CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 14: Finite State Verification

Gregory Gay

DIT636/DAT560 - March 5, 2025

How do we know a system is correct?

Rationalists

"It is correct because I **proved** that certain errors do not exist in the system."

Empiricists

"It is correct because I never **observed** incorrect behaviors."

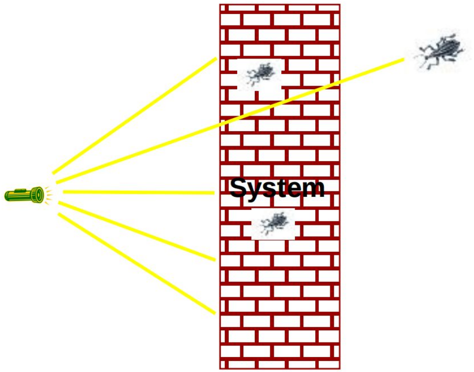
So, You Want to Perform Verification...

- You have a requirement the program must obey.
- Great! Let's write some tests!
- **Does testing prove the requirement is met?**
  - Not quite…
    - Testing can only make a **statistical** argument.

"It is correct because I **proved** that certain errors do not exist in the system."

Testing

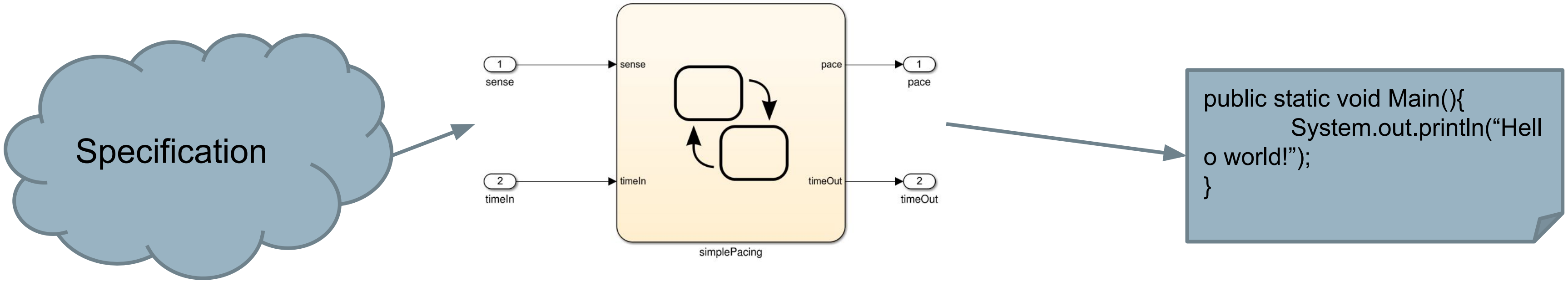
- Most systems have near-infinite possible inputs.
- Some failures are rare or hard to recreate.
  - Or require very specific input.
- How can we ***prove*** that our system meets the requirements?



What About a Model?

- We have previously used models to create tests.
  - Models are simpler than the real program.
- 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...

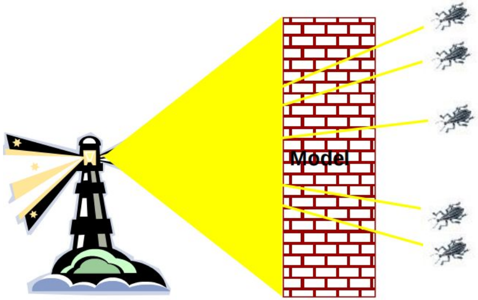
And If the model is well-formed, consistent, and complete.

And If the model accurately represents the program.

If we can **prove** that the model satisfies the requirement, then we can **argue** that 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.
  - Boolean expressions, representing facts we asset over execution paths.
  - Expressions contain boolean variables, subexpressions, and operators… as well as **temporal operators**.
- Ensures that **properties hold over execution paths**, not just at a single point in time.

Expressing Properties

- **Safety Properties**
  - Check that a specific event or sequence happens **exactly as specified**.
    - “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.”

