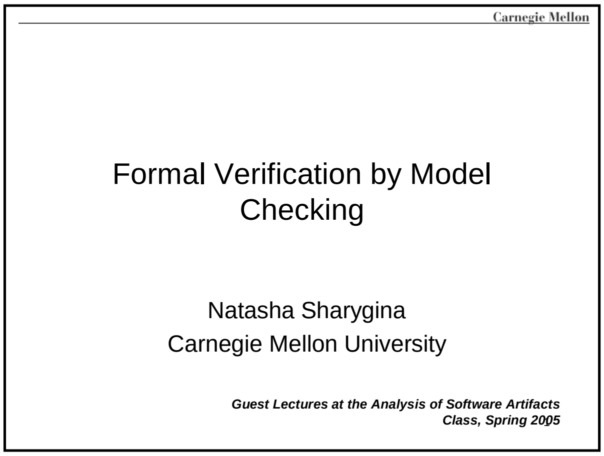
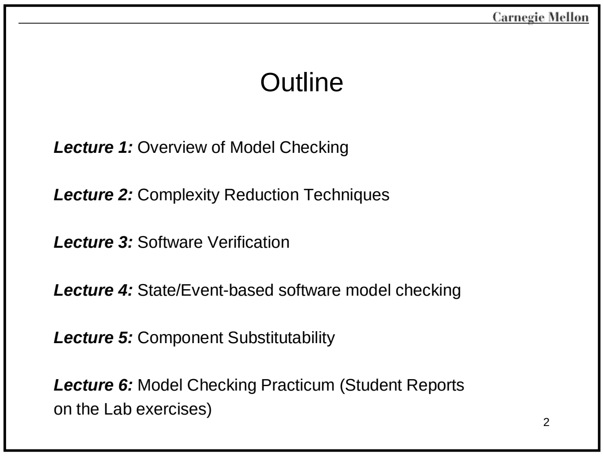


Formal Verification by Model Checking

Natasha Sharygina
Carnegie Mellon University

Guest Lectures at the Analysis of Software Artifacts Class, Spring 2005



Outline

Lecture 1: Overview of Model Checking

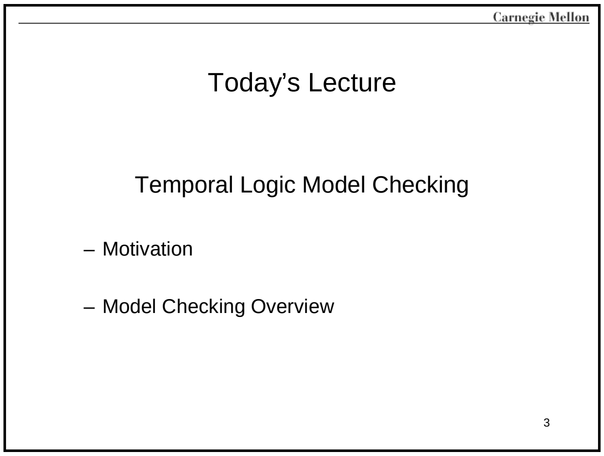
Lecture 2: Complexity Reduction Techniques

Lecture 3: Software Verification

Lecture 4: State/Event-based software model checking

Lecture 5: Component Substitutability

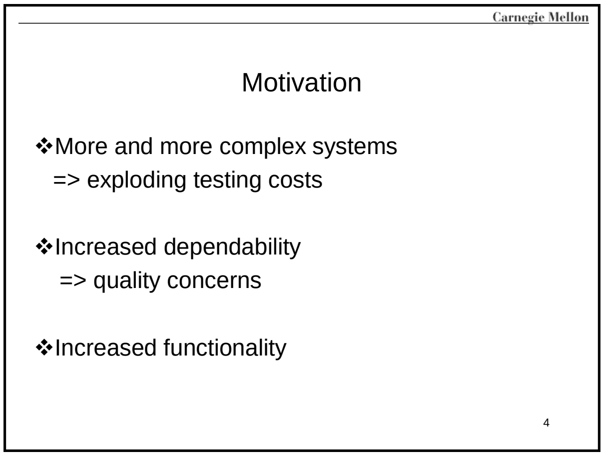
Lecture 6: Model Checking Practicum (Student Reports on the Lab exercises)



