COMP6210 Automated Software Verification

Lecture 1: Introduction

Pavel Naumov

Outline

Outline

Module Overview

Administrative Info

Introduction to Software Verification

Introduction to Model Checking

Module Aims

To give an understanding of both the theory and the practice of Automated Software Verification

- key principles and techniques
- use of a range of verification tools
- both formal verification and testing covered

Module Learning Outcomes

- Understand a range of testing and formal verification approaches applicable to software systems
- Use logic as a specification language for software correctness
- Apply automated verification techniques to software
- Assess the limitations of current verification techniques and tools
- Select appropriate verification tools to analyse and verify small-scale systems

What is Automated Verification

- Verification amounts to checking that a (software) system behaves as intended. In this case the system is said to be correct.
- Automated verification automates this.
    - intrinsic limits, e.g. cannot prove program termination
- wide range of automated verification techniques:
    - ...
    - model checking
    - deductive verification
    - symbolic execution
    - automated testing
    - ...
- some of these provide guarantees on correctness (i.e. that the system always behaves as intended)

Module Outline

1. Model checking
    - Explicit state model checking (≈ 2 weeks)

        modelling software systems;
        the temporal logic LTL;
        model checking LTL
    - Symbolic model checking (≈ 1 week)

        symbolic modelling of software
        Binary Decision Diagrams
        the NuXMV model checker `https://nuxmv.fbk.eu`
    - Bounded model checking (≈ 1.5 weeks)

        basic concepts of BMC
        SAT solvers
        the CBMC model checker `https://www.cprover.org/cbmc/`

Module Outline (Cont'd)

2. Deductive program verification ($\approx$ 3.5 weeks)
    - Hoare logic
    - loop termination
    - Frama-C `https://frama-c.com`
        - use WP plugin for verification
3. Testing ($\approx$ 2 weeks)
    - types of testing
    - automated software testing
        - regression testing, fuzz testing
        - symbolic and concolic testing

$\Longrightarrow$ experience with a wide range of technologies !

Outline

Module Overview

Administrative Info

Introduction to Software Verification

Introduction to Model Checking

Administrative Info (2)

Prerequisites:

- Java, C (basic knowledge)
- logic (basic knowledge, e.g. propositional logic)

Assessment:

- coursework on model checking using NuSMV and CBMC (20%; due week 7/8)
- coursework on deductive verification using Frama-C (15%; due week 15)
- final assessment: (65%)

Coursework feedback:

- within three weeks of submission

Course Materials

- slides (self-contained)
  - will be posted after the lectures
- lecture recordings
  - available shortly after lectures
- weekly exercises
  - set during lectures
  - to be attempted before tutorial session
- background resources
  - see next slide
  - more to be announced later

Background Reading (Explicit/Symbolic Model Checking)

- E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 1999. (Chapters 1-5)
- M. Huth and M. Ryan, *Logic in Computer Science – Modelling and Reasoning about Systems* (second edition), Cambridge University Press, 2004. Available electronically through WebCat. (Chapters 3 and 6)
- E. M. Clarke, T. A. Henzinger, H. Veith and R. Bloem (editors), *Handbook of Model Checking*, Springer, 2018. Available online from Springer. (Chapters 1-5, 7, 8, 10)
- C. Baier and J.-P. Katoen, *Principles of Model Checking*, MIT Press, 2008. (Chapters 1-5)

Note: latter two provide in-depth coverage that goes beyond what is expected for the module!

Outline

Verification versus Validation

- validation: check that the program we are building fulfills its intended purpose (meets the user needs)

  Are we building the right product?

- verification: check that the program meets its requirements and design specifications

  Are we building the product right?