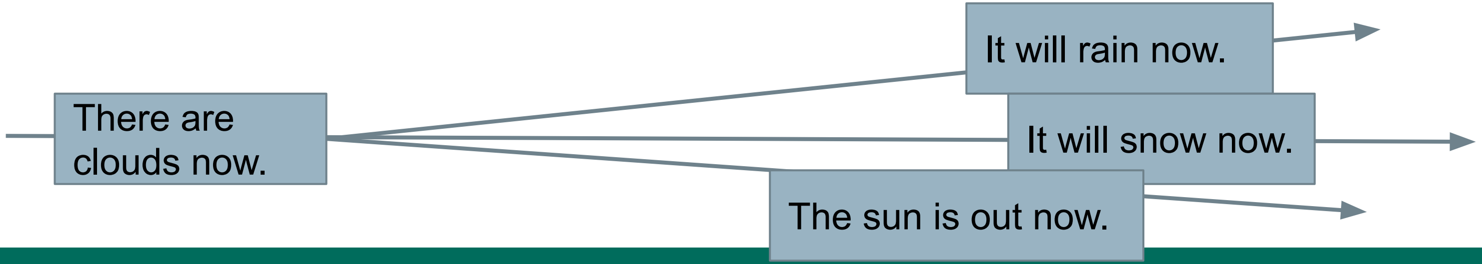
Expressing Properties

- **Liveness Properties**
  - **Eventually** something specific happens.
  - **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

- Linear Time Logic (LTL)
  - Reason about events over a single timeline.

- Computation Tree Logic (CTL)
  - Branching logic that can reason about multiple timelines.



Linear Time Logic Formulae

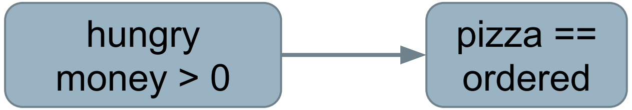
Formulae written with boolean predicates, logical operators (and, or, not, implication), and operators:

X (next)	X (weather == rain)	In the next state, it will be raining.
G (globally)	G (weather == rain)	Now and in all future states, it will be raining.
F (finally)	F (weather == rain)	Eventually, there will be a state where it is raining.
U (until)	(weather == rain) U (temperature < 0)	It will rain until the temperature drops below 0. (The value of “weather” can change once temperature is less than 0)
R (release)	weather == rain) R (temperature < 0)	It will cease to rain after the temperature drops below 0. (Both operands must be true at the same time for at least one state before the value of “weather” can change)

LTL Examples

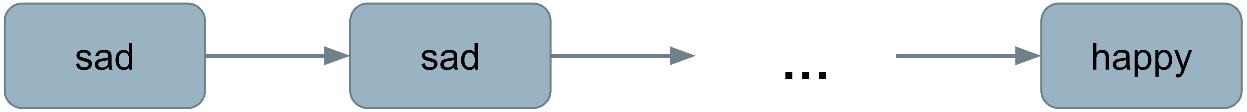
- **X (next)** - This operator provides a constraint on the next moment in time.
  - `((emotion == sad) && (money == 0)) -> X(emotion == sad)`

  - `((emotion == hungry) && (money > 0)) -> X(pizza == ordered)`



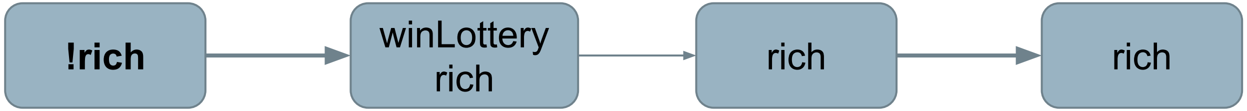
LTL Examples

- **F (finally)** - At some unknown point in the future, this property will be true.
  - `((status == funny) && ownCamera) -> F(status == famous)`
  - `(emotion == sad) -> F(emotion == happy)`
  - `(letter == sent) -> F(letter == received)`