Today’s Lecture

Temporal Logic Model Checking

- Motivation
- Model Checking Overview

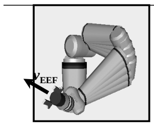


Motivation

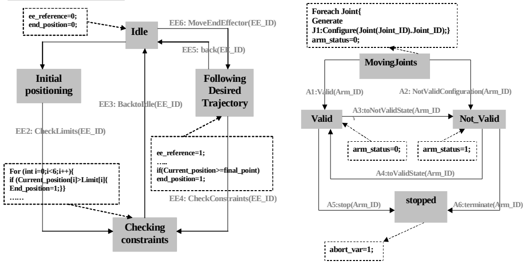
❖More and more complex systems
=> exploding testing costs

❖Increased dependability
=> quality concerns

❖Increased functionality



NASA Robot Controller



```
ee_reference=0;
end_position=0;
```

```
For (int i=0;i<6;i++){
if (Current_position[i]>Limit[i]{
End_position=1;}}
……
```

```
ee_reference=1;
…..
if(Current_position>=final_point)
end_position=1;
```

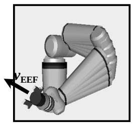
```
Foreach Joint{
Generate
J1:Configure(Joint(Joint_ID).Joint_ID);}
arm_status=0;
```

```
arm_status=0;
```

```
arm_status=1;
```

```
abort_var=1;
```

EndEffector || ***Arm***



Verification Properties

- ***Configuration Validity Check***

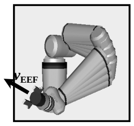
  If the *EndEffector* is in the “FollowingDesiredTrajectory” state, then the *Arm* is in the ‘Valid” state

  Always((ee_reference=1) ->(arm_status=1))

- ***Safety***

  If the *EndEffector* reaches an undesired position, then the program terminates prior to a new move of the *EndEffector*

  AfterAlwaysUntil(undesired_position =1,ee_reference=1,abort_var=1)



Verification Properties

- ***Configuration Validity Check (FALSE)***

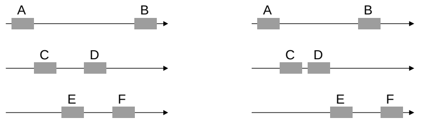
  If the *EndEffector* is in the “FollowingDesiredTrajectory” state, then the *Arm* is in the ‘Valid” state

  Always((ee_reference=1) ->(arm_status=1))

Why is this hard?

– Generating all possible paths of execution is hard; ask any testing expert.

– In particular, forcing a collection of (potentially distributed) processes to execute in all possible relative orders is notoriously difficult.



Motivation

What are the problems with the analysis techniques you have learned so far?

- not exhaustive (missed behaviors)
- not all are automated (manual reviews, manual testing)
- do not scale (large programs are hard to handle)
- no guarantee of results (no mathematical proofs)
- concurrency problems

Need for automated, exhaustive and mathematically sound analysis!

Formal Verification by Model Checking

Domain: Continuously operating concurrent systems (e.g. operating systems, hardware controllers and network protocols)

- Ongoing, reactive semantics
  - Non-terminating, infinite computations
  - Manifest non-determinism

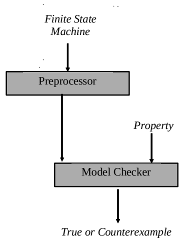
Instrument: Temporal logic [Pnueli 77] is a formalism for reasoning about behavior of reactive systems

Temporal Logic Model Checking

[Clarke,Emerson 81][Queille,Sifakis 82]

- Systems are modeled by **finite state machines**
- **Properties** are written in **propositional temporal logic**
- Verification procedure is an **exhaustive search of the state space** of the design
- **Diagnostic counterexamples**

Temporal Logic Model Checking



Finite State Machine

Preprocessor

Property

Model Checker

True or Counterexample

What is Model Checking?