Software Verification Approaches

- peer review
  - manual code inspection, no software execution
  - subtle errors (algorithm design, concurrency) difficult to detect
- testing
  - software is executed to catch errors
  - amounts to 30-50% of development costs
  - effective in the early stages but time-consuming in later stages
  - difficult for concurrent or distributed systems
  - only *some* executions are explored
- simulation
  - performed on an *abstraction*, or model, of the actual program
  - only *some* behaviours are explored

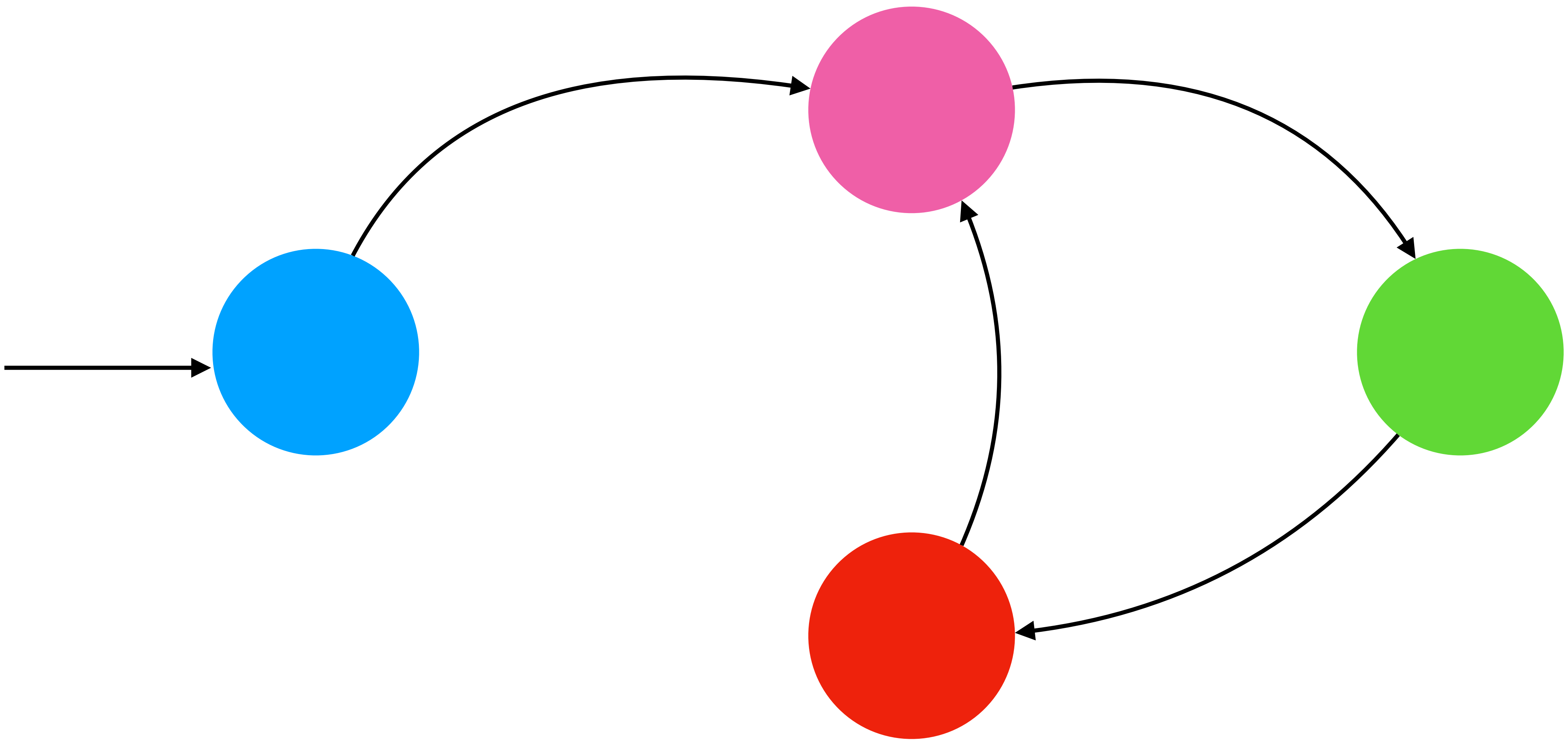
Software Verification Approaches (Cont'd)

- formal verification approaches offer correctness guarantees:
  - deductive verification:
    - checking correctness properties of either programs or their models
    - theorem provers used to prove correctness (e.g. *invariant* properties)
    - not fully automatic...

