

Finite State Verification

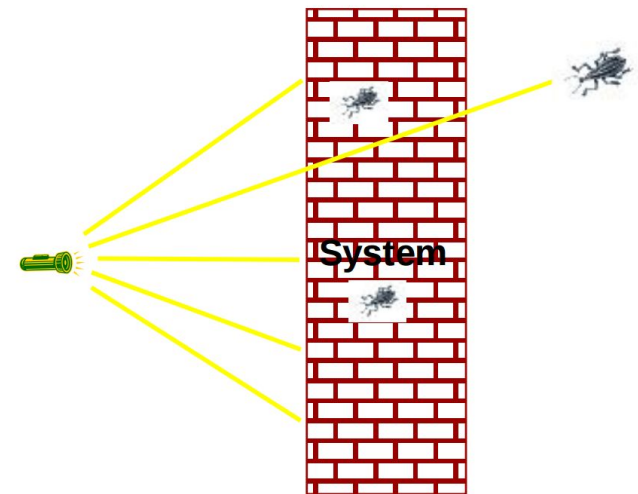
CSCE 747 - Lecture 14 - 02/25/2016

So, You Want to Perform Verification...

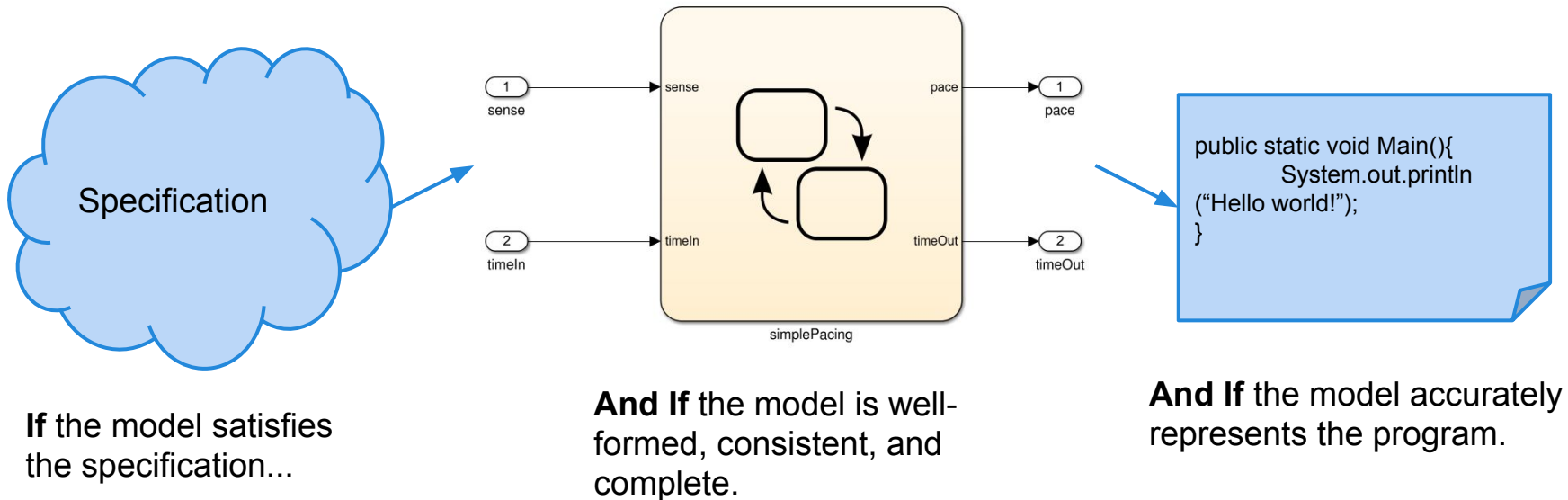
- You have a property that you want your program to obey.
- Great! Let's write some tests!
- **Does testing guarantee that the requirement is met?**
 - Not quite...
 - Testing can make a **statistical** argument in favor of verification, but usually cannot guarantee that the requirement holds in *all* situations.

Testing

- Any real system has a near-infinite number of possible inputs.
 - Models are simplified, but still may have trillions of inputs.
- Some faults trigger failures extremely rarely, or under conditions that are hard to control and recreate through testing.
- How can we *prove* that our system meets the property?