Does model M satisfy a property P ?
(written M |= P)

What is “M”?

What is “P”?

What is “satisfy”?

What is “M”?

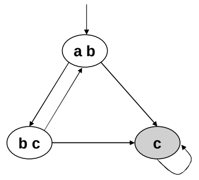
States: valuations to all variables

Initial states: subset of states

Arcs: transitions between states

Atomic Propositions:
e.g. x = 5, y = true

Observation (color):
Valuation to all atomic propositions



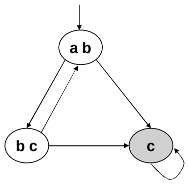
State Transition Graph or Kripke Model

What is “M”?

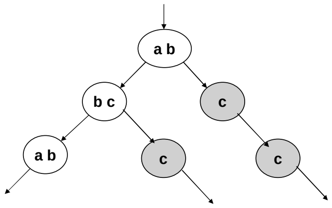
$M = \langle W, I, R, L, \Gamma \rangle$

W - set of states
$I \subseteq W$ – set of initial states
$R \subseteq W \times W$ - set of arcs
L - set of atomic propositions
$\Gamma : W \rightarrow 2^L$ – mapping from states to colors

Model of Computation



State Transition Graph

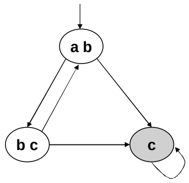


Infinite Computation Tree

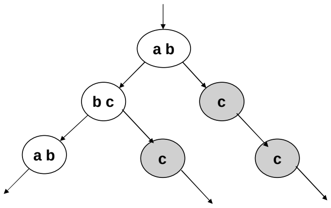
Unwind State Graph to obtain Infinite Tree.

A trace is an infinite sequence of states.

Semantics



State Transition Graph



Infinite Computation Tree

The semantics of a FSM is a set of traces. Semantics of the composition of FSMs is the intersection of traces of individual FSMs.

What is “P”?

Different kinds of temporal logics

Syntax: What are the formulas in the logic?

Semantics: What does it mean for model **M** to satisfy formula **P**?

Formulas:

- Atomic propositions: properties of states
- Temporal Logic Specifications: properties of traces.

Computation Tree Logics

Examples: ***Safety*** (mutual exclusion): no two processes can be at a critical section at the same time

Liveness (absence of starvation): every request will be eventually granted

Temporal logics differ according to how they handle branching in the underlying computation tree.

In a linear temporal logic (LTL), operators are provided for describing system behavior along a single computation path.

In a branching-time logic (CTL), the temporal operators quantify over the paths that are possible from a given state.

Computation Tree Logics

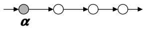
Formulas are constructed from ***path quantifiers*** and ***temporal operators:***

1. ***Path Quantifiers:***
   - **A** – ‘‘for every path’’
   - **E** – ‘‘there exists a path’’
2. ***Temporal Operator:***
   - $\mathbf{X}\alpha$ - α holds **next** time
   - $\mathbf{F}\alpha$ - α holds sometime in the **future**
   - $\mathbf{G}\alpha$ - α holds **globally** in the **future**
   - $\alpha \mathbf{U} \beta$ - α holds **until** β holds

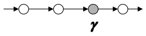
The Logic LTL

Linear Time Logic (LTL) [Pnueli 77]: logic of temporal sequences.

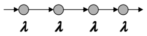
- **AXα: α holds in the next state**



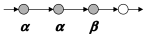
- **AFγ: γ holds eventually**



- **Aλ: λ holds from now on**

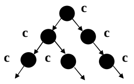


- ***α* U *β*: *α holds until β holds***

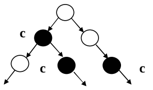


The Logic CTL

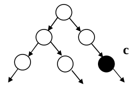
In a branching-time logic (CTL), the temporal operators quantify over the paths that are possible from a given state (s_0).