      ...but several good tools exist, e.g. KeY, Frama-C
    - works on infinite state systems
  - model checking:
    - checking properties of either models or code
    - performs exhaustive exploration of *all* possible behaviours
    - works only on systems with finite state spaces
    - fully automatic

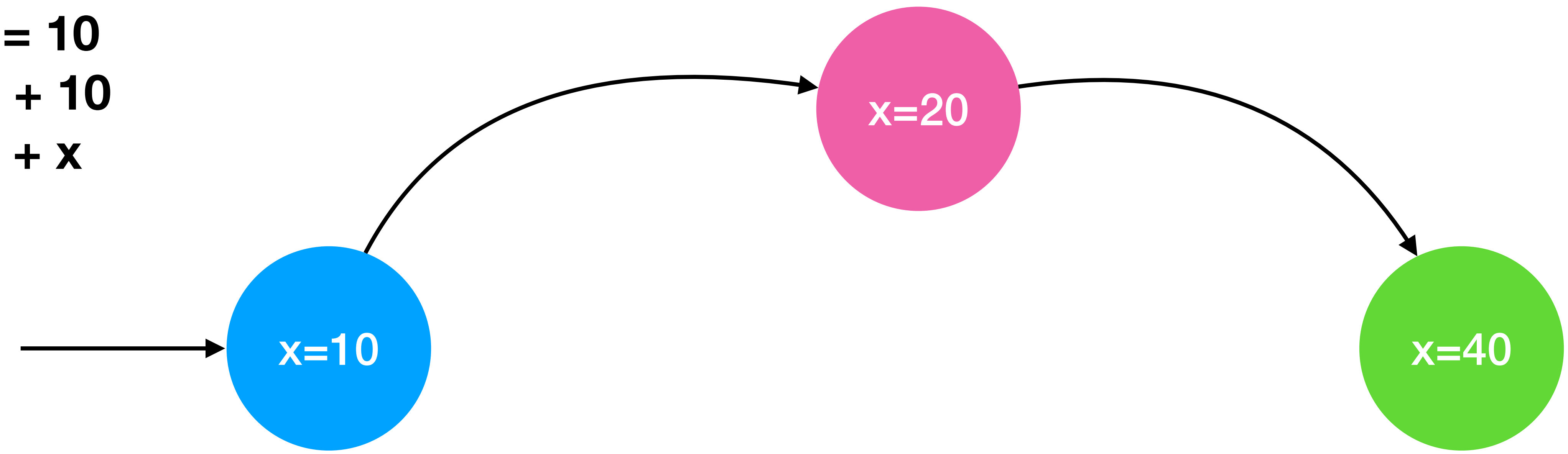
Outline

Transition System



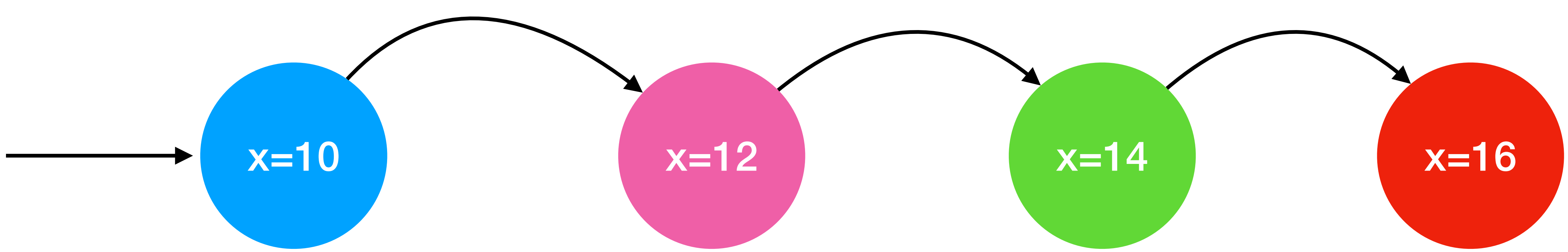
From Code to Transition System

```
int x = 10
x = x + 10
x = x + x
```



From Code to Transition System

```
int x = 10
while (x<13)
   x = x + 2
x = x + 2
```



A Small Example

```
process Inc = while (true) { if (x < 200) x = x + 1 }
process Dec = while (true) { if (x > 0) x = x - 1 }
process Reset = while (true) { if (x == 200) x = 0 }
```

- three processes (threads) running concurrently
- shared variable x (integer)
- assume the initial value of x is 0

A Small Example

```
process Inc = while (true) { if (x < 200) x = x + 1 }
process Dec = while (true) { if (x > 0) x = x - 1 }
process Reset = while (true) { if (x == 200) x = 0 }
```

- three processes (threads) running concurrently
- shared variable x (integer)
- assume the initial value of x is 0

Some questions:

1. How many reachable program states does the program have?
2. How many ways to move between states does the program have?
3. How many possible ways to execute this program are there?
4. Is the value of x always between 0 and 200?

