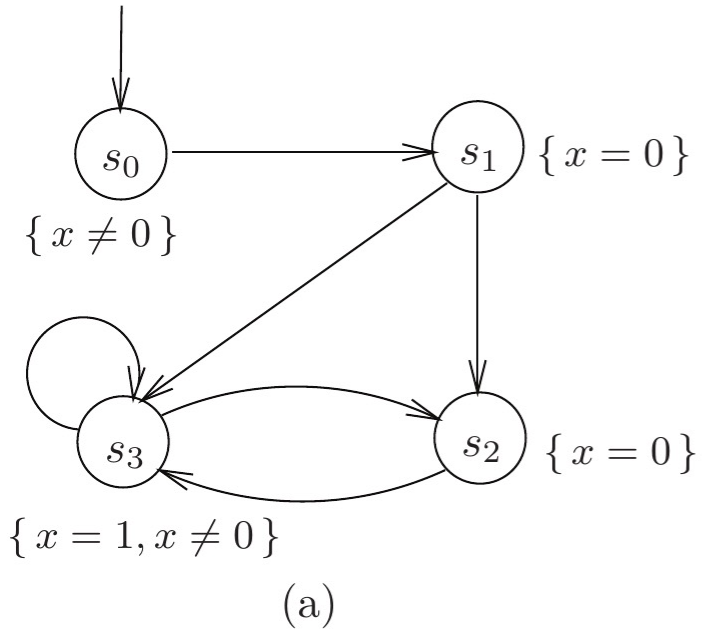
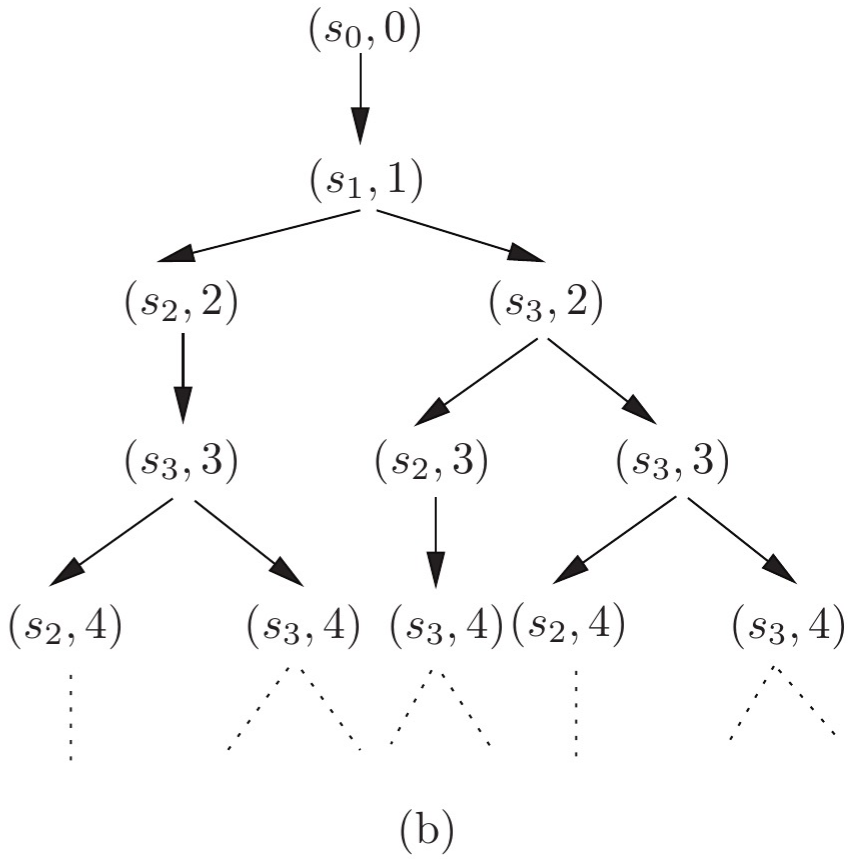
Computation Tree Logic

Branching Time Logics

Computation Tree

- Often we are interested in expressing properties of the *computation tree* of all executions of a transition system





Informal examples of properties

- "All executions starting from state *init* are such that state *critical* is never entered"
- "If the system enters state *error*, there is a way for the system to get back to the *working* state"
- "The system cannot make a transition from state *normal* to state *critical*"
- "There is no execution that, from the initial state of the system, reaches state *critical* without first entering state *warning*"

What about LTL?

- Interpretation of a LTL formula is in terms of *paths (traces)*.
- A LTL formula Φ holds in a state s for a transition system if *all possible computations* starting in s satisfy Φ
- In LTL it is not easy to talk about «some computations», «possibility»
- For instance, «for every computation it should be *possible* to return to the initial state», (i.e., …there is a computation going back to the initial state) cannot be expressed in LTL
  - *GF start*? NO, this requires that all computations go back to start, not that they *can* go back even if they might not actually go back!

Branching Time Logics

- In Branching-time TL, the underlying structure of time is assumed to have a branching tree-like nature where each moment may have many successor moments.
- Typically used to talk of “all/some possible futures”
- We will consider only the «simplest» logics, *CTL*
- **Computation Tree Logic**

CTL: State formulae and Path Formulae

- CTL formulae are of two types.
- *State formulae* are assertions about the atomic propositions in the states and their branching structure
  - **Eφ** means “there exists a path in the computation tree from the current state such that φ holds on the path; dually for **Aφ**.(“for all paths from the current state φ holds on the path”
- *Path formulae* express temporal properties of paths.
  - They are similar to LTL formulae, but “simpler”: as in LTL they are built by the *next-ste*p and *until* operators, but they cannot be combined with Boolean connectives and no direct nesting of *temporal* modalities is allowed.

CTL syntax

- *State* formulae:

$$\Phi ::= \text{true} \mid a \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid E\varphi \mid A\varphi$$

  - where *a* is a propositional letter and φ is a *path* formula
- *Path* formulae:

$$\varphi ::= X\,\Phi \mid \Phi_1\, U\, \Phi_2$$

  - where Φ is a state formula
- with the usual abbreviations $\vee$, $\Rightarrow$,…
  - Hence, you cannot nest directly two temporal operators without using an E or A operator.
  - These restrictions are introduced to make CTL easy to verify.