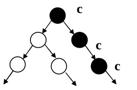
$M, s_0 \models AG\ c$



$M, s_0 \models AF\ c$



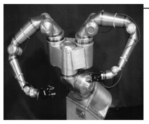
$M, s_0 \models EF\ c$



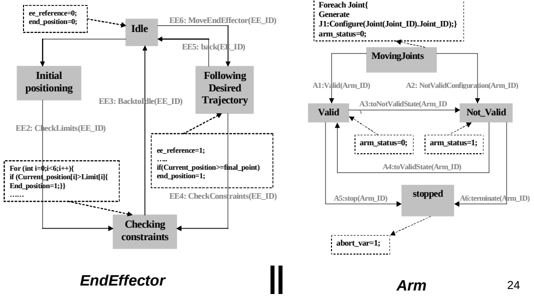
$M, s_0 \models EG\ c$

Typical CTL Formulas

- **EF** (*Started* $\wedge \neg$ *Ready*): it is possible to get to a state where *Started* holds but *Ready* does not hold.
- **AG** (*Req* $\Rightarrow$ **AF** *Ack*): if *Request* occurs, then it will be eventually *Acknowledged.*
- **AG** (*DeviceEnabled*): *DeviceEnabled* holds infinitely often on every computation path.
- **AG** (**EF** *Restart*): from any state it is possible to get to the *Restart* state.



Robot Controller System



```
ee_reference=0;
end_position=0;
```

```
For (int i=0;i<6;i++){
if (Current_position[i]>Limit[i]{
End_position=1;}}
……
```

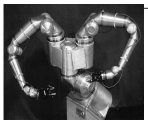
```
ee_reference=1;
…..
if(Current_position>=final_point)
end_position=1;
```

```
Foreach Joint{
Generate
J1:Configure(Joint(Joint_ID).Joint_ID);}
arm_status=0;
```

```
arm_status=0;
```

```
arm_status=1;
```

```
abort_var=1;
```



Examples of the Robot Control Properties

- ***Safety Operation:*** If the *EndEffector* reaches an undesired position, then the program terminates prior to a new move of the *EndEffector*

  AfterAlwaysUntil(undesired_position =1,ee_reference=1,abort_var=1)

- ***Configuration Validity Check:***
  If an instance of *EndEffector* is in the “FollowingDesiredTrajectory” state, then the instance of the corresponding *Arm* class is in the ‘Valid” state

  Always((ee_reference=1) ->(arm_status=1)

- ***Control Termination:*** Eventually the robot control terminates

  EventuallyAlways(abort_var=1)

Linear vs. branching-time logics

some advantages of LTL

- LTL properties are preserved under “abstraction”: i.e., if M “approximates” a more complex model M’, by introducing more paths, then

  $M \models \psi \Rightarrow M' \models \psi$
- “counterexamples” for LTL are simpler: consisting of single executions (rather than trees).
- The automata-theoretic approach to LTL model checking is simpler (no tree automata involved).
- anecdotally, it seems most properties people are interested in are linear-time properties.

some advantages of BT logics

- BT allows expression of some useful properties like ‘reset’.
- CTL, a limited fragment of the more complete BT logic CTL*, can be model checked in time linear in the formula size (as well as in the transition system). But formulas are usually far smaller than system models, so this isn’t as important as it may first seem.
- Some BT logics, like µ-calculus and CTL, are well-suited for the kind of fixed-point computation scheme used in symbolic model checking.

What is “satisfy”?

M satisfies ***P*** if *all* the reachable states satisfy ***P***

Different Algorithms to check if *M* |= *P*.

Common Feature: Exhaustive State Space Exploration!

What is “satisfy”?

Explicit state space exploration:

1) Invariant checking Algorithm.

Invariant:

φ:

atomic propositions (e.g. x =5, b=true)

$\neg\varphi$

$\varphi \wedge \psi$

$\varphi \vee \psi$

satisfaction:

M satisfies φ if all the reachable states satisfy φ

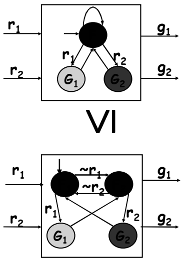
Invariant checking Algorithm

Does M satisfy P?

