) = \{(1,1,0),(1,0,1)\}$
2. $x \wedge y = \{(1,1,0),(1,1,1))\}$
3. z
4. $true$
5. $\exists t, x \vee t = (y \wedge z) = \{(0,0,0),(0,1,0),(0,0,1),(0,1,1),(1,1,1)\}$

Consider the last one. One way to calculate it $(x \vee 0 = (y \wedge z)) \vee (x \vee 1 = (y \wedge z)))$. Leaving it this way its quite big. But working on it we can see: $x \to y \wedge z$.

Notice how variables can actually disappear and we can get small formulas. Its not always clear which variables should really appear in the formula.

Definition 1. *We say variable $x \in V$ is in the support of formula $\phi(V)$, if there is some assignment $\alpha : V \to \{0,1\}$, such that $\phi(\alpha) \neq \phi(\alpha')$, where*

$$\alpha'(v) = \begin{cases} \alpha(v) & v \neq x \\ \neg\alpha(v) & v = x \end{cases}$$

The other way around also works, We can find a formula describing every set of assignments. start with one assignment: $\alpha = (x = 1, y = 0, z = 0)$. This is easily $\phi_\alpha(x, y, z) = x \wedge \neg y \wedge \neg z$. How about a set of n states (assignments) : $\alpha_1, \alpha_2, ..., \alpha_n$: $\phi(x, y, z) = \wedge_i \phi_i(x, y, z)$.

So we see a one-to-one correspondence between formulas on V and sets of assignments to V.

Lets look how this can help us. First we want:

14.1 A Formula Manipulator

We would like some kind of formula handler, that has:

- all basic formulas: *true*, *false*, and v for every variable we like.
- basic boolean operations on formulas: $\vee, \wedge, \neg, \oplus$, etc.
- be able to do $\exists$ and $\forall$.
- be able to check a specific assignment in a formula.
- be able to check equality of formulas. is $\phi = \psi$? This actually means it knows how to check if a formula is a tautology. we check whether $\phi = \psi$.
- be able to say what is the support of a formula.

What is a simple formula manipulator : keeps all satisfying assignments. Its easy to handle all requests but it will not be efficient.

14.2 lets do symbolic model checking

our running example:

```
VAR
  b0, b1, e: boolean;
ASSIGN
  init(b0) := 0;
  init(b1) := 0;
  next(b0) := e xor b0;
  next(b1) := b1 xor (e & b0);
```

Our variables will actually be V and another set V' with all variables tagged. For each variable, we have to formulas:

- $init_v$, a formula of the form: $v = \phi(V)$.
- $next_v$, a formula of the form: $v' = \phi(V)$.
- well, not exactly, what about non-determinism...

So what do we get:

$$\begin{aligned}
init_{b0} &:= b0 = 0 \\
init_{b1} &:= b1 = 0 \\
init_{e} &:= true \\
next_{b0} &:= b0' = e \oplus b0 \\
next_{b1} &:= b1' = b1 \oplus (e \wedge b0) \\
next_{e} &:= true
\end{aligned}$$

Now we can derive the set of initial states. it is simply the formula: $\wedge_{v \in V} init_v$. Any assignment satisfying this is an initial state. Mark by $Init$ this formula.

In our case:

$$Init = (b0 = 0) \wedge (b1 = 0)$$

See that assigning valid initial states gives true.

Remember we want R, the transition relation. Well, here it is: $\wedge_{v \in V} next_v$. Any assignment satisfying this formula means that the tagged variables represent a state who is a possible successor to the state marked by the assignment to the untagged variables. Mark by $Trans$ this formula. Notice its support can use variables both from V and V', unlike init.

So in our case:

$$Trans = (b0' = e \oplus b0) \wedge (b1' = b1 \oplus (e \wedge b0))$$

Try assigning pairs of states and see what happens.

What is the following formula?

$$\exists V, Init \wedge Trans$$

What is it in our case:

$$\exists b0, b1, e : (b0 = 0) \wedge (b1 = 0) \wedge (b0' = e \oplus b0) \wedge (b1' = b1 \oplus (e \wedge b0))$$

simplify:

$$\exists b0, b1, e : (b0 = 0) \wedge (b1 = 0) \wedge (b0' = e) \wedge (b1' = 0)$$

Now remember the support of this formula should be all with tags - we should do the quantification now. We get:

$$b' = 0$$

If we now detagify the result, and you get a formula for the set of next states. Lets do it again and see.

14.3 Reachability

How about the set of reachable states?

```
1. $R = Init$
2. while $R$ changes do
   (a) $Next' = \exists V(R \wedge Trans)$
   (b) Detagify $Next'$ to get $Next$.
   (c) $R = R \vee Next$
```

Notice we secretly used something from our formula manipulator. For checking if R changes we have to be able to compare to formulas.

There is another version too.

```
1. $R = Init$
2. $S = Init$
3. while $R$ changes do:
   (a) $S' = \exists V, S \wedge Trans$
   (b) $S = detag(S')$
   (c) $R = R \cup S$
```

14.3.1 Checking $G(p)$ formulas

.

How do we now check a formula $G(p)$. We represent p as a formula, and check if $R \wedge p \neq False$. If it is we would like find a counter example, but first lets return to our formula manipulator.

14.3.2 finding a cycle

When we want a counter example for an $AF(\neg p)$ formula, we are looking for a cycle containing only p. We look at the states satisfying $EG(p)$. It contains all sorts of strongly connected components, and trails leading too them. If we catch a state s in a strongly connected components we can use the same technique as used for AG, and find a path starting from s and ending in s - as we know it exists.

How do we remove all the trails? We start with $Y = EG(p)$, and do $Y = Y \wedge EX(Y)$. This removes the very ends of the trails. We keep going until we reach a fixed point. This is it. Take any state s from there and run:

1. $S_0 = \{s\}$
2. $S_1 = next(S_0) \wedge Y$
3. $S_2 = next(S_1) \wedge Y$
4. until some S_n contains s, where $n > 0$.
5. start from s, and pick a state in $prev(s) \wedge S_{n-1}$,
6. pick a state from there and go back, till you reach s.

14.4 Fairness

Its very useful, but $GF()$ is not CTL. We would like to add it, and ask for a given CTL formula whether is satisfied only over $GF()$ paths. This is a mixed logic. But we can support it.

We can have many fairness constraints but lets assume we just have one: $GF(q)$. Lets say our formula is $EG(p)$. So we would like to find out the set of states that satisfy $EG(p)$ under the fairness constraint (is this more or less than the original $EG(p)$ - it is actually more states).

This set Z must satisfy:

1. all states of Z satisfy p.
2. for every state $s \in Z$, there is some non-empty path leading to a $q \in Z$ state where all the path upto q is made up of p's.

But this gives an algorithm:

1. start with $Z = p$.
2. calculate $Z = Z \wedge EXE(p \; U \; Z \wedge q)$
3. continue until fixed point.

To calculate EX and EU, we just need to know who are fair states, this is $EG(true)$ using the above algorithm.

What if we have many fairness constraints? its basically the same. For the counter example computation it makes our life harder. There are heuristic solutions for it, but its not perfect.