Syntax in practice

- State and temporal operators are always combined in pairs:
- EX Φ, AX Φ, E(Φ_1 U Φ_2), A(Φ_1 U Φ_2)

- Very common shorthand:
- EF Φ is E(true U Φ) AF Φ is A(true U Φ)
- EG Φ is $\neg$AF$\neg\Phi$ AG Φ is $\neg$EF$\neg\Phi$

- NB: Often $\exists$ for E, $\forall$ for A; $\bigcirc$ for X, $\Diamond$ for F, $\Box$ for G

Informal semantics

- Path operators E and A “choose the path”
  - Eφ = "there is a path from the current state along which φ holds"
  - Aφ = "along all paths from the current state φ holds"
- Temporal operators talk of the “chosen” (*current*), path
  - XΦ = "in the next state along the current path Φ holds"
  - FΦ = "there is a state along the current path in which Φ holds"
  - GΦ = "in all states along the current path Φ holds“
  - $\Phi_1 U \Phi_2$ = "there is a state along the current path in which Φ_2 holds, and Φ_1 holds from the current state until then"

Examples of formulae

$init \Rightarrow AG(\neg critical)$ If current state is *init,* state *critical* is never entered

$error \Rightarrow EF(working)$ If current state is *error*, we can get back to *working*

$normal \Rightarrow AX(\neg critical)$ *critical* cannot be entered immediately if current state is *normal*

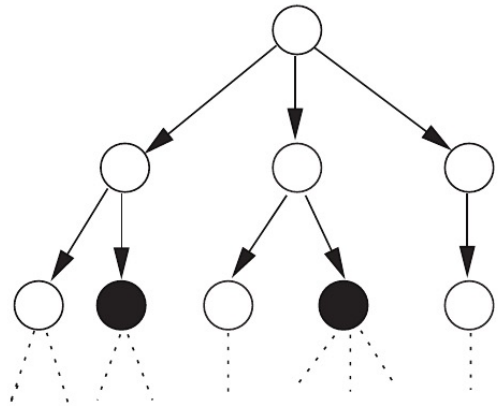
$\neg E(\neg working\ U\ critical)$ It is not possible that critical is entered after an interval when the system is not working

$AG(req \Rightarrow EF\ grant)$ After a *request* it is possible to have it *granted*

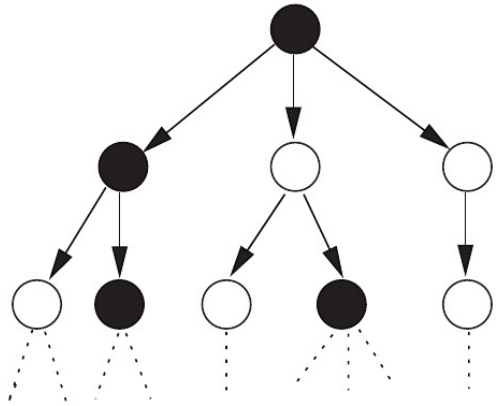
$EG(req \Rightarrow AF\ grant)$ It is possible that every *request* will be always *granted*

$AG(req \Rightarrow AF\ grant)$ (It is necessary that) every *request* will be always *granted*

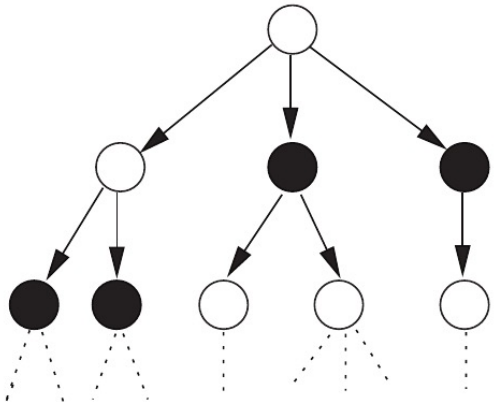
Ex: computation trees and formulae (evaluated at the root)