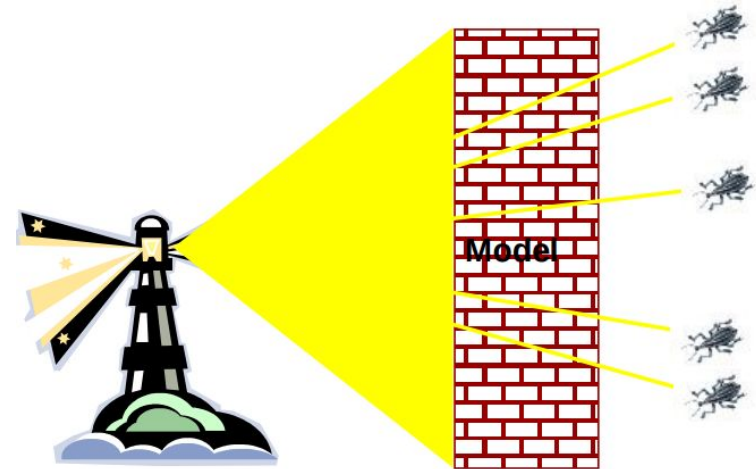
What Can We Do With This Model?



If we can show that the model satisfies the requirement, then the program should as well.

Finite-State Verification

- Express specification as a set of logical properties, written as Boolean formulae.
- Exhaustively search the state space of the model for violations of those properties.
- If the property holds - proof that the model is correct.
- Contrast with testing - no violation might just mean bad tests.



Today's Goals

- Formulating specification statements as formal logical expressions.
 - Introduction to temporal logic.
- Building behavioral models in NuSMV.
- Performing finite-state verification over the model.
 - Exhaustive search algorithms.

Expressing Specification Statements as Provable Properties

Expressing Properties

- Properties expressed in a formal logic.
 - Temporal logic ensures that properties hold over execution paths, not just at a single point in time.
- Safety Properties
 - System **never** reaches bad state.
 - **Always** in some good state.
- Liveness Properties
 - **Eventually** useful things happen.
 - **Fairness** criteria.

Temporal Logic

- Sets of rules and symbolism for representing propositions qualified over time.
- Linear Time Logic (LTL)
 - Reason about events over a timeline.
- Computation Tree Logic (CTL)
 - Branching logic that can reason about multiple timelines.
- We need both forms of logic - each can express properties that the other cannot.

Linear Time Logic Formulae

Formulae written with propositional variables (boolean properties), logical operators (and, or, not, implication), and a set of modal operators:

X (next)	X hunger	In the next state, I will be hungry.
G (globally)	G hunger	In all future states, I will be hungry.
F (finally)	F hunger	Eventually, there will be a state where I am hungry.
U (until)	hunger U burger	I will be hungry until I start to eat a burger.
R (release)	hunger R burger	I will cease to be hungry after I eat a burger.

LTL Examples

- **X (next)** - This operator provides a constraint on the next moment in time.
 - $(\text{sad} \ \&\& \ \text{!rich}) \rightarrow X(\text{sad})$
 - $((x==0) \ \&\& \ (\text{add3})) \rightarrow X(x == 3)$
- **F (finally)** - At some point in the future, this property will be true.
 - $(\text{funny} \ \&\& \ \text{ownCamera}) \rightarrow F(\text{famous})$
 - $\text{sad} \rightarrow F(\text{happy})$
 - $\text{send} \rightarrow F(\text{receive})$

LTL Examples

- G (globally) - This property must always be true.
 - winLottery \rightarrow G(rich)
- U (until) - One property must be true until the second becomes true.
 - startLecture \rightarrow (talk U endLecture)
 - born \rightarrow (alive U dead)
 - request \rightarrow (!reply U acknowledgement)

More LTL Examples

- $G(\text{requested} \rightarrow F(\text{received}))$
- $G(\text{received} \rightarrow X(\text{processed}))$
- $G(\text{processed} \rightarrow F(G(\text{done})))$
- If the above are true, can this be true?
 - $G(\text{requested}) \ \&\& \ G(\text{!done})$

Computation Tree Logic Formulae

Combines quantifiers over all paths and path-specific quantifiers:

A (all)	A hunger	Starting from the current state, I must be hungry on all paths.
E (exists)	E hunger	There must be some path, starting from the current state, where I am hungry.

