Proving the Shalls:

Requirement Analysis and Verification

CSCE 740 - Lecture 10 - 09/28/2015

How do we know that the software will work?

(AKA: How do we know that our specification is correct?)

(Also… free of contradictions and complete)

The Power of Argument

- Once the software is complete, we perform verification (does the software meet the requirements?).
    - We **argue** that the software is correct.
    - We **argue** that the software meets the users’ needs.
- Before we build the software, we want to know that the specifications are complete, correct, and not contradictory.
- How can we analyze the specification without code?

Behavior Modeling

- **Abstraction** - simplifying a problem by identifying important aspects, focusing on those, and pretending other details don’t exist.
- The key to solving **many** computing problems.
  - Solve a simpler version, then apply to the big problem.
- Don’t have code? A design? Hardware? Ignore those and focus on the core behavior.

Behavior Modeling

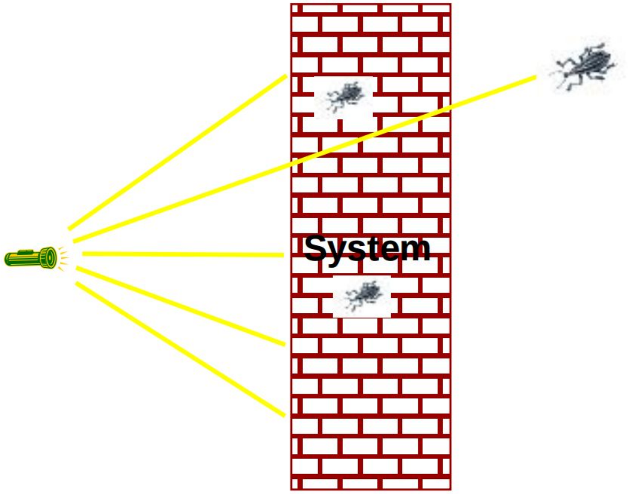
- Requirements analysis can be performed by modeling behavior as **state machines**.
    - Input causes the system to change state (transition).
    - Use the requirements to develop a model of how the system responds to different types of input when performing a function.
- Not as complex as the real code (states summarize *types of responses*).
- Can be “executed”.

So, You Want to Perform Verification...

- You have a property that you want your program to obey (i.e., a requirement).
- 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.

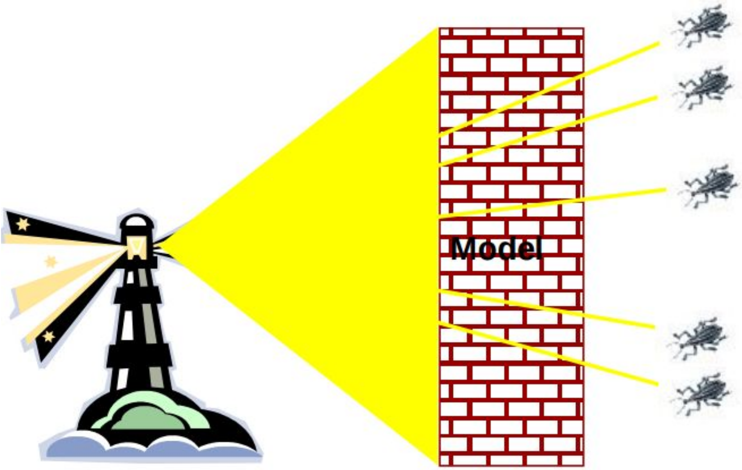
Testing

- 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?