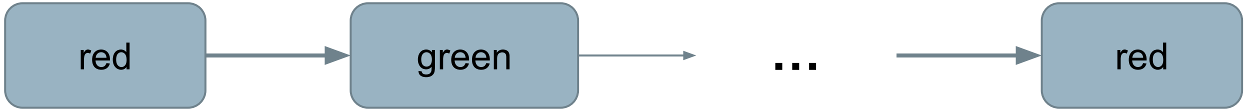


LTL Examples

- **G (globally)** - Property must be true now and forever.
  - `G(winLottery -> G(rich))`

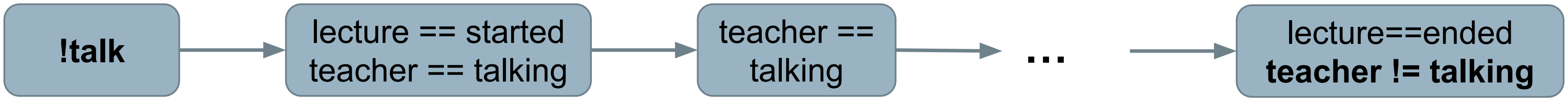


  - `G((light==green) -> F(light==red))`



LTL Examples

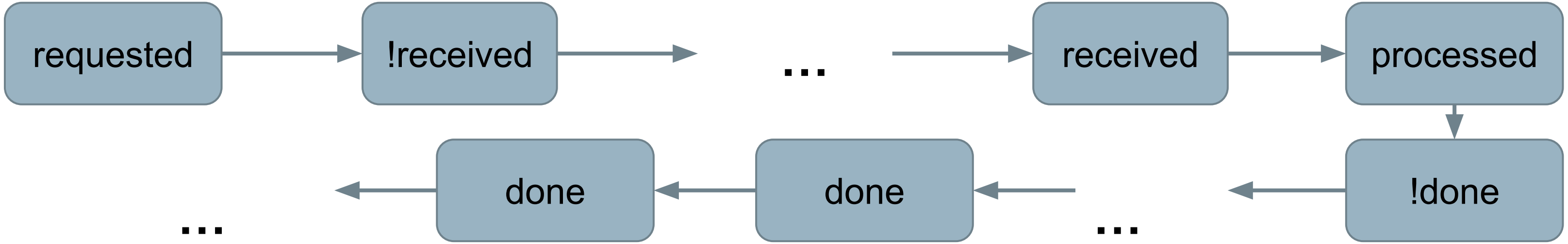
- **U (until)** - One property must be true until the second becomes true.
  - `(lecture==started) -> ((teacher==talking) U (lecture==ended))`
  - `born -> (alive U dead)`
  - `requested -> (!replied U acknowledged)`



More LTL Examples

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

- `G (requested -> F (received))`
- `G (received -> X (processed))`
- `G (processed -> F (G (done)))`
  - `G (requested -> G (!done))` not possible, based on above.



More LTL Examples

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

- `G (requested -> F (received))`
  - **At any point in this timeline**, if the action is requested, the request must eventually be received.
- `X (requested -> F (recieved))`
  - **If a request is made in the next step**, it must eventually be received.
  - A request made **now** or **after the next step** does not have this guarantee.