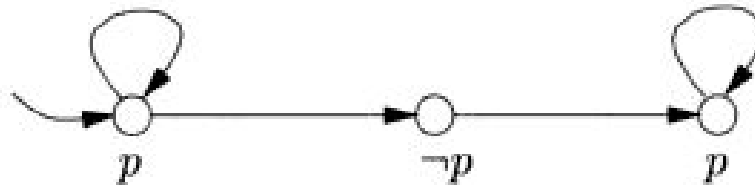
X (next)	X hunger	In the next state on this path, I will be hungry.
G (globally)	G hunger	In all future states on this path, I will be hungry.
F (finally)	F hunger	Eventually on this path, there will be a state where I am hungry.
U (until)	hunger U burger	On this path, I will be hungry until I start to eat a burger. (I must eventually eat a burger)
W (weak until)	hunger W burger	On this path, I will be hungry until I start to eat a burger. (There is no guarantee that I eat a burger)

CTL Examples

- chocolate = “I like chocolate.”
- warm = “It is warm outside.”
- AG chocolate
- EF chocolate
- AF (EG chocolate)
- EG (AF chocolate)
- AG (chocolate U warm)
- EF ((EX chocolate) U (AG warm))

Examples

- It is always possible to reach a state where we can reset.
 - **AG (EF reset)**
 - Is the LTL formula **G (F reset)** the same expression?
- Eventually, the system will reach a good state and remain there.
 - **F (G good)**
 - Is the CTL formula **AF (AG good)** the same?