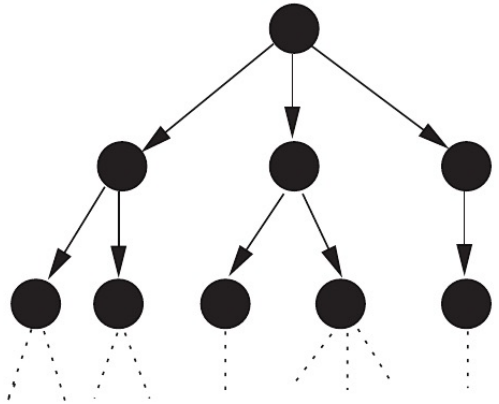
$\exists \Diamond \, black$



$\exists \Box \, black$

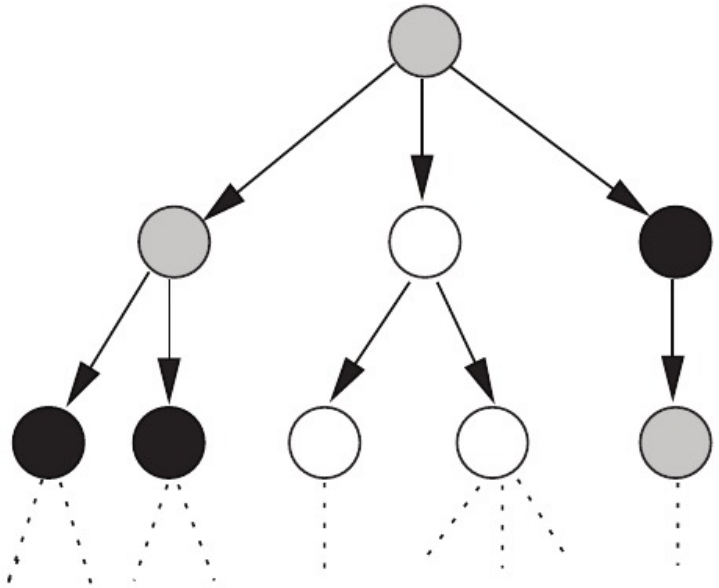


$\forall \Diamond \, black$

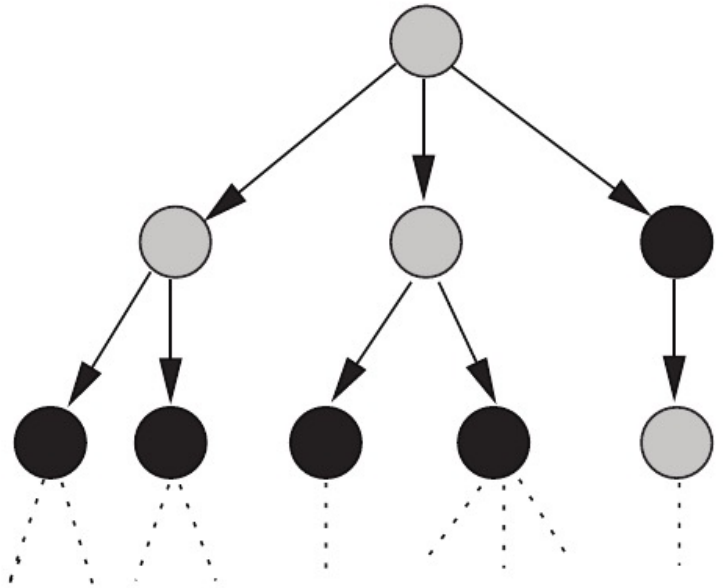


$\forall \Box \, black$

Until



$\exists(gray \,\mathsf{U}\, black)$



$\forall(gray \,\mathsf{U}\, black)$

CTL semantics over a TS = $\langle S, Act, \rightarrow, I, AP, L \rangle$

- For *state* formulae, given $s \in S$:
  - $s \models a$ iff $a \in L(s)$
  - $s \models \neg\Phi$ iff it is not the case that $s \models \Phi$
  - $s \models \Phi_1 \wedge \Phi_2$ iff both $s \models \Phi_1$ and $s \models \Phi_1$ hold
  - $s \models E\varphi$ iff $\sigma \models \varphi$ for some path σ starting from state s
  - $s \models A\varphi$ iff $\sigma \models \varphi$ for all paths starting from state s

- For *path* formulae, given a path σ (i.e., a sequence of states $s_0 s_1 s_2 \ldots$ of a run) in TS
  - $\sigma \models X\Phi$ iff $s_1 \models \Phi$
  - $\sigma \models \Phi_1 \, U \, \Phi_2$ iff there exists $j \geq 0$ such that $s_j \models \Phi_2$ and for all i such that $0 \leq i < j$ $s_i \models \Phi_1$

- TS $\models \Phi$ iff for all $s_0 \in I$ we have $s_0 \models \Phi$

CTL can be hard to understand!

- CTL is infamous for being difficult to interpret
  - a designer may fail to understand what property has been actually verified!
  - It is important to be familiar with the most common constructs of CTL used in verification

A typical CTL pattern

1. *AG(req $\Rightarrow$ AF good)*

- For all reachable states (AG), if *req* is true in the state, then always at some later point (AF) we must reach a state where good is true.
  - Hence, it is always the case that if *req* is true, then eventually *good* will also be true.
  - Helpful to express *total correctness* (termination eventually occurs with correct answers), *accessibility* (eventually a requesting process will enter its critical section), *starvation freedom* (eventually service will be granted to a waiting processor).
  - In this formula, AG is interpreted relative to the initial states of the system. AF is interpreted relative to every state where req holds.
  - A common mistake is to write (1) as (1') req $\Rightarrow$ AF ack. The meaning of (1') is that if req is true in the initial state, then it is always the case that eventually we reach a state where ack is true. (1) requires that the condition is true for any reachable state where req holds.

Other patterns

- AG AF enabled
  - From every reachable state, for all paths starting at that state we must reach another state where enable is asserted. In other words, enabled must be asserted infinitely often. It can be used, for instance, to enforce a reset condition from any state
- **AF AG good**
  - eventually the system is confined to states where good is always true, i.e., ``the system stays out of states where good is true only a finite number of times". It can be used to specify the property of finite number of failures in the system.
  - *This is an example of a CTL property that cannot be represented by a Buchi automaton (hence not even in LTL)*
- **AG EF restart**
  - From any reachable state, there must exist a path starting at that state that reaches a state where restart is asserted. In other words, it must always be possible to reach the restart state. It can detect, for instance, the absence of deadlocks, by requiring that is it always possible to reach deadlock-free states.
  - *This is another example of a CTL property that cannot be represented by a Buchi automaton*
- **EF(started $\wedge$ $\neg$ready)**
  - It is *possible* to get to a state where started holds, but ready does not hold.
  - *This is another example of a CTL property that cannot be represented by a Buchi automaton.*

Harder examples

- LTL: XFp e FXp are equivalent: «P holds sometimes in the strict future»
- AF AXp and AX AFp are NOT equivalent
  - AX AF p asserts that «on all paths, p holds sometimes in the strict future»
  - AF AX p ?
    - eventually all paths reach a state where p holds in each of its next states.

Comparison of LTL and CTL

- Def.: A CTL formula Φ and an LTL formula φ (both over AP) are equivalent, denoted $\Phi \equiv \varphi$, if for all transition systems TS over AP:

$$TS \models \Phi \text{ if and only if } TS \models \varphi.$$

- Theorem: ***CTL and LTL are incomparable***

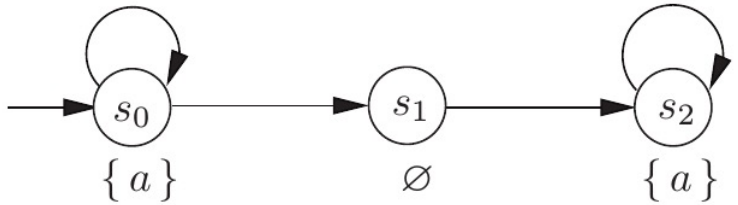
There exist LTL formulae for which no equivalentl CTL formula exists:

FG a, F(a $\wedge$ Xa) … $\Diamond\Box a$ $\Diamond(a \wedge \bigcirc a)$

There exist CTL formulae for which no equivalentl LTL formula exists:

AFAG a, AGEF a, AF(a $\wedge$ AXa) …

Lemma: *AFAGa* and *FGa* are not equivalent



LTL formula *FGa* ($\Diamond\Box a$) holds in the initial state s_0, since each path starting in s_0 eventually remains forever in one of the two states s_0 or s_2 , which are both labeled with a.

CTL formula *AFAGa* does not hold in s_0, because the path s_0^ω (staying in s_0 forever) does not satisfy it: in fact, there is a path out of s_0 going through ¬a-state s_1. Thus, s_0^ω is a path starting in s_0 which will never reach a state satisfying *AGa*.
Therefore, s_0 does not satisfy *AFAGa*

Criterion for Transforming CTL Formulae into Equivalent LTL Formulae

- **Theorem**: Let Φ be a CTL formula, and φ the LTL formula that is obtained by eliminating all path quantifiers in Φ. Then:
  $\Phi \equiv \varphi$ *or there does not exist any LTL formula that is equivalent to CTL formula* Φ*.*
  - For instance, AGAFa is equivalent to GFa
- However, since AFAGa is not equivalent to FGa, it follows that:
  - there is no LTL equivalent of AFAGa
  - (but also there is no CTL equivalent of FGa)

Fairness?

