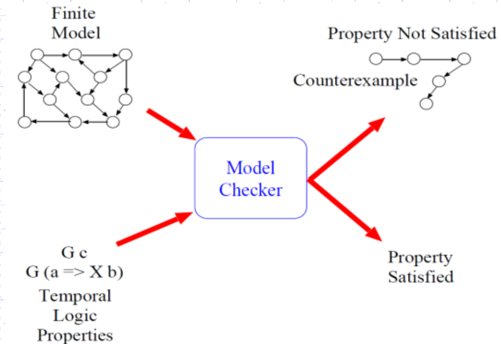
CS6133 Fall 2011
Software Specification and Verification

Lecture 5

Model Checking

Model Checking



What is Model Checking

- Model checking is a method for formally verifying finite-state concurrent systems.
  - Finite models are Kripke structures
  - Specifications (i.e., desired properties) about the system are expressed as temporal logic formulas
  - Model checking algorithms traverse the model and check if the specification holds against the model
  - A counterexample is a trace of the system that violates the property

Why Use Model Checking

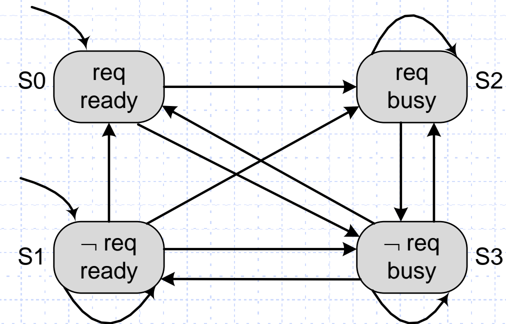
- Model checking is fully automated
  - No proofs have to be devised manually
  - No human interaction is needed
- The counterexample can help you to debug your model or property specification
- The model can be partial
- However, model checking suffers from state explosion problem : many techniques have been developed to alleviate this problem

Kripke Structure

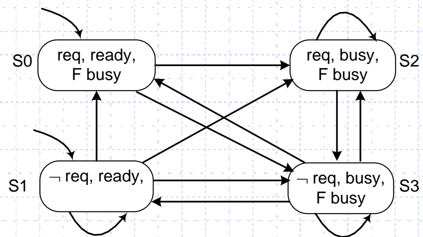
- A Kripke structure is a labeled state-transition graph
- Formally, let AP be a set atomic propositions. A finite Kripke structure over AP is a 4 tuple

  M = <S, S_0, R, L>, where
  - S is the finite set of states
  - $S_0 \subseteq S$ is the set of initial states
  - $R \subseteq S \times S$ is the transition relation
  - $L: S \rightarrow 2^{AP}$ is the labeling function maps each state to a set of atomic propositions true in that state

Kripke Structure Example

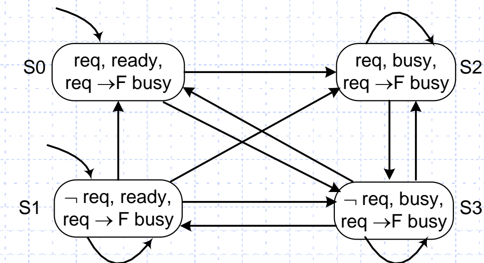


Basic Model Checking Algorithm



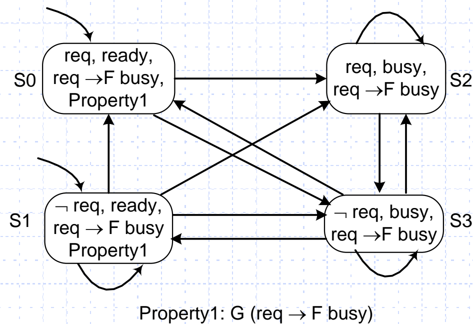
Property1: G (req → F busy)

Basic Model Checking Algorithm



Property1: G (req → F busy)

Basic Model Checking Algorithm



Property1: G (req → F busy)

What is NuSMV

- NuSMV (New Symbolic Model Verifier) is an open source software system
- NuSMV or SMV is a symbolic model checker, which provides a language for describing models
- SMV language: Communicating finite state machines
  - Finite state machines with variables of finite types
  - Finite state machines execute concurrently (in parallel or interleaving)
- Properties are specified as Temporal Logic formulas