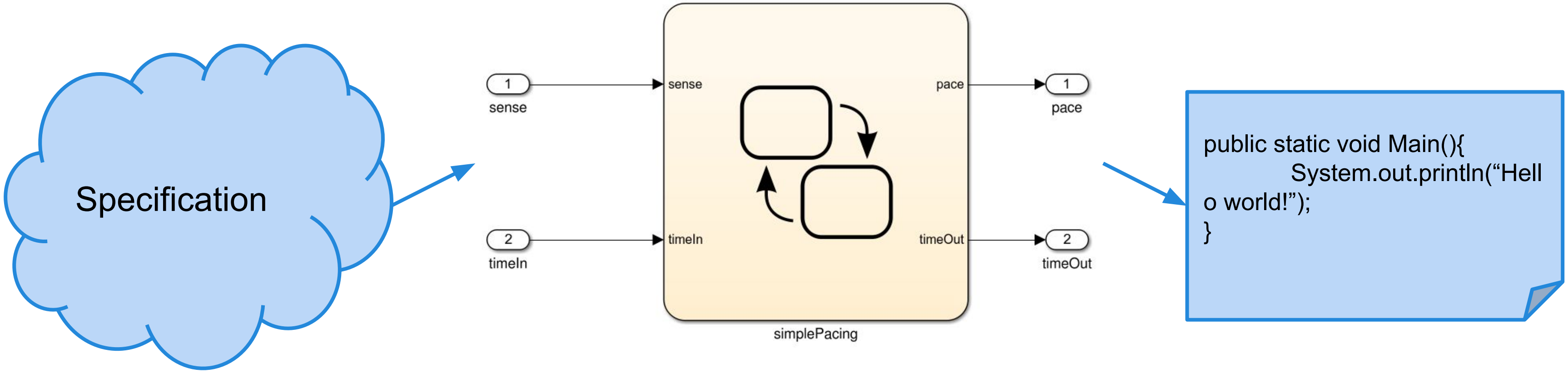


Finite-State Verification

- 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.



What Can We Do With This Model?



```
public static void Main(){
            System.out.println(“Hell
o world!”);
}
```

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.

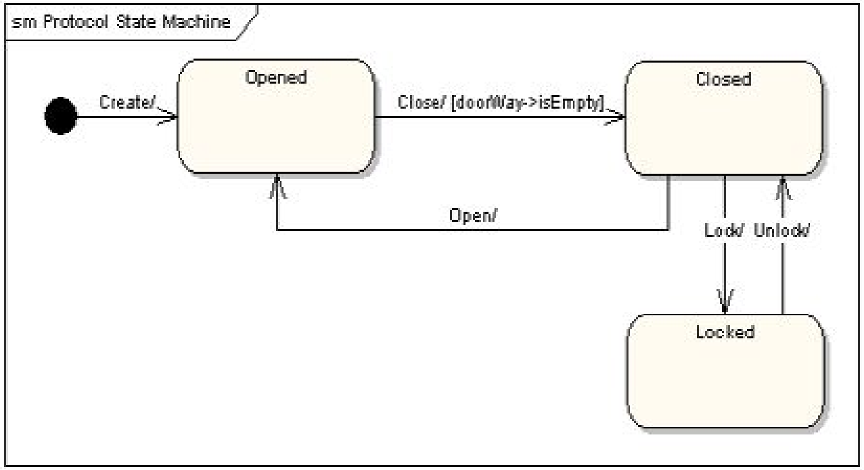
Today's Goals

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

State Machine Models

Finite State Machines

- A common method of modeling behavior of a system.
- A directed graph: nodes represent states, edges represent transitions.
- Not a substitute for a program, but a way to explore functionality.
    - Typically build a model for each major feature.



Some Terminology

- **Event -** Something that happens at a point in time.
    - Operator presses a self-test button on the device.
    - The alarm goes off.
- **Condition** - Describes a property that can be true or false and has duration.
    - The fuel level is high.
    - The alarm is on.
- **State** - An abstract description of the current value of an entity's attributes.
    - The controller is in the "self-test" state after the self-test button has been pressed, and leaves it when the rest button has been pressed.
    - The tank is in the "too-low" state when the fuel level is below the set threshold for N seconds.