Computation Tree Logic Formulae

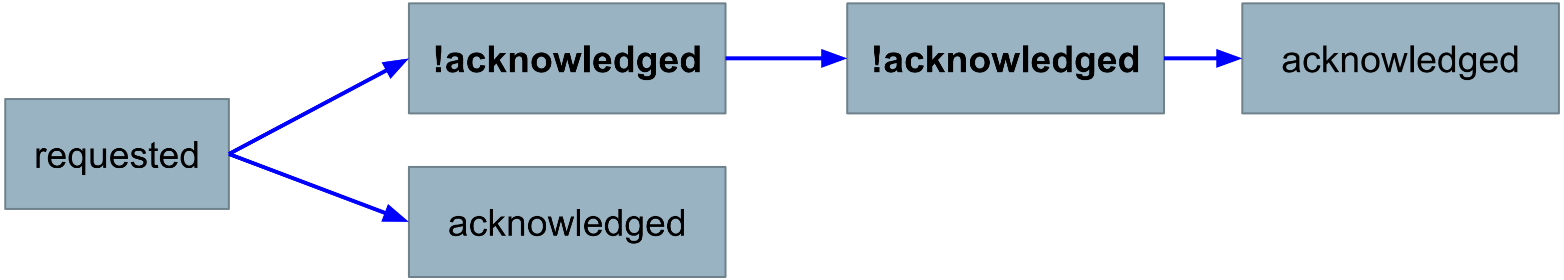
Combines multi-path quantifiers (A,E) with path-specific quantifiers:

A (all)	Affects **all paths** branching out from the current state.
E (exists)	Affects **at least one path** branching out from the current state.

X (next)	X (weather == rain)	In the next state on this path, it will be raining.
G (globally)	G (weather == rain)	Now and in all future states on this path, it will be raining.
F (finally)	F (weather == rain)	Eventually on this path, there will be a state where it is raining.
U (until)	(weather == rain) U (temperature < 0)	On this path, it will rain until the temperature drops below 0. (The temperature must eventually be less than 0)
W (weak until)	weather == rain) W (temperature < 0)	On this path, it will rain until the temperature drops below 0. (The temperature could remain above 0 forever)

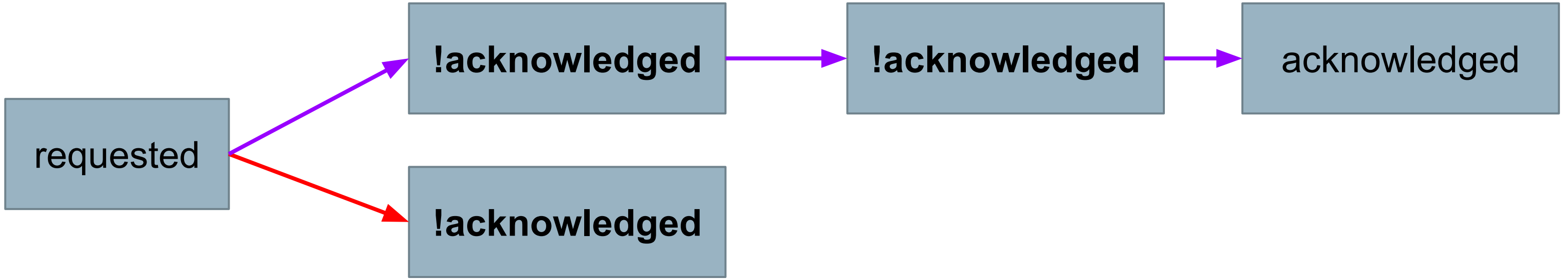
CTL Examples

- **requested**: a request has been made
- **acknowledged**: request has been acknowledged.
  - `AG (requested -> AF acknowledged)`
    - On all paths, at every state in the path (AG)
    - **If** a *request* is made, then for **all paths starting at that point**, eventually (AF), it must be *acknowledged*.



