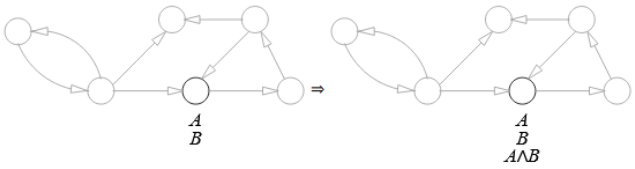
Planning and Reasoning

Gallotta Roberto

Thursday 17th December, 2020

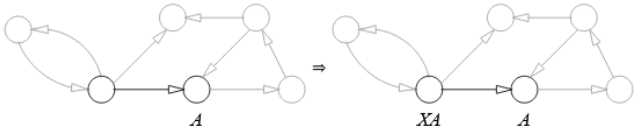
1 Labeling algorithm rules

- Propositional: $A \wedge B$ is true for all states where A is already marked and B is already marked:



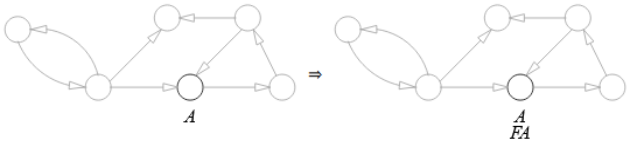
  Similarly for $\vee$ and $\neg$.

- Next: XA is true for all states that are predecessors of a state where A is true. We go backwards in one step (no multiple steps).

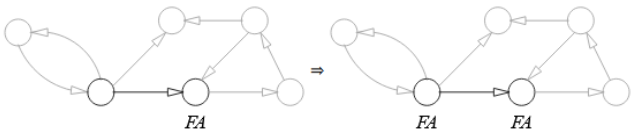


- Finally: We have the base case and the recursive case:

  1. Base case: FA is true in all nodes where A is true; this is because FA means that A is true now or in the future. This is the **only** case where we look at A.



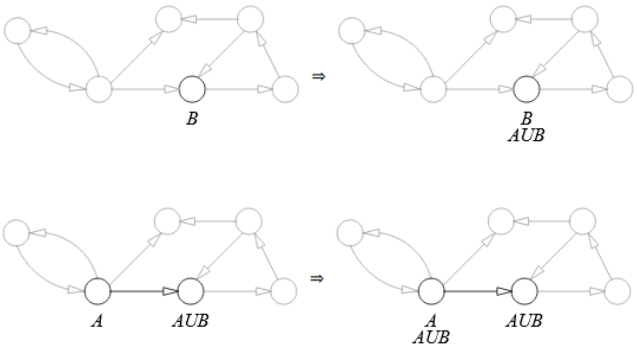
  2. Recursive case: We look at the formula itself: the precondition is FA and the consequence is that we label with FA the predecessors (from FA to FA).



- Globally: We can always reformulate Globally in terms of Finally. A formula is globally true if it's not the case that it's finally false. *skipped*

- Until: AUB; we have two cases:

  1. Base case: we look at B. If a state has B true, we label it as having AUB true. This works because the sequence of As that precedes B may be null, but AUB still would be valid.
  2. Recursive case: we don't look at B anymore. We have 2 preconditions: one is A true in the predecessor, the other is AUB true in the state. The consequence is that AUB is true in the predecessor.

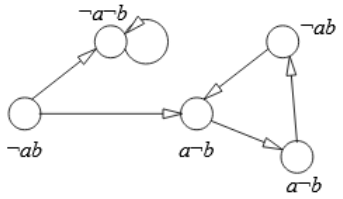


1.1 Example 1

Check the formula

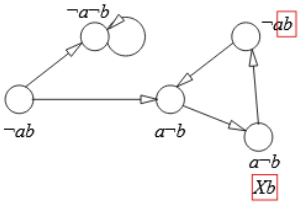
$$F(a \wedge XXb)$$

on:

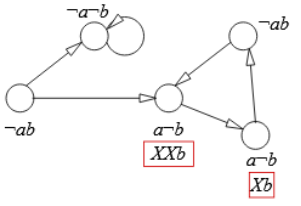


We first break the formula into its parts:

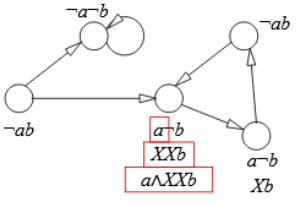
- $F(a \wedge XXb)$
- $a \wedge XXb$
- XXb
- Xb



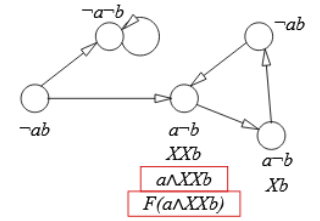
The first formula we evaluate is Xb. If we have b in a state, we go backward on the arrow and we conclude Xb in the predecessor. If the state has no predecessor (like $\neg ab$), we ignore it.