- ***Fairness Constraints*** are often necessary to verify concurrent systems
  - For example, if the system allocates a shared resource among several users, only those paths along which no user keeps the resource forever should be considered.
- A *fair path* is a path along which each fairness condition is satisfied infinitely often.
- CTL cannot express fairness conditions with path formulae, hence by itself cannot express assertions about *correctness along fair paths*.
  - "for all paths" should be replaced by "for all fair paths" and similar for there exists a path
  - Need **A(fair ==> F)**, which is not allowed in CTL
- Fair CTL is a modification of CTL, with the same syntax but whose semantics is defined *only over fair paths*.
  - fair paths are defined by a CTL formula interpreted "as if it is LTL"

CTL MODEL CHECKING

Normal Forms

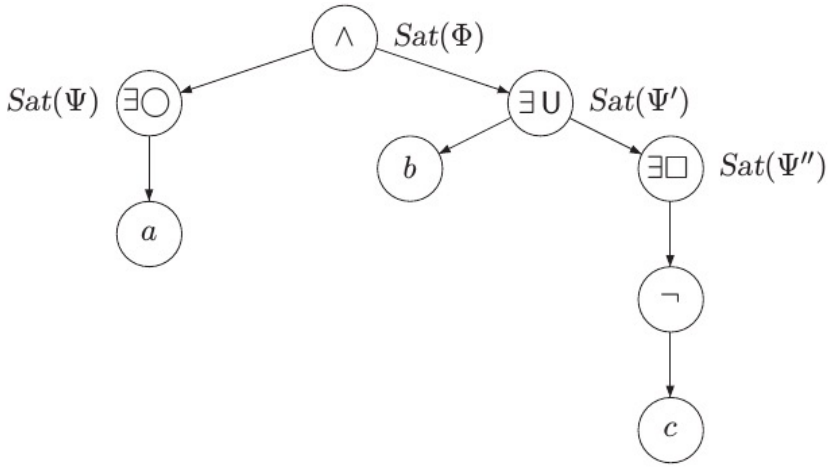
- Positive Normal Forms can be defined analogously to LTL.
- Introducing Weak Until, «W», and Release «R»
- It is useful to define also an existential normal form (only modalities EX, EU, EG)
  - negation cannot be pushed to the atoms in this case
  - but model checking algorithms usually handle also AX, AU and AG
- We skip details

Model Checking CTL formulae

- The problem: given a transition system TS and a CTL formula Φ, does TS $\models \Phi$?
- The idea: for each state subformula ψ of Φ, explore the state space of TS and determine the set of states in which ψ holds
- We start from atomic propositions (with nesting depth of temporal operators 0), then work our way up examining subformulae of Φ of ever increasing nesting depth, until we get to Φ
  - of course Φ has the greatest nesting depth among all its subformulae
- We assume formulae in *existential normal form*
  - the only admissible form of temporal subformulae is: $E(\varphi_1 U \varphi_2)$, $EX\varphi$ and $EG\varphi$

Example: a formula and its syntax tree

$$\Phi = \underbrace{\exists \bigcirc a}_{\Psi} \wedge \underbrace{\exists (b \,\mathsf{U}\, \underbrace{\exists \Box \neg c}_{\Psi''})}_{\Psi'} .$$



The nodes of the parse tree represent the subformulae of Φ.
The leaves stand for the constant true or an atomic proposition a $\in$ AP.
All inner nodes are labeled with an operator.

The MC algorithm is a bottom-up traversal of the parse tree of the CTL state formula Φ, computing for each node the set of states satisfying the formula at the node.

Rules to compute set Sat(Φ) of states satisfying a formula Φ in ENF

(a) $Sat(true) = S$,

(b) $Sat(a) = \{ s \in S \mid a \in L(s) \}$, *for any* $a \in AP$,

(c) $Sat(\Phi \wedge \Psi) = Sat(\Phi) \cap Sat(\Psi)$,

(d) $Sat(\neg\Phi) = S \setminus Sat(\Phi)$,

(e) $Sat(\exists \bigcirc \Phi) = \{ s \in S \mid Post(s) \cap Sat(\Phi) \neq \varnothing \}$,

(f) $Sat(\exists(\Phi \cup \Psi))$ *is the smallest subset* T *of* S, *such that*

(1) $Sat(\Psi) \subseteq T$ *and* (2) $s \in Sat(\Phi)$ *and* $Post(s) \cap T \neq \varnothing$ *implies* $s \in T$,

(g) $Sat(\exists \Box \Phi)$ *is the largest subset* T *of* S, *such that*

(3) $T \subseteq Sat(\Phi)$ *and* (4) $s \in T$ *implies* $Post(s) \cap T \neq \varnothing$.

The complete algorithm (from BK08)

Algorithm 14 Computation of the satisfaction sets

Input: finite transition system TS with state set S and CTL formula Φ in ENF
Output: $Sat(\Phi) = \{ s \in S \mid s \models \Phi \}$

(* recursive computation of the sets $Sat(\Psi)$ for all subformulae Ψ of Φ *)

switch(Φ):

true	:	**return** S;
a	:	**return** $\{ s \in S \mid a \in L(s) \}$;
$\Phi_1 \wedge \Phi_2$	:	**return** $Sat(\Phi_1) \cap Sat(\Phi_2)$;
$\neg\Psi$	:	**return** $S \setminus Sat(\Psi)$;
$\exists\bigcirc \Psi$	:	**return** $\{ s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing \}$;

