# Verification and Abstraction (VA 2015)

Giorgio Delzanno

DIBRIS, Università di Genova

# What is Computer Aided Verification?

- Automated verification method aimed at finding bugs in hardware design and software

# What is Computer Aided Verification?

- Automated verification method aimed at finding bugs in hardware design and software
- The basic idea is to exhaustively search for bugs

# What is Computer Aided Verification?

- Automated verification method aimed at finding bugs in hardware design and software
- The basic idea is to exhaustively search for bugs
- Particularly useful for verification of concurrent and reactive systems

# What to verify?

- We have to specify the behavior of the system we are considering

# What to verify?

- We have to specify the behavior of the system we are considering
- and the corresponding safety requirements (e.g. nothing bad happens when the program runs)

# Example: Race conditions

Let us consider a multitasking computation where several processes operate on shared data

- A race condition occurs when the result of the computation depends on the order of execution

# Example: Race conditions

Let us consider a multitasking computation where several processes operate on shared data

- A race condition occurs when the result of the computation depends on the order of execution
- They occur frequently in multitasking application (e.g. OS Kernel, multithreaded programs)

# Example: Race conditions

Let us consider a multitasking computation where several processes operate on shared data

- A race condition occurs when the result of the computation depends on the order of execution
- They occur frequently in multitasking application (e.g. OS Kernel, multithreaded programs)
- They are dangerous: we must ensure consistency of shared data

# Example: Race conditions

Let us consider a multitasking computation where several processes operate on shared data

- A race condition occurs when the result of the computation depends on the order of execution
- They occur frequently in multitasking application (e.g. OS Kernel, multithreaded programs)
- They are dangerous: we must ensure consistency of shared data
- They are difficult to find and to reproduce:
  a different execution $\rightarrow$ a possible different instruction order $\rightarrow$ a possible different output

# Back to ... What to verify?

- Absence of race conditions in all possible executions of a concurrent system

# Back to . . . What to verify?

- Absence of race conditions in all possible executions of a concurrent system
- It is a classical problem!

# Back to ... What to verify?

- Absence of race conditions in all possible executions of a concurrent system
- It is a classical problem!

  (Critical section problem, semaphores, etc).

# Example: Critical Section Problem

- $N$ processes compete to use shared resources

# Example: Critical Section Problem

- $N$ processes compete to use shared resources
- Each process has a critical section in its code in which the shared resource is used

## Example: Critical Section Problem

- $N$ processes compete to use shared resources
- Each process has a critical section in its code in which the shared resource is used
- Property to verify = mutual exclusion, i.e., in each execution at most one process is in the critical section

# A Good Solution

1. Mutual Exclusion: at most two processes in critical section

# A Good Solution

1. Mutual Exclusion: at most two processes in critical section
2. Progress: no deadlock

# A Good Solution

1. Mutual Exclusion: at most two processes in critical section
2. Progress: no deadlock
3. Bounded Waiting: no starvation

# A Good Solution

1. Mutual Exclusion: at most two processes in critical section
2. Progress: no deadlock
3. Bounded Waiting: no starvation
4. Typical assumption Fairness: enabled instructions are eventually executed

# Example: Lamport's Bakery Algorithm

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

# Automated Verification

- Automated verification methods like model checking can be applied to verify finite-state models of concurrent systems

# Automated Verification

- Automated verification methods like model checking can be applied to verify finite-state models of concurrent systems
- To obtain a finite-state model: fix the number of processes, bound the domain of variable, use abstractions

How does a model checker work?

- Input:

How does a model checker work?

- Input:
  - system requirements given in form of a finite-state model $M$

How does a model checker work?

- Input:
  - system requirements given in form of a finite-state model $M$
  - a property $\varphi$ (called specification) that the final system is expected to satisfy.

How does a model checker work?

- Input:
  - system requirements given in form of a finite-state model $M$
  - a property $\varphi$ (called specification) that the final system is expected to satisfy.
- Output: yes if $M$ satisfies $\varphi$, otherwise a counterexample

How does a model checker work?

- Input:
  - system requirements given in form of a finite-state model $M$
  - a property $\varphi$ (called specification) that the final system is expected to satisfy.
- Output: yes if $M$ satisfies $\varphi$, otherwise a counterexample
- The counterexample details why the model doesn't satisfy the specification.

