Finite State Verification

CSE 4495 - Lecture 9 - 10/09/2022

Instructor : Md. Mohaiminul Islam

mohaiminul@cse.uiu.ac.bd

So, You Want to Perform Verification...

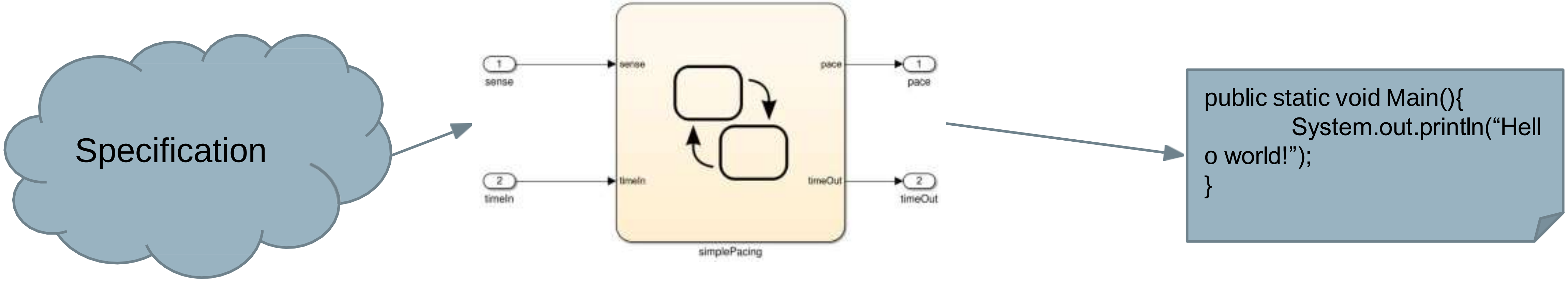
- You have a requirement the program must obey.
- Great! Let's write some tests!
- **Does testing guarantee the requirement is met?**

  - Not quite…
    - Testing can only make a **statistical** argument.

What About a Model?

- We have previously used models to create tests.
  - Models are simpler than the real program.
  - By abstracting away unnecessary details, we can learn important insights.
- Models can be used to verify full programs.
  - Can see if properties hold exhaustively over a model.

What Can We Do With This Model?



If the model satisfies the specification...

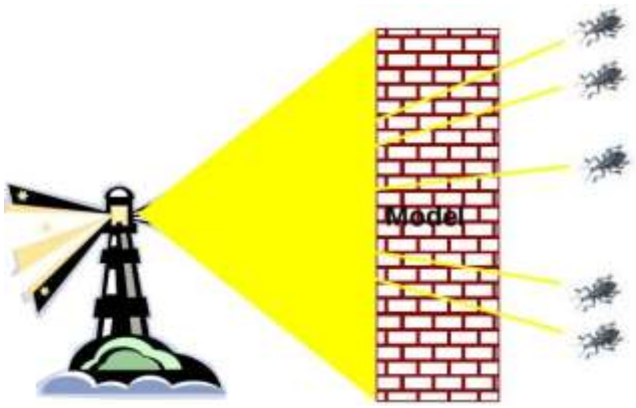
And If the model is well-formed, consistent, and complete.

And If the model accurately represents the program.

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

Finite State Verification

- Express requirements as Boolean formulae.
- Exhaustively search state space of the model for violations of those properties.
- If the property holds - proof of correctness
- Contrast with testing - no violation might mean bad tests.



Today’s Goals

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

Expressing Requirements in Temporal Logic

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.
    - “If the traffic light is red, it will always turn green within 10 seconds.”
    - “If an emergency vehicle arrives at a red light, it must turn green in the next time step.”

Expressing Properties

- Liveness Properties
  - **Eventually** useful things happen.
  - **Fairness** criteria.
  - Reason over paths of unknown length.
    - "If the light is red, it must eventually become green."
    - "If the package is shipped, it must eventually arrive."
    - "If Player A is taking a turn, Player B must be allowed a turn at some time in the future."