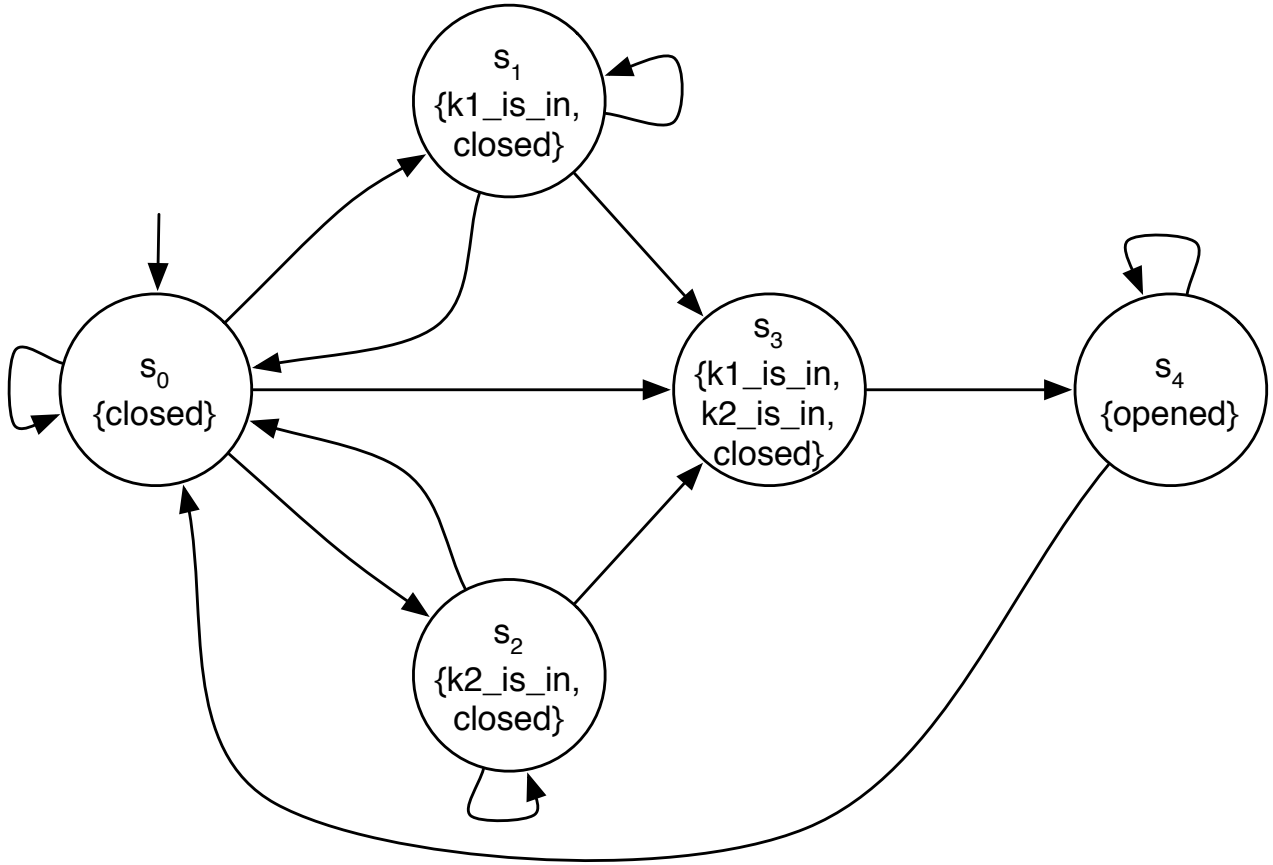
$\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$	:	$T := Sat(\Phi_2)$; (* compute the smallest fixed point *)
		while $\{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \} \neq \varnothing$ **do**
		  let $s \in \{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \}$;
		  $T := T \cup \{ s \}$;
		od;
		return T;

$\exists\Box \Phi$	:	$T := Sat(\Phi)$; (* compute the greatest fixed point *)
		while $\{ s \in T \mid Post(s) \cap T = \varnothing \} \neq \varnothing$ **do**
		  let $s \in \{ s \in T \mid Post(s) \cap T = \varnothing \}$;
		  $T := T \setminus \{ s \}$;
		od;
		return T;