A Small Example

```
process Inc = while (true) { if (x < 200) x = x + 1 }
process Dec = while (true) { if (x > 0) x = x - 1 }
process Reset = while (true) { if (x == 200) x = 0 }
```

- three processes (threads) running concurrently
- shared variable x (integer)
- assume the initial value of x is 0

Some questions:

1. How many reachable program states does the program have?
2. How many ways to move between states does the program have?
3. How many possible ways to execute this program are there?
4. Is the value of x always between 0 and 200?

A Small Example

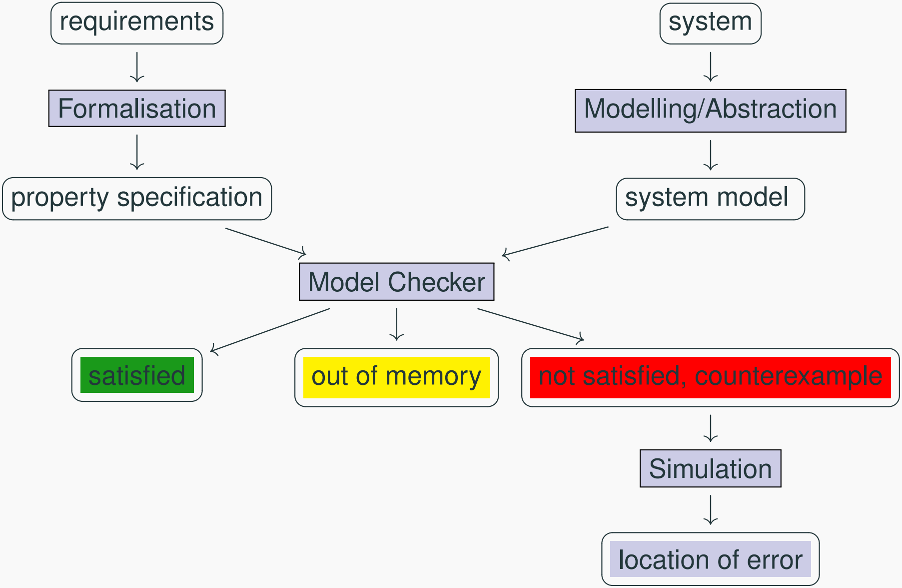
```
process Inc = while (true) { if (x < 200) x = x + 1 }
process Dec = while (true) { if (x > 0) x = x - 1 }
process Reset = while (true) { if (x == 200) x = 0 }
```

- three processes (threads) running concurrently
- shared variable x (integer)
- assume the initial value of x is 0

Some questions:

1. How many reachable program states does the program have?
2. How many ways to move between states does the program have?
3. How many possible ways to execute this program are there?
4. Is the value of x always between 0 and 200?

A Typical Model Checker



Model Checking – the Process

1. modelling: build *abstraction* of the program as a model (e.g. transition system)
    - can use simulation to *validate* model
2. specification: specify desired properties of the model/system in a suitable formalism (e.g. model annotations, temporal logic)
3. verification with a model checker:
    - explore *all* possible behaviours / executions in order to verify that specified properties hold
        - e.g. all possible thread interleavings for concurrent programs !
    - counter-example (*error trace*) produced when the model / program does not satisfy the specification

      $\Downarrow$

      simulate counter-example

      $\Downarrow$

      revise model/code and/or specification

Model Checking in Practice

- more recent model checkers work on software, not models
- correctness properties can be generic (e.g. arithmetic overflow, array bounds, division by zero, pointer safety, deadlock) or program specific (e.g. *"value of x is always between 0 and 200"*, or *"value of x eventually reaches 100"*)
- symbolic model checking: the states that the program / model can reach are represented symbolically
- bounded model checking: *bounded* exploration (only up to a certain depth) is performed to improve performance/ensure finite state space
    - impact on correctness guarantees

Next Time

- Modelling programs with transition systems