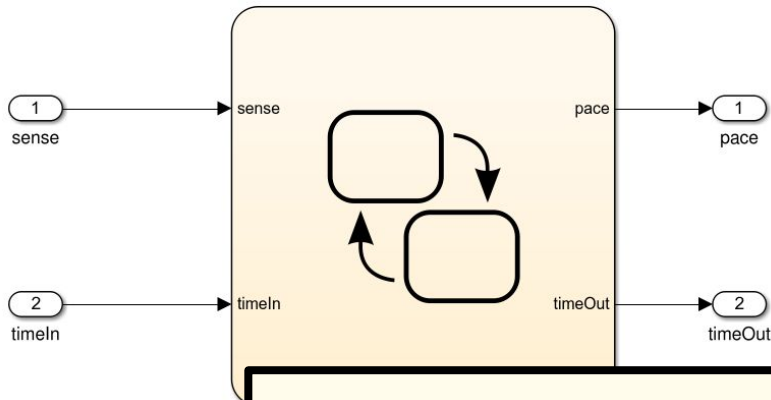


Proving Properties Over Models

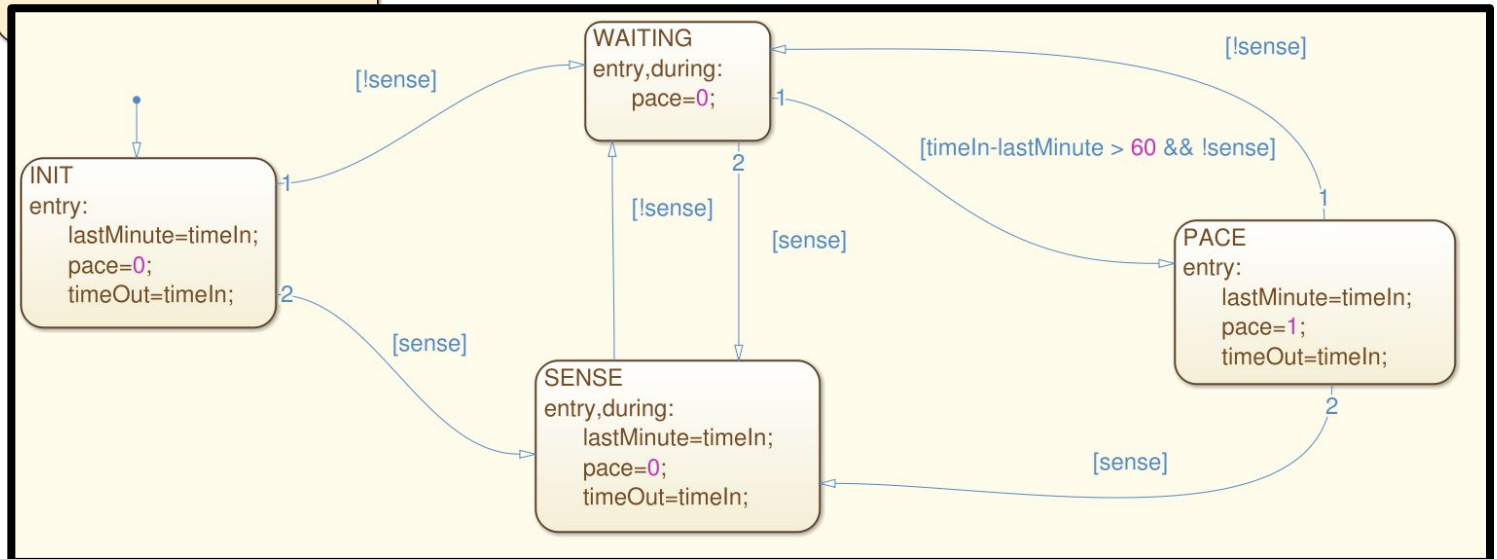
Building Models

- Many different modeling languages.
- Most verification tools use their own language.
- Conceptually, most map to state machines.
 - Define a list of variables.
 - Describe how their values are calculated.
 - Each “time step”, recalculate the values of these variables.
 - The state is the current set of values for all variables.

Creating Models - Graphical



- Most common industrial framework: Stateflow



Creating Models - Written Language

```
MODULE main
VAR
request: boolean;
status: {ready, busy};
ASSIGN
init(status) := ready;
next(status) :=
case
    status=ready & request: busy;
    status=ready & !request : ready;
    TRUE: {ready, busy};
esac;
```

- NuSMV modeling language
- Part of a framework for model analysis.

Proving Properties

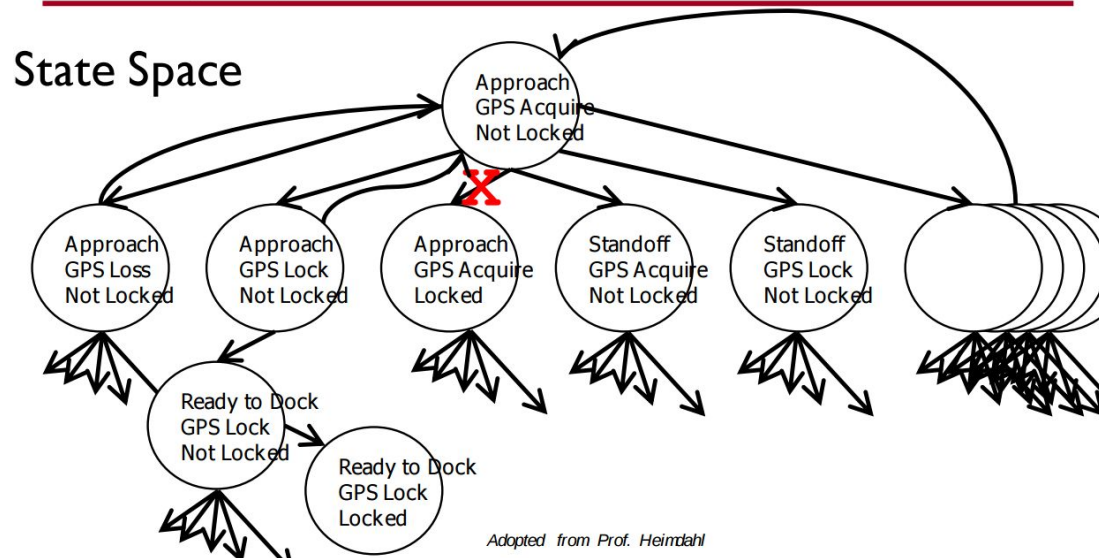
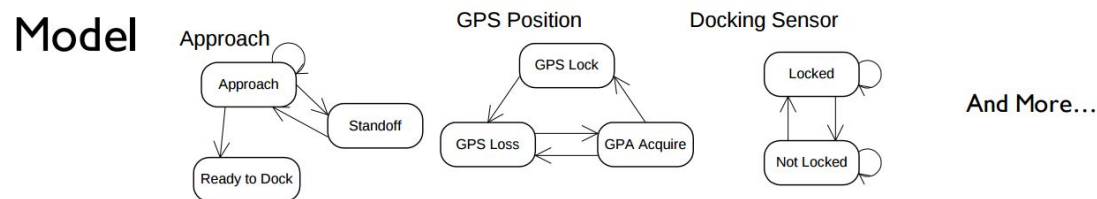
- To perform verification, we take properties and exhaustively search the state space of the model for violations.
- Violations give us counter-examples
 - A path that demonstrates how the property has been violated.
- Implications:
 - Property is incorrect.
 - Model does not reflect expected behavior.
 - Real issue found in the system being designed.