end switch

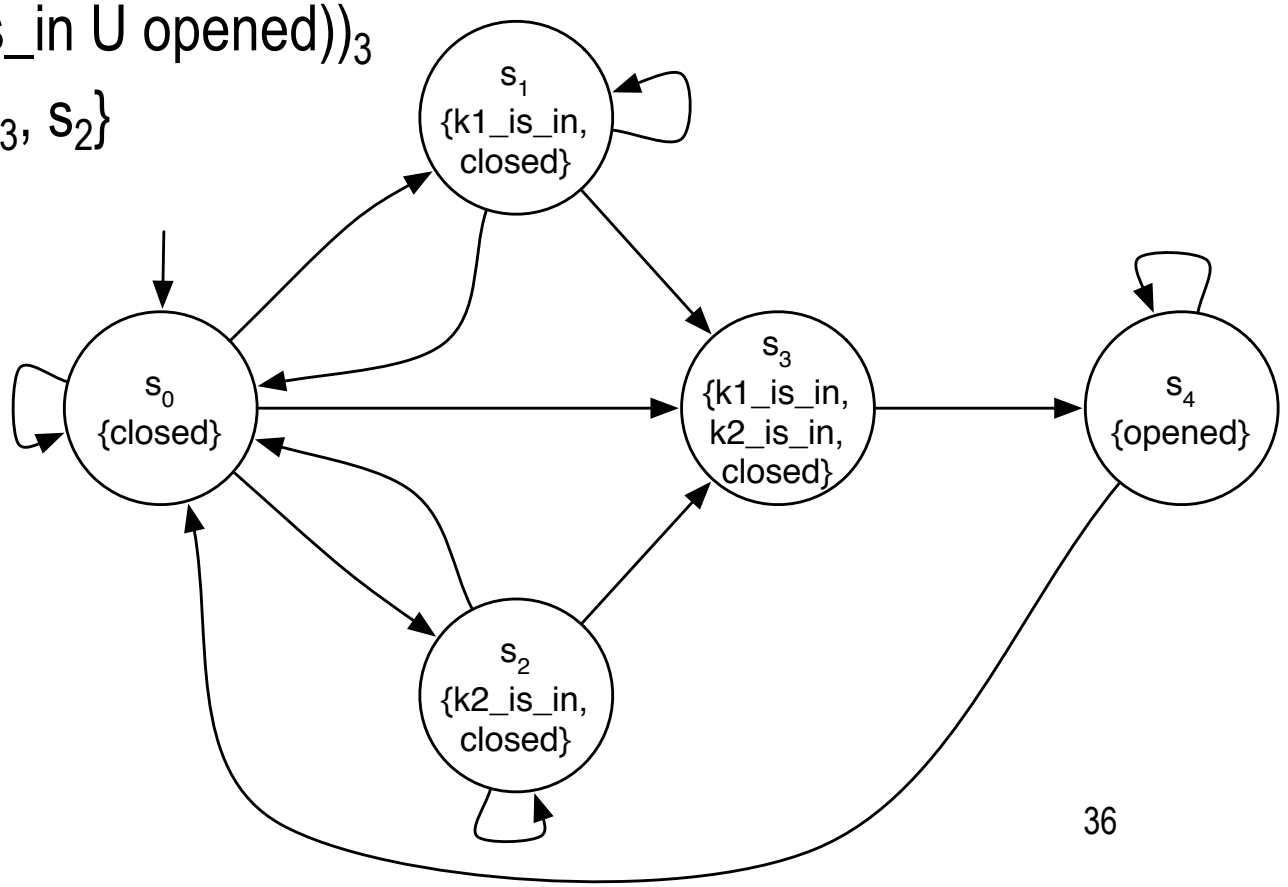
An example

TS =

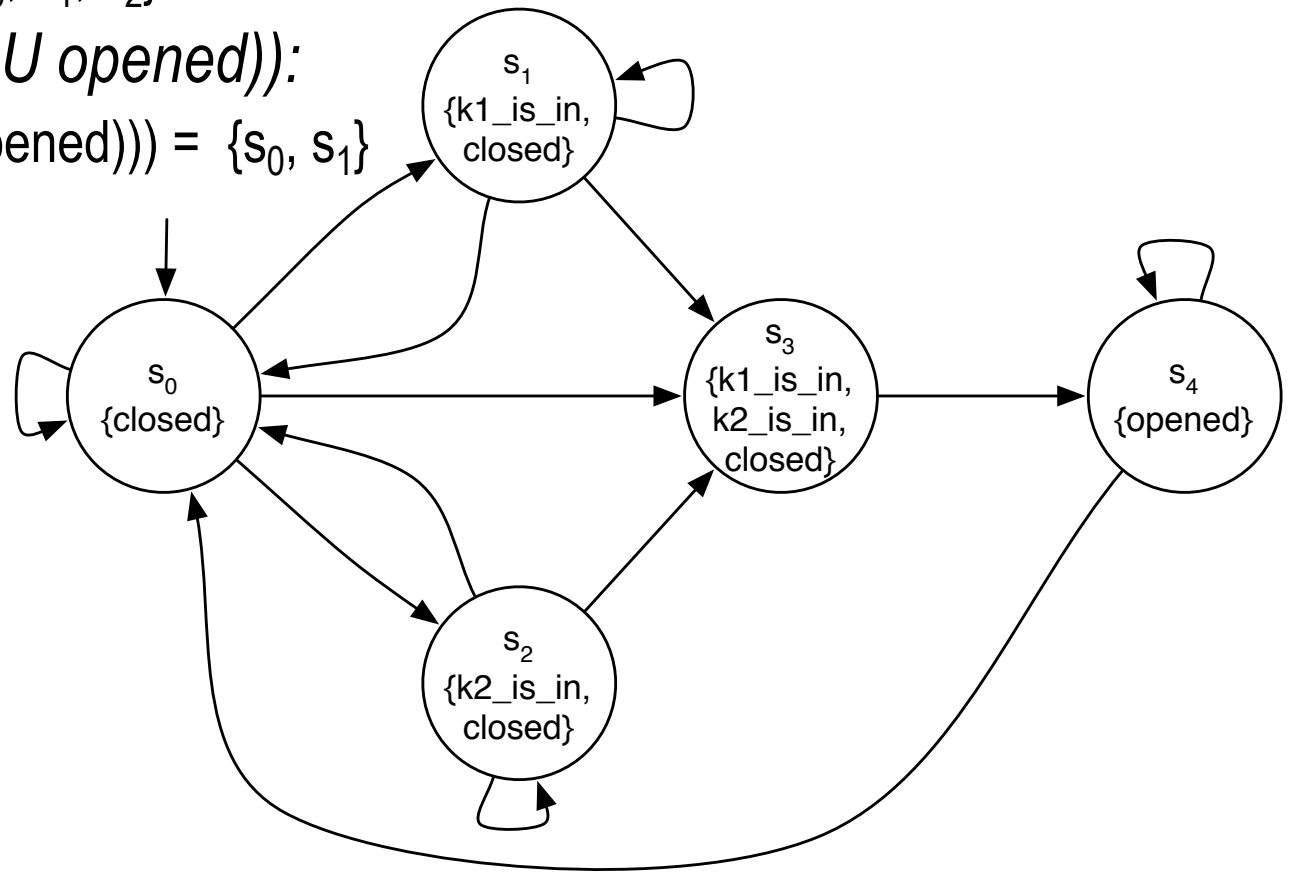


$$P = EF(\neg k2\_is\_in \wedge EX(k2\_is\_in \wedge E(k2\_is\_in\ U\ opened)))$$

- Consider subformula *E(k2_is_in U opened);* we have the following iterations:
  - *Sat*(E(k2_is_in U opened))$_1$ = {s_4}
    Sat(E(k2_is_in U opened))$_2$ = {s_4, s_3}
    Sat(E(k2_is_in U opened))$_3$ = {s_4, s_3, s_2}
    Sat(E(k2_is_in U opened))$_4$ = {s_4, s_3, s_2} =
    Sat(E(k2_is_in U opened))$_3$
  - Hence *Sat*(E(k2_is_in U opened)) = {s_4, s_3, s_2}