States, Transitions, and Guards

- **State** - An abstract description of the current value of an entity’s attributes.
- States change in response to events.
    - A state change is called a **transition**.
- When multiple responses to an event (transitions triggered by that event) are possible, the choice is guided by the current conditions.
    - These conditions are also called the **guards** on a transition.

State Transitions

Transitions are labeled in the form:

`event [guard] / activity`

- `event`: The event that triggered the transition.
- `guard`: Conditions that must be true to choose this transition.
- `activity`: Behavior exhibited by the object when this transition is taken.
- All three are optional.
  - Missing Activity: No output from this transition.
  - Missing Guard: Always take this transition if the event occurs.
  - Missing Event: Take this transition immediately.

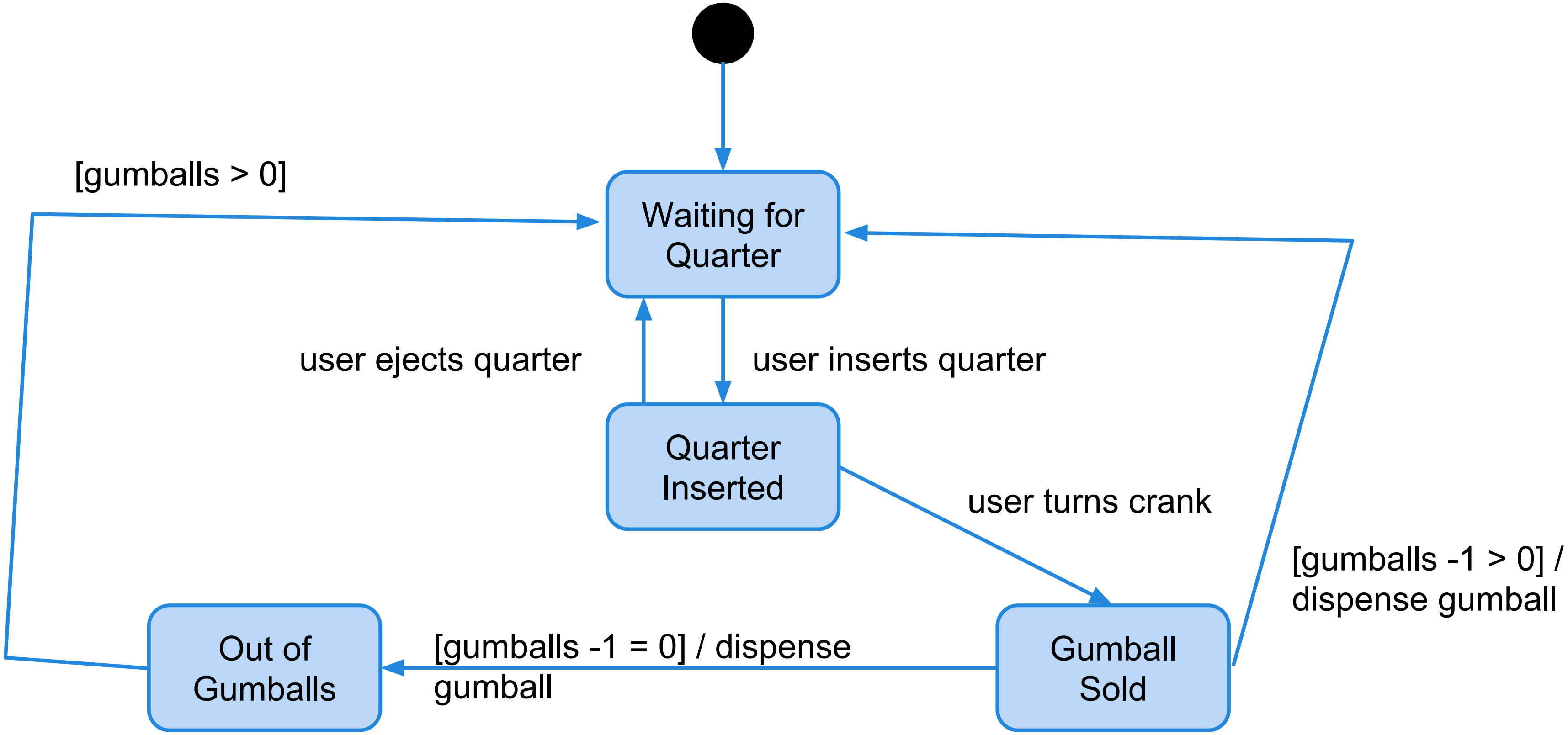
State Transition Examples

Transitions are labeled in the form:

`event [guard] / activity`

- The controller is in the “self-test” state after the self-test button has been pressed, and leaves it when the rest button has been pressed.
  - Pressing self-test button is an **`event`**.
- The tank is in the “too-low” state when the fuel level is below the set threshold for N seconds.