CTL Examples

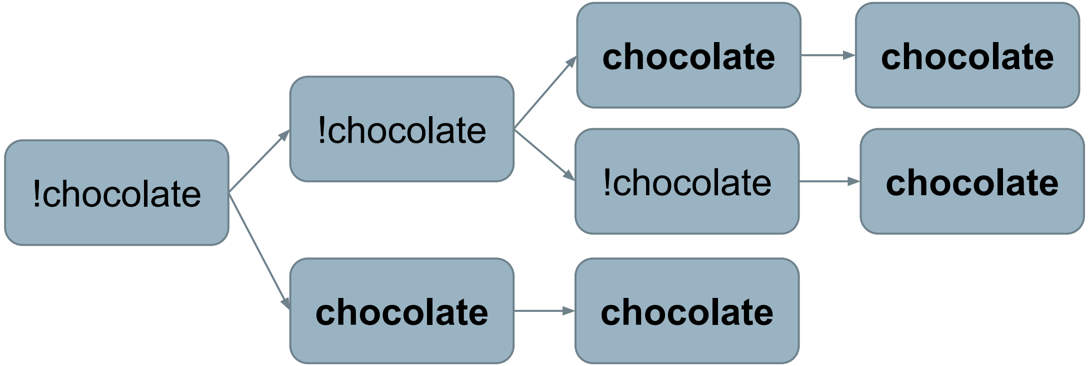
- **requested**: a request has been made
- **acknowledged**: request has been acknowledged.
  - `AG (requested -> EF acknowledged)`
    - On all paths, at every state in the path (AG)
    - If a *request* is made, then for **a subset of paths starting at that point**, eventually (EF), it must be *acknowledged*.



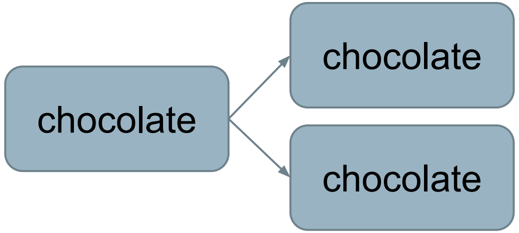
CTL Examples

chocolate = “I like chocolate.”