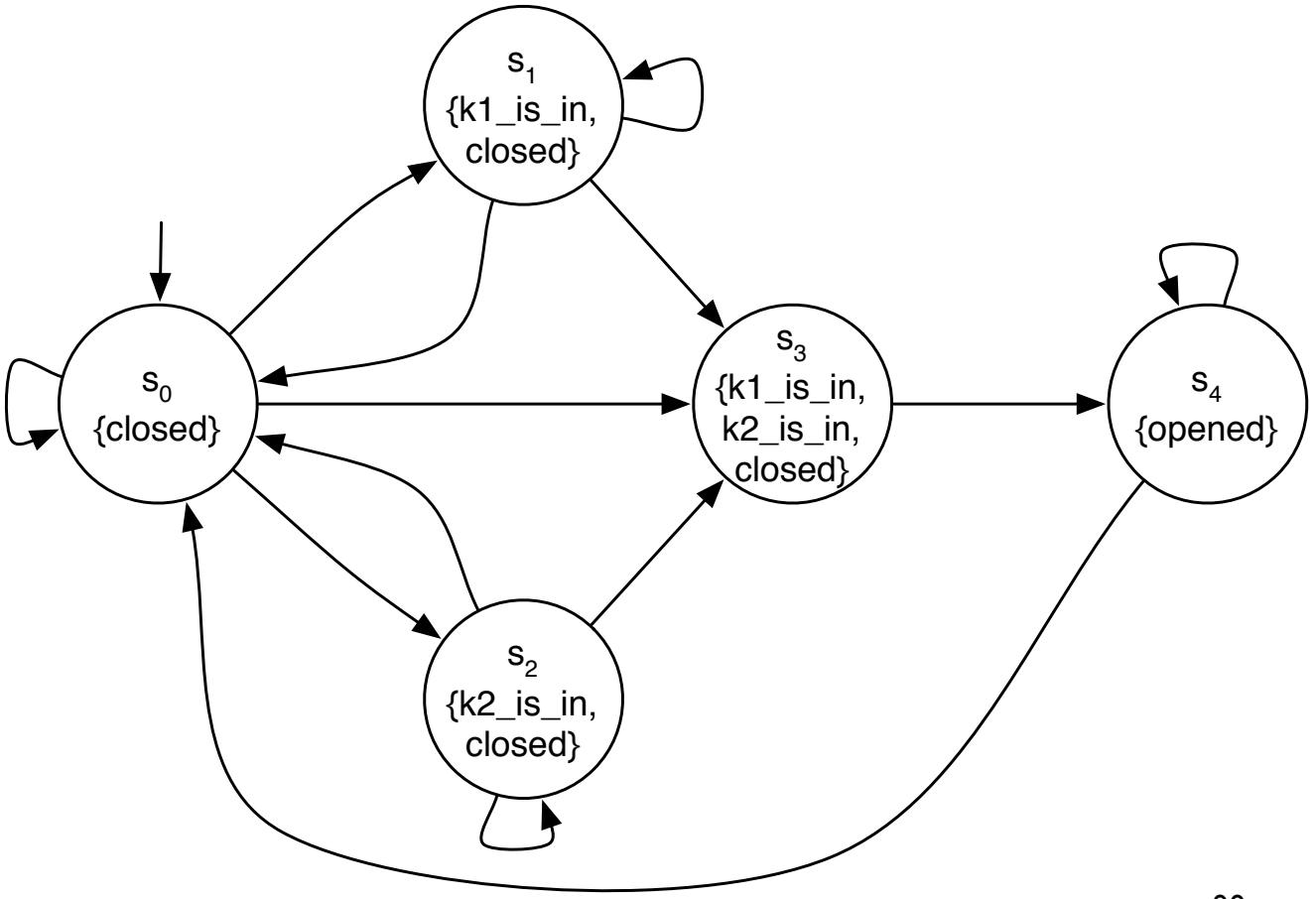


- ***k2_is_in*** $\wedge$ *E(k2_is_in U opened)*: (from *Sat*(E(k2_is_in U opened)) = {s_4, s_3, s_2})
  - *Sat*(k2_is_in $\wedge$ E(k2_is_in U opened)) = {s_2, s_3}
- ***EX****(k2_is_in* $\wedge$ *E(k2_is_in U opened)):*
  - *Sat*(EX(k2_is_in $\wedge$ E(k2_is_in U opened))) = {s_0, s_1, s_2}
- For ***¬k2_is_in*** $\wedge$ *EX(k2_is_in* $\wedge$ *E(k2_is_in U opened)):*
  - *Sat*(¬k2_is_in $\wedge$ EX(k2_is_in $\wedge$ E(k2_is_in U opened))) = {s_0, s_1}



- P = **EF**(¬k2_is_in ∧ EX(k2_is_in ∧ E(k2_is_in U opened))), since EFφ =E(⊤Uφ):
  - *Sat*(P) = *Sat*(EF(¬k2_is_in ∧ EX(k2_is_in ∧ E(k2_is_in U opened)))) = $\{s_0, s_1, s_2, s_3, s_4\}$

Since $S_0 = \{s_0\} \subseteq$ *Sat*(P), the property holds for the model



Time and Space Complexity

For transition system TS with N states and K transitions, and CTL formula Φ, the CTL model-checking problem $TS \models \Phi$ can be determined in time $\mathcal{O}((N+K) \cdot |\Phi|)$.

- It then seems that CTL model checking is much more efficient than LTL model checking!
- …but LTL formulae may be exponentially shorter than any of their equivalent formulation in CTL (if they exist).
- In fact, if P ≠ NP, then there exist LTL formulae φ_n of polynomial length in n, such that all CTL-equivalent formulae have not polynomial length
  - Also, for those properties that can be expressed in both CTL and LTL, it is possible to make LTL model checking linear as well.

Aspect	*Linear time*	*Branching time*
"behavior" in a state s	path-based: $trace(s)$	state-based: computation tree of s
temporal logic	LTL: path formulae φ $s \models \varphi$ iff $\forall \pi \in Paths(s).\, \pi \models \varphi$	CTL: state formulae existential path quantification $\exists\varphi$ universal path quantification: $\forall\varphi$
complexity of the model checking problems	PSPACE–complete $\mathcal{O}\left(\lvert TS\rvert \cdot \exp(\lvert\varphi\rvert)\right)$	*PTIME* $\mathcal{O}(\lvert TS\rvert \cdot \lvert\Phi\rvert)$
implementation-relation	trace inclusion and the like (proof is PSPACE-complete)	simulation and bisimulation (proof in polynomial time)
fairness	no special techniques needed	special techniques needed

Table 6.1: Linear-time vs. branching-time in a nutshell.

SYMBOLIC MODEL CHECKING

Symbolic Model Checking

- Procedure we showed has explicit representation of all states. What if number of states is very large?
- Model checking procedure has been reformulated *symbolically*: *sets of states* and *sets of transitions* are represented rather than single states and single transitions
- Operation on states are replaced by set operations, that might be computed more efficiently.
  - analogy with BMC
- Typically, boolean encoding of the states, with Ordered Binary Decision Diagrams to encode transition relation.

Binary encoding of states and transitions

- Fix an arbitrary binary encoding of the states in S:
- $\eta : S \rightarrow \{0, 1\}^n$ with $n = \lceil \log_2(|S|) \rceil$
- Represent any subset X of S with a boolean function f_X (characteristic function)
  - f_I is the characteristic function of the set I of initial states
- Represent the transition relation with a boolean function $f_{\rightarrow}: \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}$
- If we can find a good method to represent boolean functions, many operations on sets may become very efficient

Example: a binary decision tree

- Shannon expansion: for every boolean function:
  $f(z_1,\ldots, z_n) = (z_1 \cdot f|_{z_1=1}) + (\overline{z}_1 \cdot f|_{z_0=0})$
- Doing it for every variable in the order 1, .., n.