  - Fuel level below threshold for N seconds is a **`guard`**.

Example: Gumball Machine



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.
    - (sad && !rich) -> X(sad)
    - ((x==0) && (add3)) -> X(x == 3)
- 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 always be true.
  - 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 the above are true, can this be true?
  - G (requested) && G (!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.

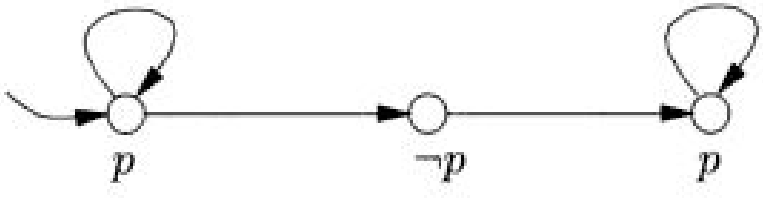
X (next)	X hunger	In the next state on this path, I will be hungry.
G (globally)	G hunger	In all future states on this path, I will be hungry.
F (finally)	F hunger	Eventually on this path, there will be a state where I am hungry.
U (until)	hunger U burger	On this path, I will be hungry until I start 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

Proving Properties

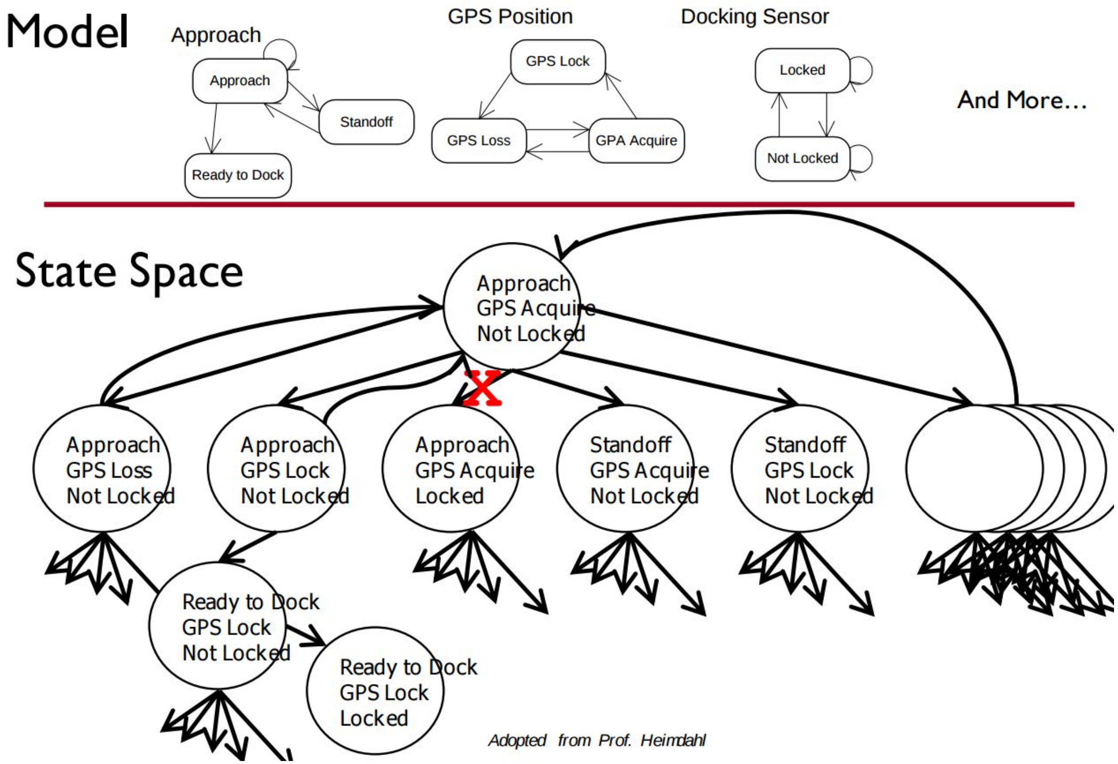
- To perform verification, we take properties and exhaustively search the state space of the model for violations.
- Violations give us counter-examples
  - A path that demonstrates how the property has been violated.
- Implications:
  - Property is incorrect.
  - Model does not reflect expected behavior.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