1. Start at the initial states and explore the states of ***M*** using DFS or BFS.
2. In any state, if ***P*** is violated then print an "error trace".
3. If all reachable states have been visited then say "yes".

Model checking complexity: |*M*|

Refinement



Idea: Implementation ≤ Specification

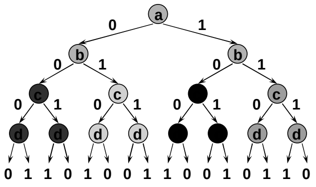
Every behavior of the implementation should be an allowable behavior of the specification.

M ≤ S - Linear Time

- Every trace of M is a trace of S
  "Language Containment"
- Every formula in linear logic (LTL) satisfied by S, is also satisfied by M
- Complexity: |M| * $2^{|S|}$
- In practice can be done using "reachability"

What is "satisfy"? (cont.)

Symbolic State Space Exploration



Idea: to use **Boolean encoding** for state machine and sets of states.

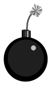
- Compact representation of transition relations. Can handle much larger designs – hundreds of state variables
- BDDs traditionally used to represent Boolean functions

State Space Explosion

Problem:

Size of the state graph can be exponential in size of the program (both in the number of the program *variables* and the number of program *components*)

$$M = M_1 \mid\mid \ldots \mid\mid M_n$$



If each M_i has just 2 local states, potentially 2^n global states

Research Directions: State space reduction (next class)

Model Checking Performance

- Model Checkers today can routinely handle systems with between 100 and 300 state variables.

- Systems with 10^{120} reachable states have been checked.

- By using appropriate abstraction techniques, systems with an essentially **unlimited number of states** can be checked.

Notable Examples

- **IEEE Scalable Coherent Interface** – In 1992 Dill's group at Stanford used **Murphi** to find several errors, ranging from uninitialized variables to subtle logical errors

- **IEEE Futurebus** – In 1992 Clarke's group at CMU found previously undetected design errors

- **PowerScale multiprocessor** (processor, memory controller, and bus arbiter) was verified by Verimag researchers using CAESAR toolbox

- **Lucent telecom. protocols** were verified by FormalCheck – errors leading to lost transitions were identified

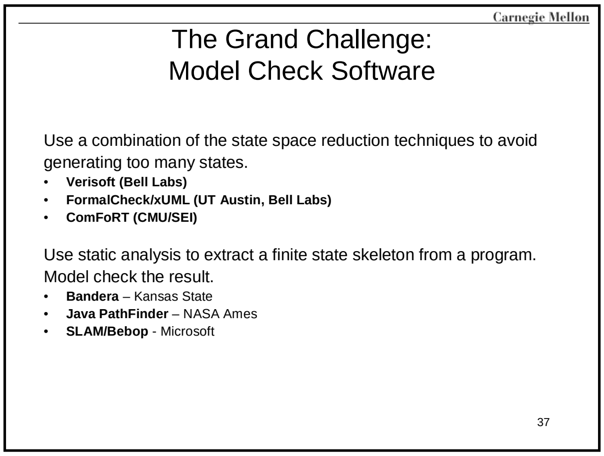
- **PowerPC 620 Microprocessor** was verified by Motorola's Verdict model checker.

The Grand Challenge:
Model Check Software

Extract finite state machines from programs written in conventional programming languages

Use a finite state programming language:
- executable design specifications (Statecharts, xUML, etc.).

Unroll the state machine obtained from the executable of the program.



The Grand Challenge:
Model Check Software

Use a combination of the state space reduction techniques to avoid generating too many states.

- **Verisoft (Bell Labs)**
- **FormalCheck/xUML (UT Austin, Bell Labs)**
- **ComFoRT (CMU/SEI)**

Use static analysis to extract a finite state skeleton from a program. Model check the result.

- **Bandera** – Kansas State
- **Java PathFinder** – NASA Ames
- **SLAM/Bebop** - Microsoft