Test Generation from FS Verification

- We can also take properties and **negate** them.
 - Called a “trap property” - we assert that a property can never be met.
- The counter-example shows one way the property can be met.
- This can be used as a test for the real system - to demonstrate that the final system meets its specification.

Exhaustive Search

- Algorithms exhaustively comb through the possible execution paths through the model.
- Major limitation - state space explosion.



Exhaustive Search - Dining Philosophers

- Problem - X philosophers sit at a table with Y forks between them. Philosophers may think or eat. When they eat, they need two forks.
- Goal is to avoid deadlock - a state where no progress is possible.
 - 5 philosophers/forks - deadlock after exploring 145 states
 - 10 philosophers/forks - deadlock after exploring 18,313 states
 - 15 philosophers/forks - deadlock after exploring 148,897 states
 - 9 philosophers/10 forks - deadlock found after exploring 404,796 states

Search Based on SAT

- Express properties as conjunctive normal form expressions:
 - $f = (!x_2 \ || \ x_5) \ \&\& \ (x_1 \ || \ !x_3 \ || \ x_4) \ \&\& \ (x_4 \ || \ !x_5) \ \&\& \ (x_1 \ || \ x_2)$
- Examine reachable states and choose a transition based on how it affects the CNF expression.
 - If we want x_2 to be false, choose a transition that imposes that change.
- Continue until CNF expression is satisfied.

Branch & Bound Algorithm

- Set a variable to a particular value (true/false).
- Apply that value to the CNF expression.
- See whether that value satisfies all of the clauses that it appears in.
 - If so, assign a value to the next variable.
 - If not, backtrack (bound) and apply the other value.
- Prune branches of the boolean decision tree as values are applied.

Branch & Bound Algorithm

$$f = (!x_2 \ || \ x_5) \ \&\& \ (x_1 \ || \ !x_3 \ || \ x_4) \ \&\& \ (x_4 \ || \ !x_5) \ \&\& \ (x_1 \ || \ x_2)$$

1. Set x_1 to false.

$$f = (!x_2 \ || \ x_5) \ \&\& \ (0 \ || \ !x_3 \ || \ x_4) \ \&\& \ (x_4 \ || \ !x_5) \ \&\& \ (0 \ || \ x_2)$$

2. Set x_2 to false.

$$f = (1 \ || \ x_5) \ \&\& \ (0 \ || \ !x_3 \ || \ x_4) \ \&\& \ (x_4 \ || \ !x_5) \ \&\& \ (0 \ || \ 0)$$

3. Backtrack and set x_1 to true.

$$f = (0 \ || \ x_5) \ \&\& \ (0 \ || \ !x_3 \ || \ x_4) \ \&\& \ (x_4 \ || \ !x_5) \ \&\& \ (0 \ || \ 1)$$

DPLL Algorithm

- Set a variable to a particular value (true/false).
- Apply that value to the CNF expression.
- If the value satisfies a clause, that clause is removed from the formula.
- If the variable is negated, but does not satisfy a clause, then the variable is removed from that clause.
- Repeat until a solution is found.

DPLL Algorithm

$f = (!x2 \ || \ x5) \ \&\& \ (x1 \ || \ !x3 \ || \ x4) \ \&\& \ (x4 \ || \ ! \ x5) \ \&\& \ (x1 \ || \ x2)$

1. Set x2 to false.

$f = (1 \ || \ x5) \ \&\& \ (x1 \ || \ !x3 \ || \ x4) \ \&\& \ (x4 \ || \ ! \ x5) \ \&\& \ (x1 \ || \ 0)$