Adopted from Prof. Heimdahl

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 = (!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.

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 applies.

Branch & Bound Algorithm

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

1. **Set x1 to false.**
   ```
   f = (!x2 || x5) && (0 || !x3 || x4) && (x4 || ! x5) && (0 || x2)
   ```
2. **Set x2 to false.**
   ```
   f = (1 || x5) && (0 || !x3 || x4) && (x4 || ! x5) && (0 || 0)
   ```
3. **Backtrack and set x1 to true.**
   ```
   f = (0 || x5) && (0 || !x3 || x4) && (x4 || ! x5) && (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 Properties

To be useful, a model must be:

- Compact
    - Models must be simplified enough to be analyzed.
    - Depends on how it will be used.
- Predictive
    - Represent the real system well enough to distinguish between good and bad outcomes of analyses.
    - No single model usually represents all characteristics of the system well enough.

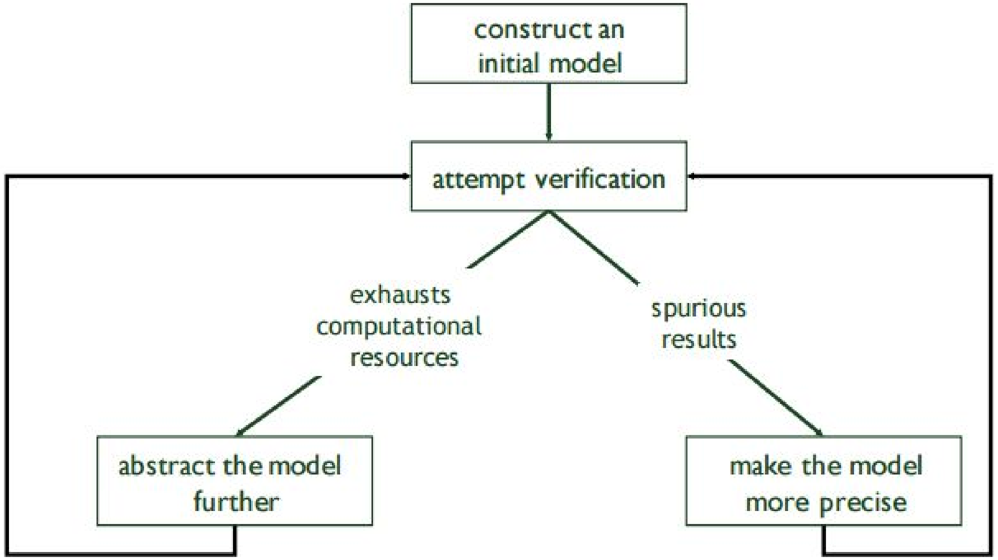
Model Properties

To be useful, a model must be:

- Meaningful
  - Must provide more information than success and failure.
- General