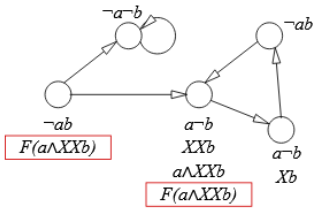
From Xb we proceed to XXb. As before, we mark the predecessor of any state that has Xb. Remember we only consider subformulas of the formula we want to prove (so, for instance, we don't care about concluding $XXXb$).



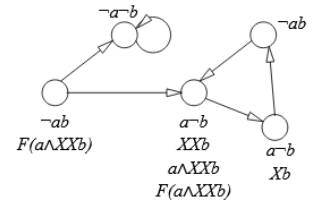
We then move to the conjunction of the two formulas. The premises are that a is true in the state and XXb is also true; the consequence is that their conjunction is true in the state.



For the Finally, we start with the base case, which is the only case where we look at the subformula. In the state where the subformula $a \wedge XXb$ is true, also $F(a \wedge XXb)$ is true.



Then we have the recursive case: we follow every arrow backwards and label the state. *Note: also the states* $\neg ab$ *and* $a\neg b$ *should be labeled with* $F(a \wedge XXb)$*!*
Remember that we always look backwards, never forward.



Since the formula $F(a \wedge XXb)$ is true in the initial state, it's true in the structure.

2 Symbolic model checking

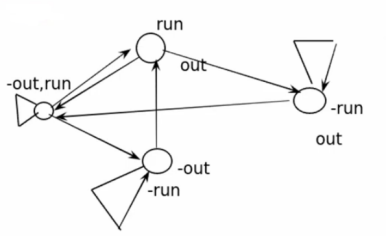
We evaluate a formula in a structure in a different way. We have a labeled graph where every mark is a subformula that we can use as a label. The semantics of the structure is not explicit. The structure is defined by a given set of variables and how they change over time.

Symbolic model checking uses CTL, *Computational Tree Logic*, which allows us to work on multiple traces at once. For example,

- AFx means that x will be true at some point in all traces
- EGx means that x is true now and forever in at least a trace

Example with NuSMV: we have 2 variables, it's basically a clock that we can freeze whenever we want, otherwise it switches between 0 and 1 (variable *out*, input *run* starts and stops it so when it's 0 the clock is frozen and when it's 1 the clock is running). We have control over run but not over out; we always move in 2 directions depending on the value of run from any state (ie: from $(\neg out, run)$ we can go to either (out, run) or $(\neg out, \neg run)$): every state has two successors.

The structure is:



The code is:

.

```
MODULE main
VAR
run: boolean;
out: boolean;
ASSIGN
init(run) := FALSE;
-- no next(run), so run can now take arbitrary values;
-- init necessary so that E = exists input sequence
-- (check is in every initial state)

init(out) := FALSE;
next(out) :=
case
!run: out;
TRUE: !out;
esac;

SPEC
AG (!run | AF out)
SPEC
EG (!out)
SPEC
AG (!out)

```

We fix the initial state.

This is the definition of next for a state; it describes how it may change.
If run is false, then out remains unchanged
, otherwise we negate it.

This are what we want to check with our program:

The output of the above code is:

```
*** This is NuSMV 2.5.4 ...

-- specification AG (!run | AF out)  is true
-- specification EG !out  is true
-- specification AG !out  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
run = FALSE
out = FALSE
-> State: 1.2 <-
run = TRUE
-> State: 1.3 <-
run = FALSE
out = TRUE

```

Note that, for formulas that evaluate to false, a trace counterexample is produced.