$f = (x1 \ || \ !x3 \ || \ x4) \ \&\& \ (x4 \ || \ ! \ x5) \ \&\& \ (x1)$

2. Set x1 to true.

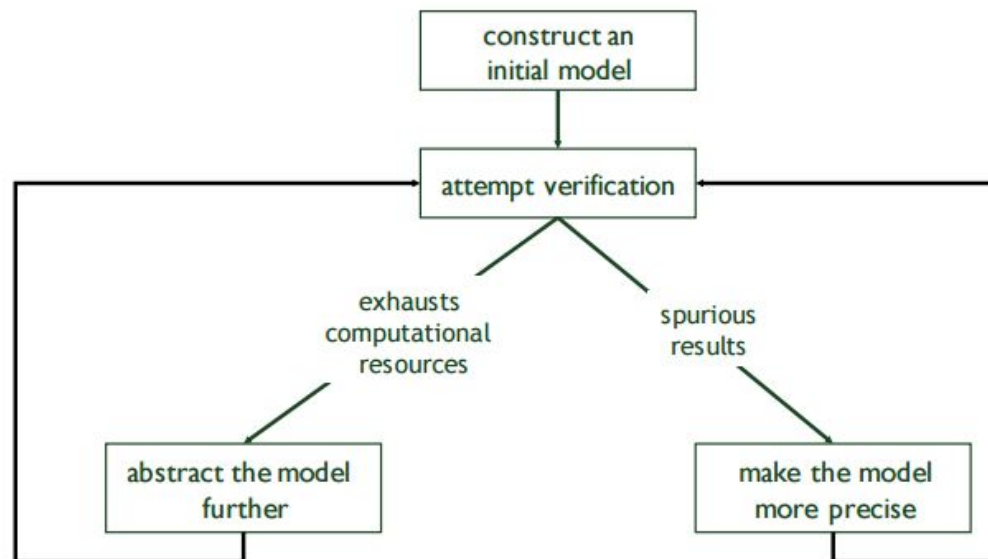
$f = (1 \ || \ !x3 \ || \ x4) \ \&\& \ (x4 \ || \ ! \ x5) \ \&\& \ (1)$

$f = (x4 \ || \ ! \ x5)$

3. Set x4 to false, then x5 to false.

Model Refinement

- Models have to balance precision with efficiency.
- Abstractions that are too simple may introduce spurious failure paths that may not be in the real system.
- Models that are too complex may render model checking infeasible due to resource exhaustion.



Intensional Models

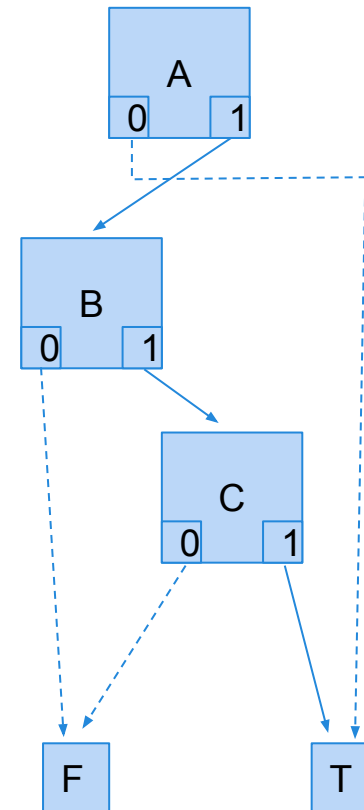
- State space can be limited by replacing *extensional* representations with *intensional* representations
 - A positive even integer < 20 :
 - Extensional: $\{2, 4, 6, 8, 10, 12, 14, 16, 18\}$
 - (All concrete values)
 - Intensional: $\{x \in \mathbb{N} \mid x \bmod 2 = 0 \wedge 0 < x < 20\}$
 - (Symbolic representation)
 - Equation called the *characteristic function*
 - A predicate true for all elements in the set of values and false otherwise.

Ordered Binary Decision Diagrams

- We can represent whether or not there is a transition between two states using a characteristic function.
 - $f(m,n) = \text{true}$ if there is a transition from m to n .
- OBDDs are a data structure representing the calculation of a binary function.
 - Such as a characteristic function.
 - Can be used to represent a subset of the state space.

OBDD Example

- $\neg a \vee (b \wedge c)$
 - Can be thought of as a function: $f(a,b,c)$
 - Returns true if the property is satisfied, false if not.