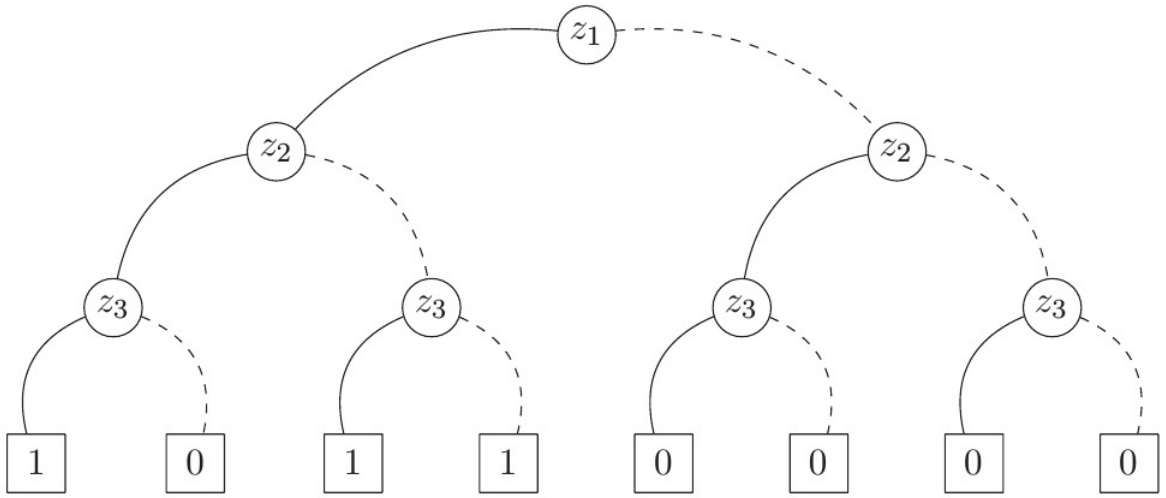


Figure 6.20: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$.

OBDD: f (z1,z2,z3) = z1 ∧ (¬z2 ∨ z3)

- All terminal nodes in the right subtree of the root have value 0: the inner tests for variables z2 and z3 in that subtree are redundant and the whole subtree can be replaced by a terminal node with value 0.
- Similarly, the subtree with root the z3 -node can be replaced with a terminal node for the value 1.

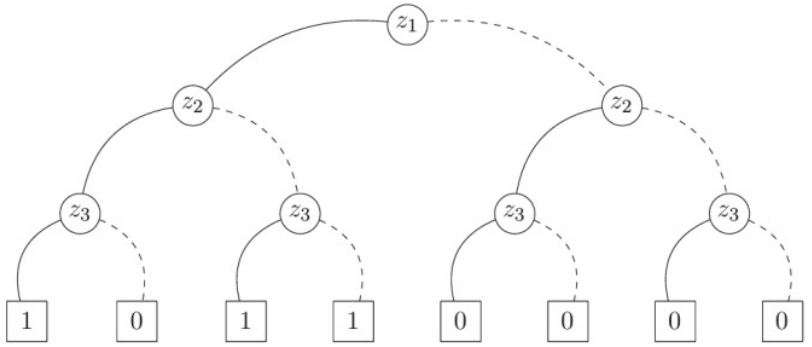
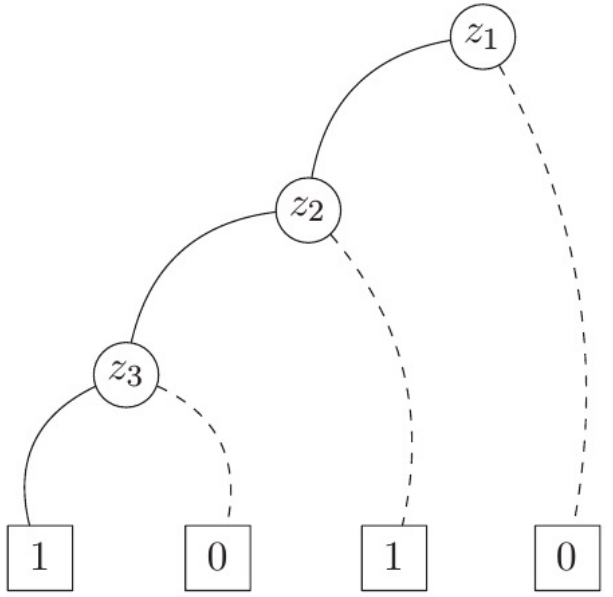
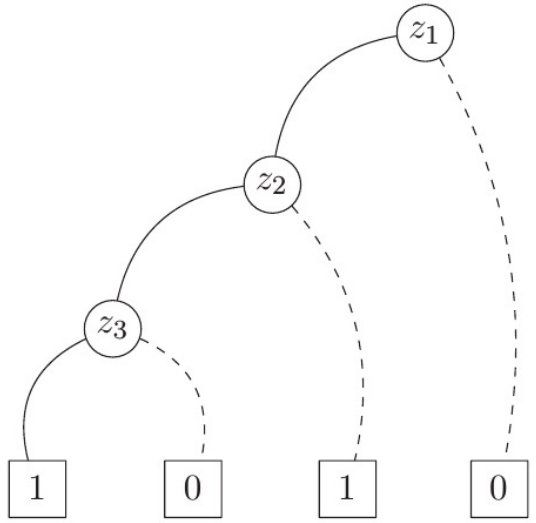


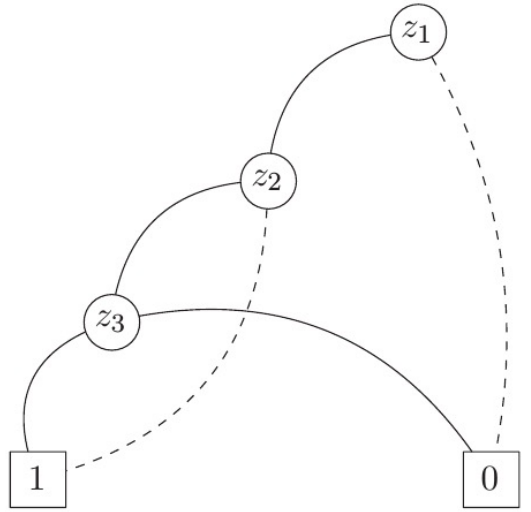
Figure 6.20: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$.



Compacting the graph

- We may identify all terminal nodes with the same value, which yields the graph on the right





In practice?

- With OBDD representation (and its many variations), performance of algorithms can be greatly improved.
  - OBDD in some cases can be very compact
  - Operations on single states are replaced by efficient operations on whole OBBDs
  - Selecting the right variable order is crucial.
- Dealing with 10^{30} or more states is often possible
  - But worst-case complexity does not change!
- Symbolic Model Checking has shown many successes in Hardware Verification
  - Less successful in software verification, where BMC has the edge
- Main tool: Nu-SMV (which also supports Bounded MC)