What is actually done is not creating a structure and then add the labels. This is because the number of states is exponential with the number of propositional variables (n variables result in 2^n states). What instead we do is use bounded model checking. We fix an horizon (number of time points) and translate the problem into propositional satisfiability and use a SAT solver.

Instead of labels we have a set of states. In every state we have a combination of the variables. We still have infinite traces since each state must always have a successor. The number of states increases exponentially with the number of variables; what we do instead is consider each label as a propositional formula over the variable (subset of states). We mark a subset of states and express the modal formula as a boolean function. Every label is a 0/1 marking of the node.

2.1 BDD

A BDD, *Binary Decision Diagram* is the same as a propositional formula which outputs True/False.

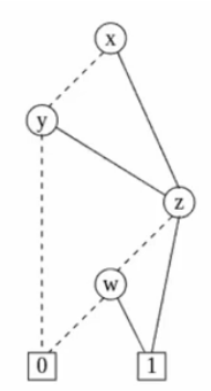
For every subformula of our original formula we will have a BDD that tells us exactly which node is labeled (given a propositional formula, it will output 1 if the node has the label and 0 if it does not). The problem is building the BDD.

Example of a BDD:

We have a root and for each node we have two children. The leaves are always 0 or 1. When the variable is 0 we follow the dashed line, when it is 1 we follow the solid line.

For example, given the evaluation

$$\{x = 0, y = 1, z = 0, w = 0\}$$



we follow the BDD and we obtain 0.

Given the interpretation

$$\{x = 0, y = 1, z = 1, w = 0\}$$

we obtain 1, ignoring in this case w.

The problem lies in the automation of building such BDD.

Additionally, note that two BDDs can be equivalent. "Equivalent" here means that for all and any propositional interpretation they give the same result. We prefer shorter BDDs due to size (the more compact, the better). Smaller BDDs usually keep related variables together (treat them as one block).

Given a propositional formula, it may be true or false. The idea of the BDD is the same: for every formula there is a BDD and viceversa. A BDD is a representation of the propositional formula if and only if for every propositional interpretation where the formula is true, the BDD produces true and similarly for propositional interpretation where the formula is false.

The principle of conversion is based on the Shannon identity:

$$F \equiv x \wedge F[True/X] \vee \neg x \wedge F[False/X]$$

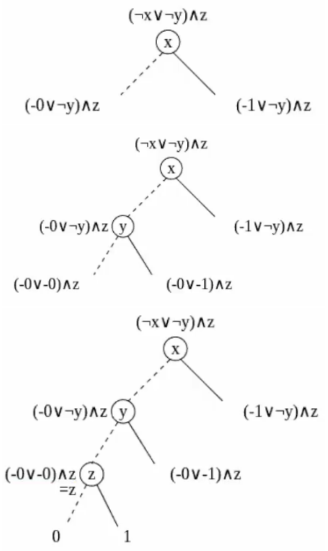
So every formula is equivalent to a conjunction where the first term is x and the second term is the formula F where every x has been replaced with $True$ and the same for the other branch. The substitution is done to reduce the number of terms in F and to simplify F.

The BDD construction is based on the principle of conversion: it's a recursive method where we start by choosing a variable and we set as successors the results of the conversion. Since the resulting subformulas are simpler than the original formula, we're guaranteed to reach either $True$ or $False$ in the end.

2.2 Example of conversion

We want to create the BDD of

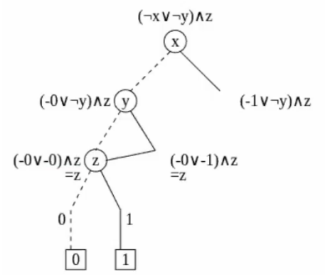
$$(\neg x \vee \neg y) \wedge z$$



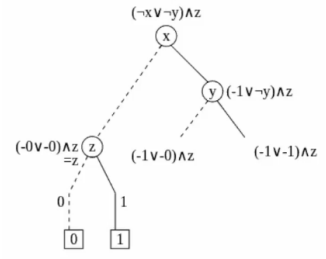
We start with the formula at the top. We choose a variable, in this case x. We substitute $True$ and $False$ in the child nodes and we simplify them.

We then select another variable, in this case y and expand the first child. Note that $(\neg False \vee \neg False) \wedge z$ is equal to z.