Temporal Logic

- Represents propositions qualified over time.
- Linear Time Logic (LTL)
  - Reason about events over a single timeline.
- Computation Tree Logic (CTL)
  - Branching logic that can reason about multiple timelines.
- Each can express properties that the other cannot.

Linear Time Logic Formulae

Formulae written with boolean predicates, logical operators (and, or, not, implication), and operators:

hunger = “I am hungry”

burger = “I eat a burger”

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. (hunger does not need to be true once burger becomes true)
R (release)	hunger R burger	I will cease to be hungry after I eat a burger. (hunger and burger are true at the same time for at least one state before hunger becomes false)

LTL Examples

- **X (next)** - This operator provides a constraint on the next moment in time.
  - `(sad && !rich) -> X(sad)`
  - `(hungry && haveMoney) -> X(orderedPizza)`
- **F (finally)** - At some point in the future, this property will be true.
  - `(funny && ownCamera) -> F(famous)`
  - `sad -> F(happy)`
  - `send -> F(receive)`

LTL Examples

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

More LTL Examples

- `G (requested -> F (received))`
- `G (received -> X (processed))`
- `G (processed -> F (G (done)))`
- If all three above are true, can this be true?
  - `G (requested -> G (!done))`

requested = action requested
received = request received
processed = request processed
done = action completed

Computation Tree Logic Formulae

Combines all-path quantifiers with 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

coffee= “*I like coffee.*” warm = “*It is warm.*”

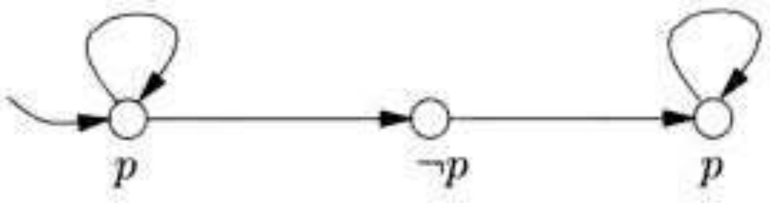
- `AG coffee`
- `EF coffee`
- `AF (EG coffee)`
- `EG (AF coffee)`
- `AG (coffee U warm)`
- `EF ((EX coffee) U (AG warm))`

Examples

- requested: a request has been made
- acknowledged: request has been acknowledged.
    - CTL: `AG (requested -> AF acknowledged)`
        - On all paths (A) from an initial state, at every state in the path (G), **if** *requested* holds true, then (->) for all paths (A) from that state, eventually (F) at some other state, *acknowledge* holds true.
    - LTL: `G (requested -> F acknowledged)`
        - On all paths from an initial state, at every state in the path (G), if *requested* holds true, then (->) eventually (F) at some other state, *acknowledge* holds true.

Examples

- It is always possible (AG) to reach a state (EF) where we can reset.
  - **`AG (EF reset)`**
  - Is LTL formula **`G (F reset)`** the same expression?



- Eventually (F), the system will reach a state where P will be true forever (G).
  - **`F (G P)`**
  - Is CTL formula **`AF (AG P)`** the same?

Building Models

Building Models

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

Building Models in NuSMV

- NuSMV is a symbolic model checker.
  - Models written in a basic language, represented using Binary Decision Diagrams (BDDs).
    - BDDs translate concrete states into compact summary states.
    - Allows large models to be processed efficiently.
  - Properties may be expressed in CTL or LTL.
  - If a model may be falsified, it provides a concrete counterexample demonstrating how it was falsified.

A Basic NuSMV Model

```
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;
SPEC AG(request -> AF (status = busy))
```

Models consist of one or more modules, which execute in parallel.

The state of the model is the current value of all variables.