SMV Module

- A SMV model consists of one or more modules, one of the modules must be called main as in C or Java
- Each module has variable declarations and assignment statements
- Assignments can be of the following forms
  - DEFINE
    x:= y+5;
  - ASSIGN
    init(x) := 0;
    next(x) := y+5;

SMV Module Composition

- Synchronous composition (parallel)
  - All modules' assignments are executed in parallel
  - A single step of the resulting model corresponds to a step in each of the components
- Asynchronous composition (interleaving)
  - A step of the composition is a step by exactly one process (assignments within the process execute in parallel)
  - Variables that are not assigned in that process are left unchanged

Example Input to NuSMV

```
MODULE main
VAR
    request : boolean;
    status : {ready, busy};
ASSIGN
    init(status) := ready;
    next(status) := case
        request : busy;
        1 : {ready, busy};
    esac;
LTLSPEC
    G(request → F(status =busy))
```


SMV Variables

- Variables must be of finite types
- Variables can be boolean, enumerated types, integer subranges, user-defined modules, or an array of any of these
  - var1 : {on, off};
  - var2 : array 2..5 of {on, off};
  - var3 : array 1..10 of array 2..5 of boolean;
  - var4 : Inverter(1); --Inverter() is a module
  - var5 : array on..off of boolean; -- error
  - var6 : {enabled, active, off}; -- error
  - var7 : 0..6;

Variables

- Enumerated values must be distinct from each other
- Boolean values
  - Constants are 0 and 1 (FALSE and TRUE)
  - Operations are:
    ~ (not),
    & (and),
    | (or), ^ (xor),
    -> (implies), <-> (iff)
- Array subscripts must evaluate to integers

Variables

- Each state variable is
  - Either controlled by the system, i.e., SMV model explicitly assigns values to the variable

```
VAR
    x: 1..20;
ASSIGN
    init(x) := 0;
    next(x) := x+2;
```

  - Or controlled by the environment, i.e., input variable

```
VAR
    y: boolean;
```

Derived Variables

- Each derived variable defines a macro

```
VAR
    p: boolean;
    q: boolean;
DEFINE
    z:= p ->q;
```

each occurrence of z is replaced by p ->q

- Derived variables don't contribute to the state space

Execution Model

- Round-based execution: initialization round followed by a sequence of update rounds
- In each round
  - Environment-controlled variables non-deterministically change values
  - System executes its parallel assignment statements to state variables
  - Result is a new system state (an assignment of values to state variables)

Example Input to NuSMV

```
MODULE main
VAR
    request : boolean;
    status : {ready, busy};
ASSIGN
    init(status) := ready;
    next(status) := case
        request : busy;
        1 : {ready, busy};
    esac;
LTLSPEC
    G(request → F(status =busy))
```


Statements

- Simple parallel assignments, e.g.,

```
next (x) := 1;
next (y) := x+1;
```

- Conditional assignments
  - If-then-else

```
next(y) := (a | b) ? x :  x+1;
```

  - Case

```
next(k) := case
  c1: x;
  c2: y;
  1: 0;
esac;
```

Case Statements

- Case statements are evaluated sequentially
- If the guard condition on the left hand side evaluates to true, the case statement returns the value of the corresponding expression on the right hand side
- If none of the guard condition evaluates to true, it returns numeric value 1
- Guards may not be mutually exclusive

Nondeterminism

- Non-determinism occurs while more than one result is possible
- Unassigned variable: model unconstrained input
- Non-deterministic assignment, e.g.,

  next (x) := {1, 2, 3, 4};
- Undefined assignment of a variable may take on any value in its type, e.g.,

  next (x) := c1? y;

  or conjunction of case guard expressions is not a tautology

Common Errors

- Single Assignment Rule: only have one assignment statement for each variable, e.g., the following is invalid

```
next(x) := y;
next(x) := z;
```

or

```
x := y + 5;
next(x) := z;
```


Common Errors

- Cannot have a cycle of dependencies all of whose assignments take effect simultaneously. That is, the system can't have a set of variables whose next values depends on the next values of each other and vice versa. e.g.,

```
next(x) := next(y);
next(y) := next(z);
next(z) := next(x);
```