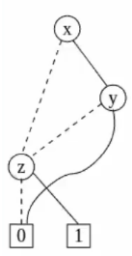
Now we only have z so we expand it and obtain the two leaf nodes. Now that this branch is complete, we move on to the others.



We note that $(\neg False \vee \neg True) \wedge z$ is also equivalent to z, so we join the two branches. We say then that y is not needed, so we can simply remove y and have the dashed line from x go directly to z.



We proceed the expansion of nodes selecting y as variable. The child when y is $False$ also evaluates to z, so it can be joined. The other child is always $False$ so we can redirect the node to $False$.



This is the final BDD.

2.3 BDD simplification

We can easily check if two formulae in the BDD are equal but the more difficult task is to check if they are equivalent (which is coNP-complete). So instead of doing that we check for equality and we also check for equality of every sub-BDD.

There are two rules for simplifying BDDs:

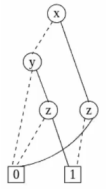
1. We merge nodes that are similar. Two nodes are similar if they behave exactly the same. Merging two nodes is done by removing one of the two and redirecting the segments that pointed to the other.
2. We remove nodes whose childs are the same.

2.4 Set value in a BDD

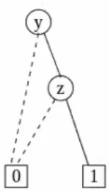
Another operation we can do on BDDs is to replace a variable in a formula with $True$ or $False$. We don't do this for temporal logic but we can do this for non-deterministic planning. We basically restrict the BDD over a value of a variable.

2.4.1 Example 1

We have the following BDD:



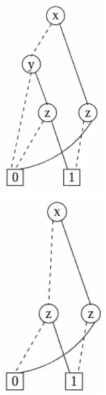
And we want to set x to $False$. We can then simply remove the branch to z from x and also remove the node x. The starting point is then y. We can then also remove the disconnected node z.



This is the resulting BDD.

2.4.2 Example 2

We have the following BDD:



And we want to set y to $True$. This is different from before since the node is not the root node (it's an internal node). We remove the node and its successors and update the branches.
After setting a variable, the BDD may be simplifiable.

This is the resulting BDD.

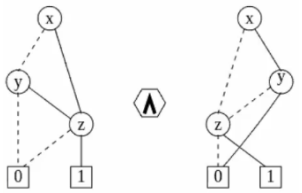
2.5 Conjunction of two BDDs

In model checking given two formula we derive their conjunction. We can do the same with BDD: given two formulas F anf G we want to obtain the BDD for $F \wedge G$. We use again the Shannon rule:

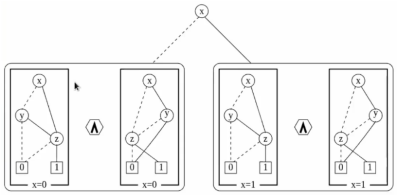
$$F \wedge G \equiv \neg x \wedge (F \wedge G)[False/x] \vee x \wedge (F \wedge G)[True/x]$$

Then we do this recursively for the subformulas.

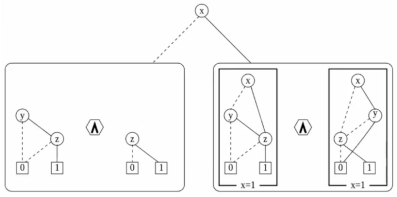
For example:



The first is the BDD for $F(...)$ and the other is the BDD for $X(...)$. We want the BDD for their conjunction.
Following the rule we have:



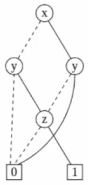
They are the original BDDs but we fix the value of the variable, so we simplify them:



We then proceed the same way, choosing a variable and simplifying the resulting BDDs. We do this until we reach only $True$ or $False$ (this is the base case of recursion) and we evaluate the current BDD.

We can speed the process up by storing each combination of BDDs we encounter and evaluate and replace them directly if they show up again.

In the end we obtain this:



2.6 Quantifiers on a BDD

This is used in temporal logic. We want the BDD equivalent to $\exists x.F$, so we want to have the BDD to be $True$ when its successor is a BDD that is also $True$. This is like applying $\exists x$ on the BDD.

We simply set the value and evaluate: $F[False/x] \vee F[True/x]$. The process is the same as before (for the conjunction of two BDDs).