# Finite-state Model

- The model represents all possible (abstract) behaviors of our design

# Finite-state Model

- The model represents all possible (abstract) behaviors of our design
- It can be given as a transition system:

# Finite-state Model

- The model represents all possible (abstract) behaviors of our design
- It can be given as a transition system:
  - A finite collection of states $S$

# Finite-state Model

- The model represents all possible (abstract) behaviors of our design
- It can be given as a transition system:
  - A finite collection of states $S$
  - A transition relation $T \subseteq S \times S$ s.t. $T(s, s')$ represents a transition from state $s$ to state $s'$

# Example

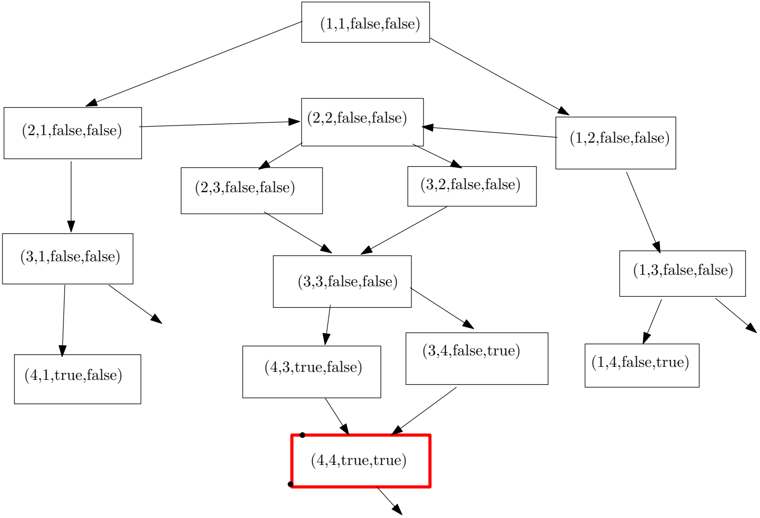
Bool *wantP,wantQ*=false;

| Proc P= | Proc Q= |
|---|---|
| 1: Loop { | 1: Loop { |
| 2: wait(!wantQ); | 2: wait(!wantP); |
| 3: *wantP* := true; | 3: *wantQ* := true; |
| 4: Critical section; | 4: Critical section; |
| 5: *wantP* := false; | 5: *wantQ* := false; |
| } | } |

States $= \{\langle 1, 1, \mathit{false}, \mathit{false} \rangle, \langle 2, 1, \mathit{false}, \mathit{false} \rangle, \ldots\}$

# Transition Relation

•



# Property

- Specifications can be given in different formats:

## Property

- Specifications can be given in different formats:
  - assertions,

# Property

- Specifications can be given in different formats:
  - assertions,
  - graphs (automata) that specify good behaviors,

# Property

- Specifications can be given in different formats:
  - assertions,
  - graphs (automata) that specify good behaviors,
  - logic formulas with navigation operators (temporal operators)

# Property

- Specifications can be given in different formats:
  - assertions,
  - graphs (automata) that specify good behaviors,
  - logic formulas with navigation operators (temporal operators)
  - ...
- In our example an assertion defined on a new variable $critical == 1$

# Models with Infinite State-Space?

- Extended Finite-State Machines
  - Data: Unbounded local and global variables
  - Stack: Recursive Boolean programs
  - Channels: (Unreliable) Communication systems
  - ...

# Models with Infinite State-Space?

- Extended Finite-State Machines
  - Data: Unbounded local and global variables
  - Stack: Recursive Boolean programs
  - Channels: (Unreliable) Communication systems
  - ...
- Parameterized Systems
  - Parameters in the transitions
  - Families of systems

# Models with Infinite State-Space?

- Extended Finite-State Machines
  - Data: Unbounded local and global variables
  - Stack: Recursive Boolean programs
  - Channels: (Unreliable) Communication systems
  - ...
- Parameterized Systems
  - Parameters in the transitions
  - Families of systems
- Computability issues: What can be verified?

# Parameterized Verification

- Model: a concurrent system with an arbitrary (finite) number of components

# Parameterized Verification

- Model: a concurrent system with an arbitrary (finite) number of components
- Classes of topologies: Unordered, Linearly ordered, Tree-shaped, Graph-based

# Parameterized Verification

- Model: a concurrent system with an arbitrary (finite) number of components
- Classes of topologies: Unordered, Linearly ordered, Tree-shaped, Graph-based
- Goal: Verify a Property for any number of processes (any topology in a given class)

# Parameterized Verification Methods

- Deductive Methods: Invariants and Theorem Proving

# Parameterized Verification Methods

- Deductive Methods: Invariants and Theorem Proving
- Abstractions: Reductions to Finite-state Systems

# Parameterized Verification Methods

- Deductive Methods: Invariants and Theorem Proving
- Abstractions: Reductions to Finite-state Systems
- Regular Model Checking: Automata-based Representation of Sets of Configurations

# Parameterized Verification Methods

- Deductive Methods: Invariants and Theorem Proving
- Abstractions: Reductions to Finite-state Systems
- Regular Model Checking: Automata-based Representation of Sets of Configurations
- **Constraint-based Model Checking**: Constraints as Representation of Sets of Configurations

# Example: Lamport's Bakery Algorithm for N-processes

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                    i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

Verify mutual exclusion for any number of processes!

# Plan of the Lessons

- Verification of Finite-state Systems and Abstractions
- Verification of Infinite-state Systems and Abstractions
- Parameterized Verification and Abstractions