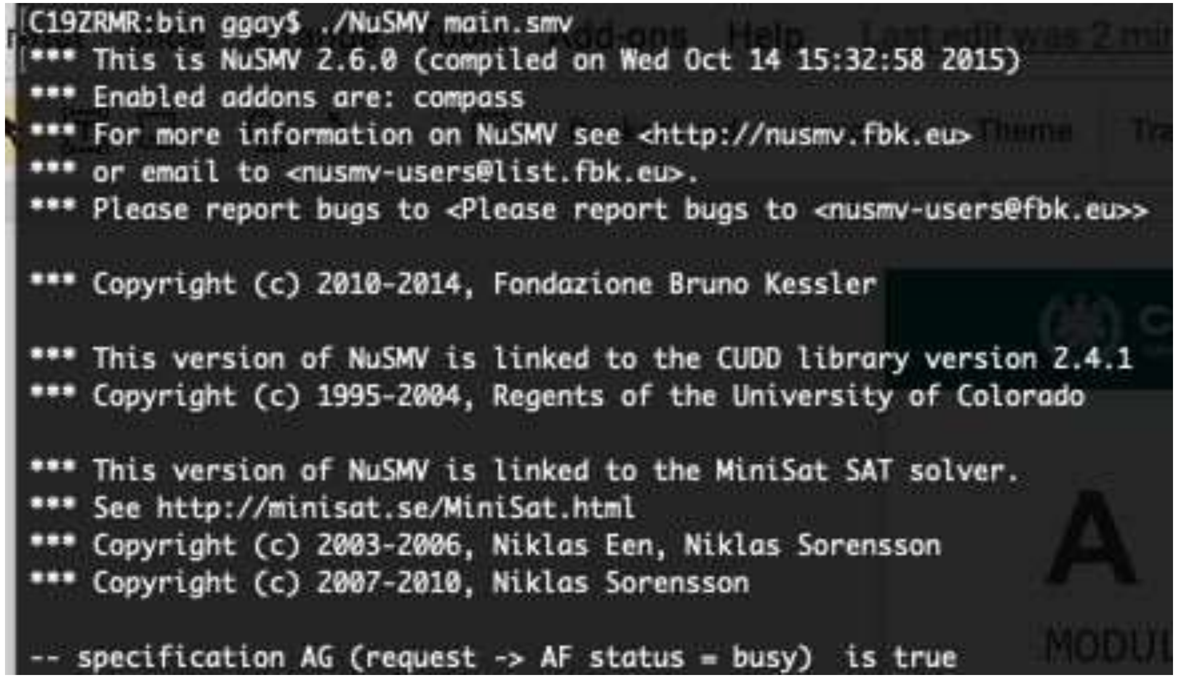
Expressions define how the state of each variable can change.

“request” is set randomly. This represents an environmental factor out of our control.

Property we wish to prove over the model.

Checking Properties

- Execute from command line:
  NuSVM <model name>
- Properties that are true are indicated as true.
- If property is false, a counter-example is shown (input violating the property).



Checking Properties

- New property: `AG (status = ready)`
- (Obviously not true - we set it randomly in the absence of a request)
- Counterexample:

```
-- specification AG status = ready  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    request = FALSE
    status = ready
  -> State: 1.2 <-
    status = busy
```

  - In first state, request = false, status = ready.
  - We set status randomly for second state (because request was false). It is set to busy, violating property.

```
MODULE main
VAR
   traffic_light: {RED, YELLOW, GREEN}; ped_light:
   {WAIT, WALK, FLASH};  button: {RESET, SET};
ASSIGN
    init(traffic_light) := RED;
    next(traffic_light) := case
        traffic_light=RED & button=RESET:
                        GREEN;
        traffic_light=RED: RED;
        traffic_light=GREEN & button=SET:
                   {GREEN,YELLOW};
        traffic_light=GREEN: GREEN;
        traffic_light=YELLOW:
                   {YELLOW, RED};
        TRUE: {RED};
    esac;

init(ped_light) := WAIT;
    next(ped_light) := case
       ped_light=WAIT &
                  traffic_light=RED: WALK;
       ped_light=WAIT: WAIT;
       ped_light=WALK: {WALK,FLASH};
       ped_light=FLASH: {FLASH, WAIT};
       TRUE: {WAIT};
    esac;
    next(button) := case
       button=SET & ped_light=WALK: RESET;
       button=SET: SET;
       button=RESET & traffic_light=GREEN:
                 {RESET,SET};
       button=RESET: RESET;
       TRUE: {RESET};
    esac;
```

Let’s Take a Break

- Describe a safety property (something does or does not happen at a specific time) and formulate in CTL.
- Describe a liveness property (something eventually happens) and formulate in LTL.

```
MODULE main
VAR
   traffic_light: {RED, YELLOW, GREEN}; ped_light:
   {WAIT, WALK, FLASH};  button: {RESET, SET};
ASSIGN
    init(traffic_light) := RED;
    next(traffic_light) := case
        traffic_light=RED & button=RESET:
                          GREEN;
        traffic_light=RED: RED;
        traffic_light=GREEN & button=SET:
                   {GREEN,YELLOW};
        traffic_light=GREEN: GREEN;
        traffic_light=YELLOW:
                   {YELLOW, RED};
        TRUE: {RED};
    esac;
```

```
init(ped_light) := WAIT;
    next(ped_light) := case
       ped_light=WAIT &
                   traffic_light=RED: WALK;
       ped_light=WAIT: WAIT;
       ped_light=WALK: {WALK,FLASH};
       ped_light=FLASH: {FLASH, WAIT};
       TRUE: {WAIT};
    esac;
    next(button) := case
       button=SET & ped_light=WALK: RESET;
       button=SET: SET;
       button=RESET & traffic_light=GREEN:
                 {RESET,SET};
       button=RESET: RESET;
       TRUE: {RESET};
    esac;
```

Activity - Potential Solutions

- Safety Property
  - A bad thing never happens, or a good thing happens at a specific time.
- `AG (pedestrian_light = walk -> traffic_light != green)`
  - The pedestrian light cannot indicate that I should walk when the traffic light is green.
  - This is a safety property. We are saying that this should NEVER happen.

Activity - Potential Solutions

- Liveness Property
  - **Eventually** useful things happen.
- `G (traffic_light = RED & button = RESET -> F (traffic_light = green))`
  - If the light is red, and the button is reset, then eventually, the light will turn green.
  - This is a liveness property, as we assert that something will eventually happen.

Proving Properties Over Models

Proving Properties

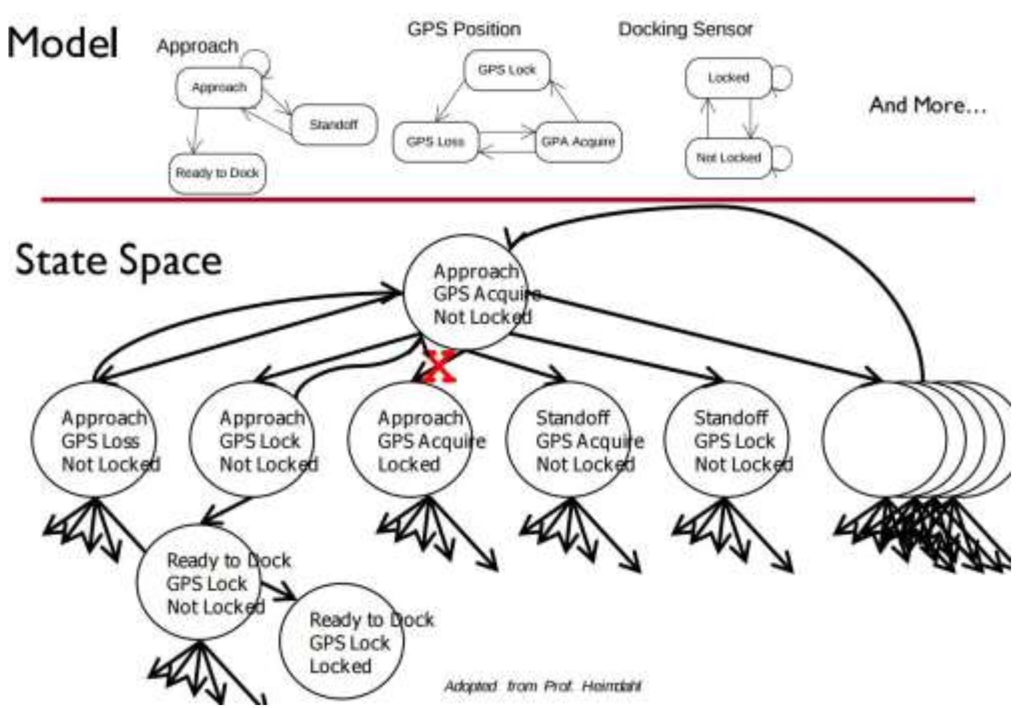
- Search state space for property violations.
- Violations give us counter-examples
  - Path that demonstrates the violation.
  - (useful test case)
- Implications of counter-example:
  - Property is incorrect.
  - Model does not reflect expected behavior.
  - Real issue found in the system being designed.

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.
- Shows one way the property can be met.
- Can be used as a test for the real system.
  - Demonstrate that final system meets specification.

Exhaustive Search

- Algorithms examine all execution paths through the state space.
- Major limitation - state space explosion.
  - Limit number of variables and possible values to control state space size.



Search Based on SAT

- Express properties in **conjunctive normal form**:
  - `f = (!x2 || x5) && (x1 || !x3 || x4) && (x4 || ! x5) && (x1|| x2)`
- Examine reachable states and choose a transition based on how it affects the CNF expression.
  - If we want `x2` to be false, choose a transition that imposes that change.
- Continue until CNF expression is satisfied.

Boolean Satisfiability (SAT)

- Find assignments to Boolean variables $X_1, X_2, \ldots, X_n$ that results in expression φ evaluating to true.
- Defined over expressions written in **conjunctive normal form**.
  - $\varphi = (X_1 \lor \neg X_2) \land (\neg X_1 \lor X_2)$
  - $(X_1 \lor \neg X_2)$ is a **clause**, made of variables, $\neg$, $\lor$
  - Clauses are joined with $\land$

Boolean Satisfiability

- Find assignment to X_1, X_2, X_3, X_4, X_5 to solve
  - $(\neg X_2 \lor X_5) \land (X_1 \lor \neg X_3 \lor X_4) \land (X_4 \lor \neg X_5) \land (X_1 \lor X_2)$
- One solution: 1, 0, 1, 1, 1
  - $(\neg X_2 \lor X_5) \land (X_1 \lor \neg X_3 \lor X_4) \land (X_4 \lor \neg X_5) \land (X_1 \lor X_2)$
  - $(\neg 0 \lor 1) \land (1 \lor \neg 1 \lor 1) \land (1 \lor \neg 1) \land (1 \lor 0)$
  - $(1) \land (1) \land (1) \land (1)$
  - 1

Branch & Bound Algorithm

- Set variable to true or false.
- Apply that value.
- Does value satisfy the clauses that it appears in?
  - If so, assign a value to the next variable.
  - If not, backtrack (bound) and apply the other value.
- Prunes branches of the boolean decision tree as values are applied.

Branch & Bound Algorithm

$\varphi = (\neg x2 \lor x5) \land (x1 \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (x1 \lor x2)$

1. **Set x1 to false.**
   $\varphi = (\neg x2 \lor x5) \land (\mathbf{0} \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (\mathbf{0} \lor x2)$
2. **Set x2 to false.**
   $\varphi = (\mathbf{1} \lor x5) \land (\mathbf{0} \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (\mathbf{0} \lor \mathbf{0})$
3. **Backtrack and set x2 to true.**
   $\varphi = (\mathbf{0} \lor x5) \land (\mathbf{0} \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (\mathbf{0} \lor \mathbf{1})$

DPLL Algorithm

- Set a variable to true/false.
  - Apply that value to the expression.
  - Remove all satisfied clauses.
  - If assignment does not satisfy a clause, then remove that variable from that clause.
  - If this leaves any **unit clauses** (single variable clauses), assign a value that removes those next.
- Repeat until a solution is found.