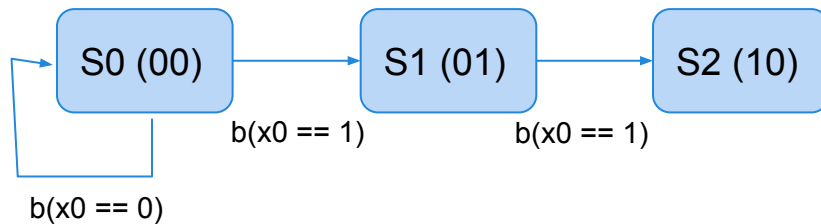


Ordered Binary Decision Diagrams

- Built by iteratively expanding the set of states reachable in $k+1$ steps.
 - Stabilizes when the number of transitions that can occur in the next step are already included.
- Most basic form - what states can we reach from the current state in n transitions?
- Often, merged with specification properties:
 - The set of transitions leading to a violation of the property.
 - If that set is empty, the property is verified.
 - Symbolic model checking.

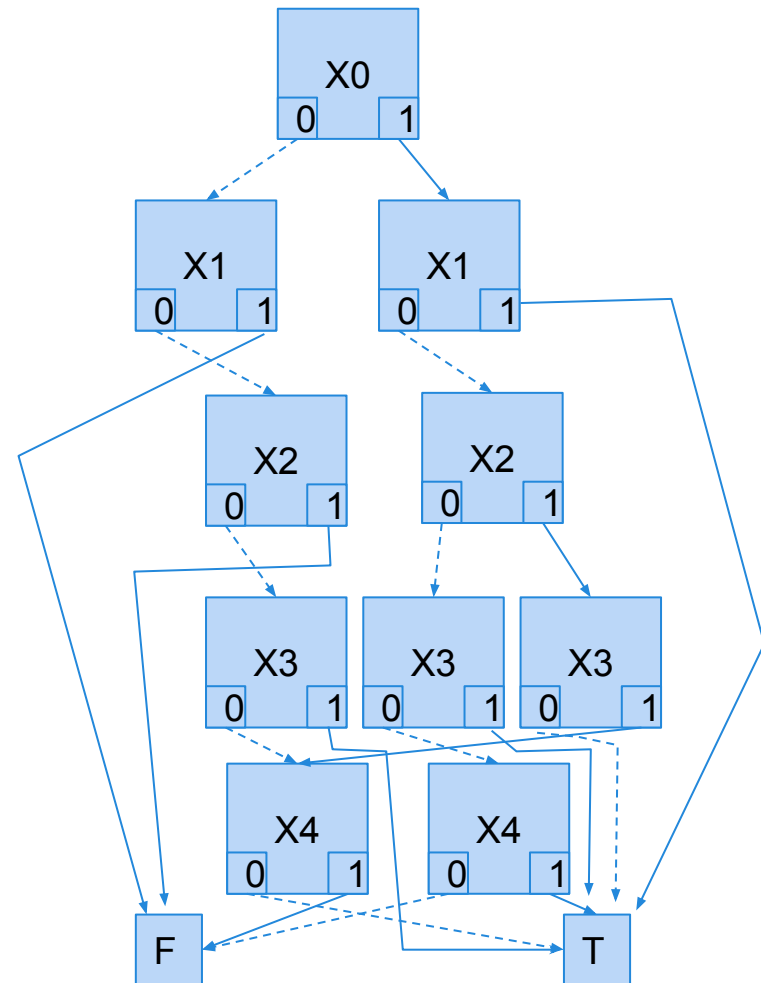
Building OBDDs

- Assign each state and symbol a boolean label.



- Encode transitions as tuples (sym, from, to)

x0	x1x2	x3x4
0	00	00
1	00	01
1	01	10
sym	from state	to state



Benefits of OBDDs

- OBDDs allow us to represent *sets of states* symbolically.
 - Rather than reasoning over the entire state space, we can reason over a small representation of a set of states (the boolean characteristic function).
- Allows verification of much larger models than explicit model checking.
 - As long as we *can* represent states with such a function.
 - Best when there is a large degree of regularity in the state space.

We Have Learned

- We can perform verification by creating models of the system and proving that the specification properties hold over the model.
- To do so, we must express specifications as sets of logical formulae written in a temporal logic.
- Finite state verification exhaustively searches the state space for violations of properties.

We Have Learned

- By performing this process, we can gain confidence that the system will meet the specifications.
 - We can even generate test cases from the model to help demonstrate that properties still hold over the final system.

Next Time

- Symbolic execution and proof of properties
- Reading: Chapters 7

- Homework:
 - Reading assignment 3 - due soon.
 - Assignment 3 now available.