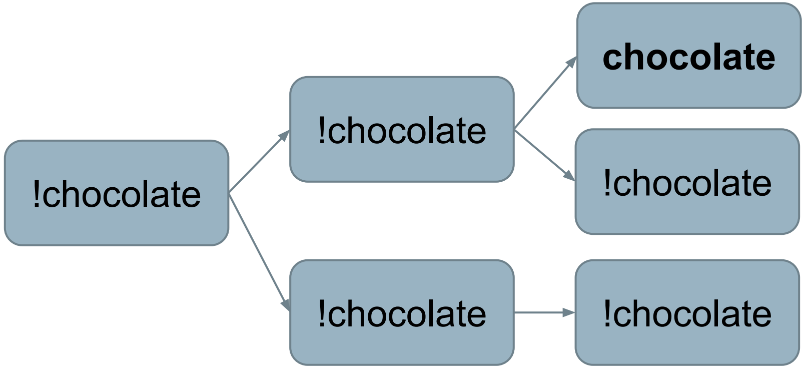
- `AF (EG chocolate)`



- `AG chocolate`



- `EF chocolate`

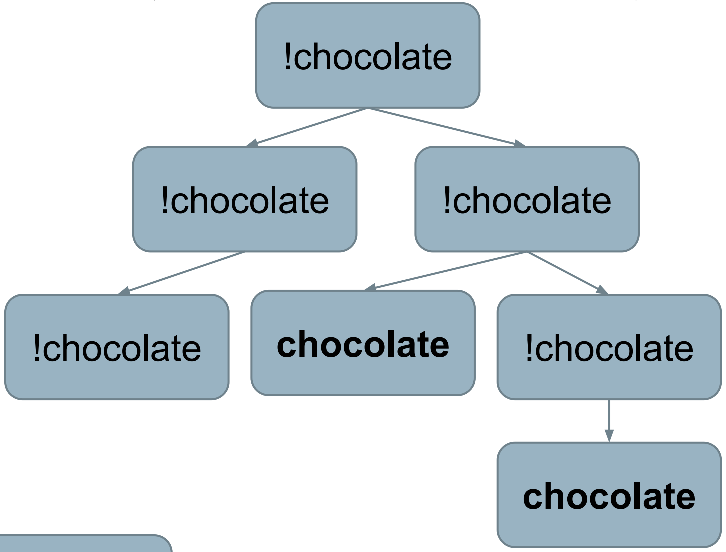


CTL Examples

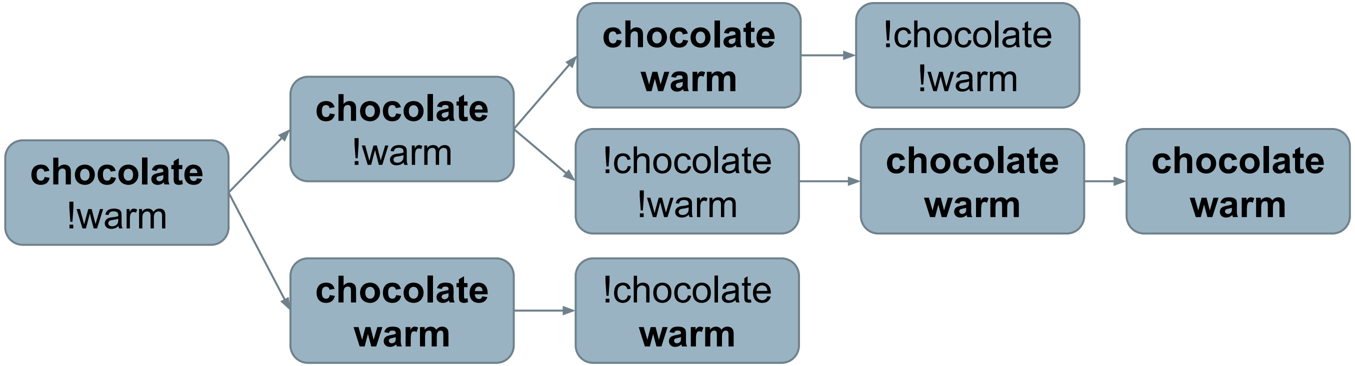
chocolate = “I like chocolate.”

warm = “it is warm”

- EG (AF chocolate)



- AG (chocolate U warm)



Example - Elevator

- If the cabin is moving, the direction is up, and it is on floor 3, then it will be at floor 4 next.
  - `G (((floor==3) && (status==moving) && (direction==up)) -> X (floor==4))`

- If I request the elevator on floor 1, and the cabin is not at that floor, it must eventually reach me (or be broken).
  - `AG ((request_floor1 && floor!=1) -> AF (floor==1 || status==broken))`

Example - Elevator

- If the elevator is requested on floor 1, and the cabin is at floor 4, it ***could*** stop at floor 3 along the way to let passengers in.
  - `AG ((request_floor1 && floor==4) -> EX (floor==3 && door==open))`
  - Leaves open possibility that the cabin is moving up, could break, could remain at floor 4 longer, no one requested it on floor 3, …

- The door must not be open while cabin moving.
  - `G (status==moving -> door==closed)`

Let’s Take a Break

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.

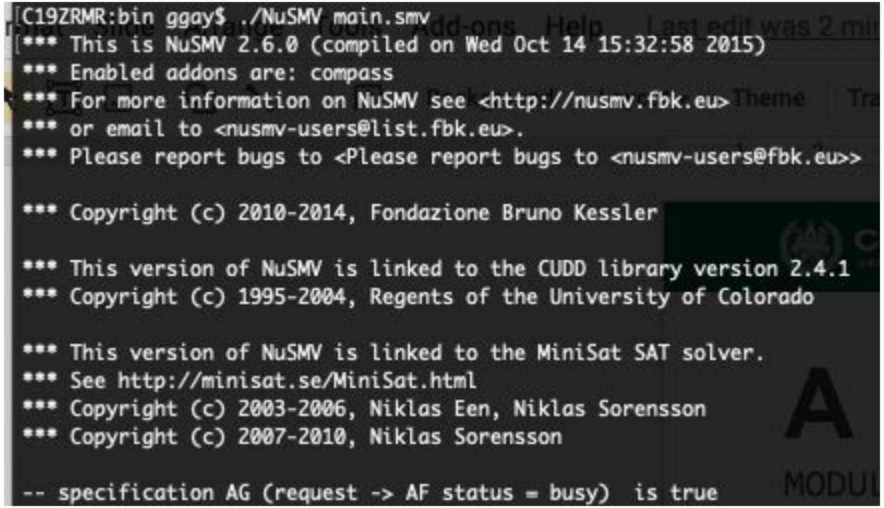
Expressions define how the state of each variable can change.