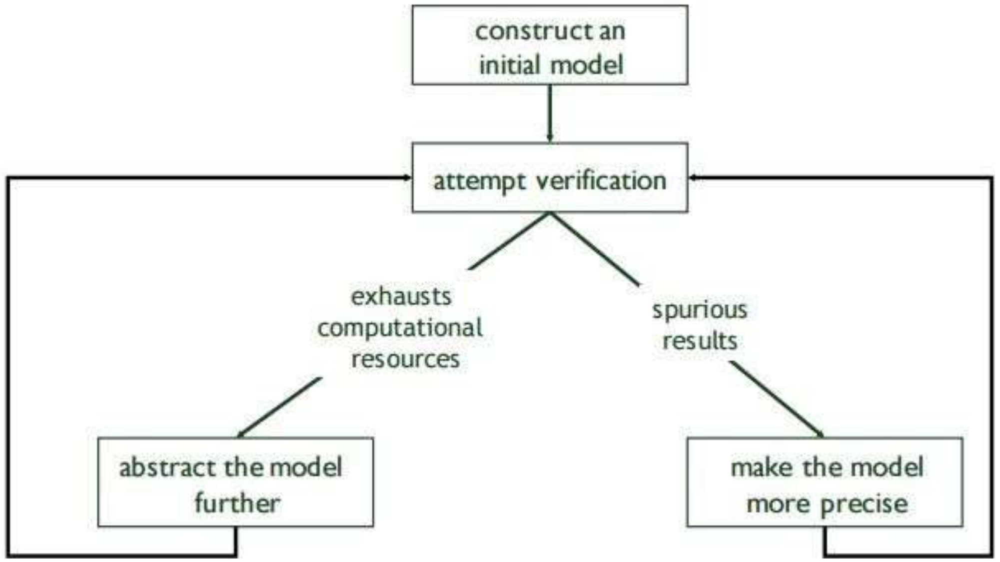
DPLL Algorithm

$\varphi = (\neg x2 \lor x5) \land (x1 \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (x1 \lor x2)$

1. **Set x2 to false.**
   $\varphi = (\neg \mathbf{0} \lor x5) \land (x1 \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (x1 \lor \mathbf{0})$
   $\varphi = (x1 \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (x1)$
2. **Set x1 to true.**
   $\varphi = (\mathbf{1} \lor \neg x3 \lor x4) \land (x4 \lor \neg x5) \land (\mathbf{1})$
   $\varphi = (x4 \lor \neg x5)$
3. **Set x4 to false, then x5 to false.**
   $\varphi = (\mathbf{0} \lor \neg x5)$
   $\varphi = (\neg \mathbf{0})$

Model Refinement

- Must balance precision with efficiency.
  - Models that are too simple introduce failure paths that may not be in the real system.
  - Complex models may be infeasible due to resource exhaustion.



Who Uses This Stuff?

- Used heavily in **safety-critical** development.
  - Verifies certain complex, critical functions.
  - Used extensively in automotive, aerospace, medical development domains.
- Used to verify security policies, stateful behaviors.
  - Uses at Amazon Web Services to verify cloud security.
- Not used for all functionality.
  - Time-consuming, requires additional effort.

We Have Learned

- We can perform verification by creating models of function behavior and proving that the requirements hold over the model.
  - To do so, express requirements as logical formulae written in a temporal logic.
  - Finite state verification exhaustively searches the state space for violations of properties.
  - Presents counter-examples showing properties are violated.

We Have Learned

- By performing this process, we can gain confidence that the system will meet the specifications.
- Can also generate test cases to demonstrate that properties hold over the final system.
  - Negate a property, the counter-example shows that the property can be met.
  - Execute the input from the counter-example on the real system - should give the same result!

Next Time

- Exercise Session: Finite-State Verification

Thank You