  - Models must be practical for use in the domain of interest.
  - An analysis of C programs is not useful if it only works for programs without pointers.

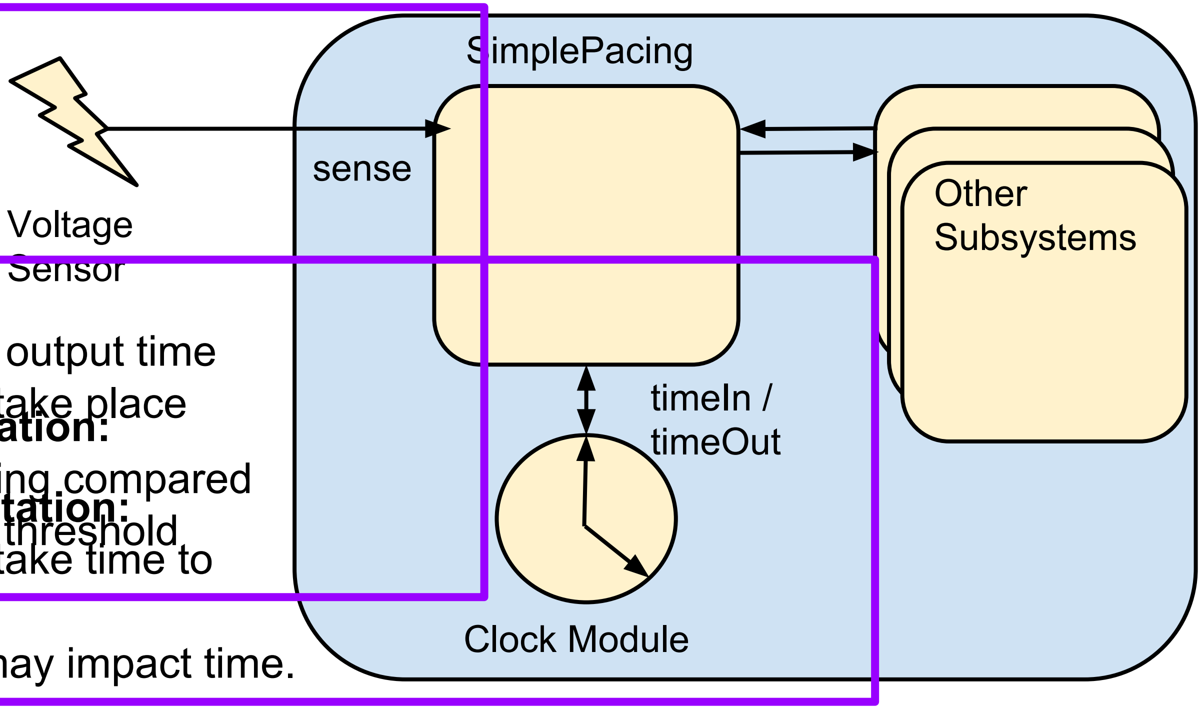
Model Refinement

- 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.



Challenge - Does the Model Match the Program?

Models require abstraction. Useful for requirements analysis, but may not reflect operating conditions.



In the model:

- Binary input

In the implementation:

- Voltage reading compared to calculated threshold

In the model:

- input time = output time
- Operations take place instantly.

In the implementation:

- Operations take time to compute.
- Clock drift may impact time.

We Have Learned

- We can analyze our specifications by creating simplified models of the system and proving that 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 specifications are correct (or fix them if they are not).
- We can also generate test cases from the model to demonstrate that properties still hold over the final system.

Next Time

- Design Fundamentals

- Readings:
  - Sommerville, chapter 6

- Homework:
  - Up on Moodle
  - Revised requirements and tests due 10/02.
  - Any questions?