"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:
  - 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.

```
-- 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
```

```
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;
```

CHALMERS UNIVERSITY OF TECHNOLOGY

- 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 - Example

- Safety Property
  - A specific event/sequence happens as specified.
- 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.
  - `AG (pedestrian_light = walk -> traffic_light != green)`

Activity - Example

- Liveness Property
  - **Eventually** something of interest happens.
- `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

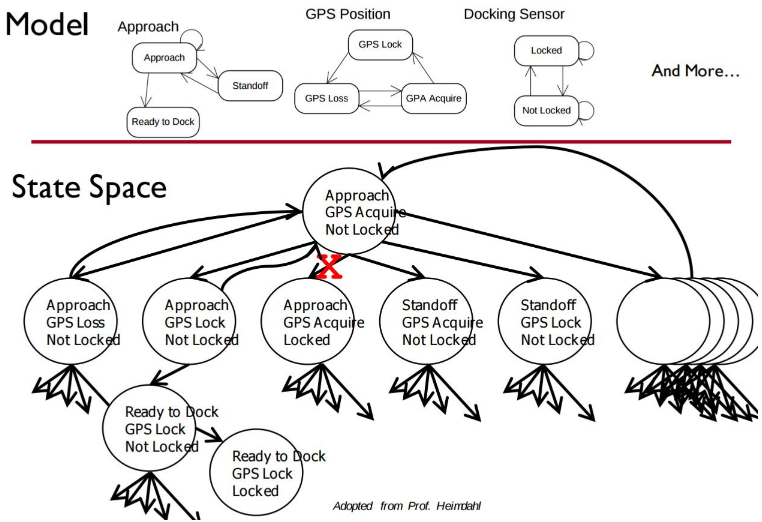
- Search state space for property violations.
- Violations give us counter-examples
  - Path that demonstrates the violation.
  - (useful test case)
- Implications of counter-example:
  - Property is incorrect.
  - Model does not reflect expected behavior.
  - Real issue found in the system being designed.

Test Generation from FS Verification

- We can also take **properties we know to be true** and **negate** them.
  - Called a "**trap property**" - we assert that a known property can never be met.
- Produces a counterexample showing the property can be met.
  - Can be used as a test for the real system.
  - Demonstrates 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, ¬, ∨
  - Clauses are joined with ∧

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

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.

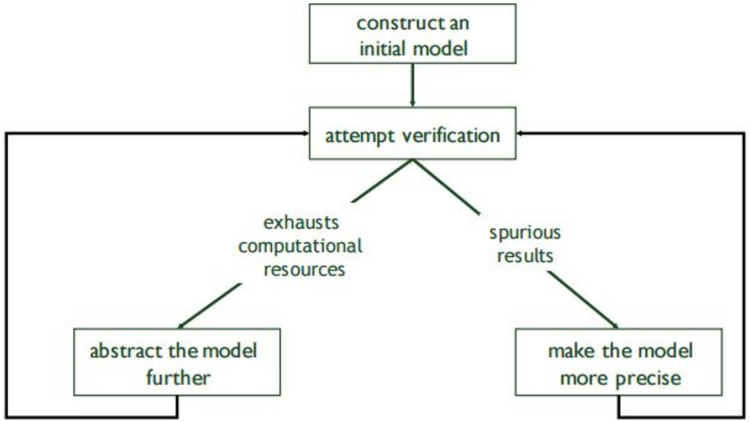
DPLL Algorithm

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

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.
- **Amazon Web Services**
  - Used to verify security policies, stateful behaviors.
  - Used to verify LLM correctness.
- 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
- Lec 15: Automated Test Generation
- Lec 16: Course Review (Practice Exam)

- Assignment 4 - Questions?

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY