Verification of Software Safety Properties — An Introduction

Terminology

- A *property* of a program is some characteristic (possibly a desired one) of its executions, for instance

  a. no overflow occurs
  b. no segmentation fault occurs
  c. the output is the factorial of the input
  d. the output is *true* if x occurs in the array v and false otherwise
  e. the output is a sorted permutation of the input
  f. the execution terminates

- Properties are classified as

  - **Safety** properties: “something” does not happen

  - **Liveness** properties: “something” eventually happens

  Technically, property f above is a liveness property, and properties a-e are safety properties (for instance property c. excludes final states in which the output does not correspond to the factorial of the input)

- Among safety properties we will distinguish:

  - Safety of executions — program “does not go wrong” (such as a. and b. above)

  - Functional properties, describing what the program does or calculates (such as c. and d.)

- A program is *correct* w.r.t. a given property if **every execution satisfies the property**

Approaches to Code Checking / Validation

- **Testing** — not exhaustive, does not produce guarantees. BUT, it is still valuable and used in practice

- **Dynamic or runtime verification** — similarly to testing, implies running the program, so executions are considered individually. The difference is that the code can be *instrumented,* i.e. modified so that specific conditions can be checked at different points of the program, or a *trace* of the execution can be stored to be examined later, offline.
  In the context of concurrent or distributed systems a *monitor* may be launched to perform the verification tasks concurrently with the program's threads / nodes

- **Static methods** — these do not require running the program, and consider the full set of executions at once. These can be broadly grouped into:

  - Static Analysis methods: techniques employing specific algorithms to analyze specific properties. This include techniques based on control-flow analysis, dataflow analysis, and abstract interpretation.
    See here an interesting article on static analysis methods

  - Software Model Checking: automatic and systematic exploration of the state space of a program

  - Deductive Program verification methods, that try to establish correctness using program logic and first-order deduction

Program Verification is the activity (and research area) that studies and establishes the correctness of programs, using a variety of methods. Can be applied at the algorithmic level or at the level of concrete programming languages and infrastructures

Quoting Great Computer Scientists:

> *Software is Hard*

— Donald E. Knuth

> *Any **non-trivial** property about the language recognized by a Turing machine is **undecidable**."*

— Rice's Theorem

It is indeed impossible in general to verify software properties automatically.

This applies already to programs of very simple languages, even if we don't consider major ingredients of modern software, such as

- Large types and user-defined types
- Pointers, aliasing, and unbounded numbers of cells allocated in dynamic memory
- Concurrency with unbounded numbers of threads
- Interrupts / exceptions
- Secondary storage: files, databases
- The sheer size of code bases!

> *In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is **abstraction***

— C. A. R. Hoare

Automated Static Verification — Software Model Checking

In principle it is easy to set up a system for checking safety properties automatically by considering a transition system where

- Configurations are *program states* (i.e. sets of variable values)
- Transitions between configurations are given by the semantics of the programming language
- Given a safety property, the states where the property does not hold are classified as forbidden, or Error states
- Safety verification can then be reduced to a **reachability** problem: showing that the forbidden states are not reachable from the initial state — in other words, no execution (sequence of transitions) leads from the initial state to an error state.

The problem is that in practice it is not possible to explore this state space algorithmically, by considering all executions, because of the **state explosion problem.**

The size of the state space explodes in general in model checking as the system grows, but in *software* model checking the problem is worse because of certain program elements like large date types and multiple threads.

There exist three possible solutions to this problem, which all involve giving up on something:

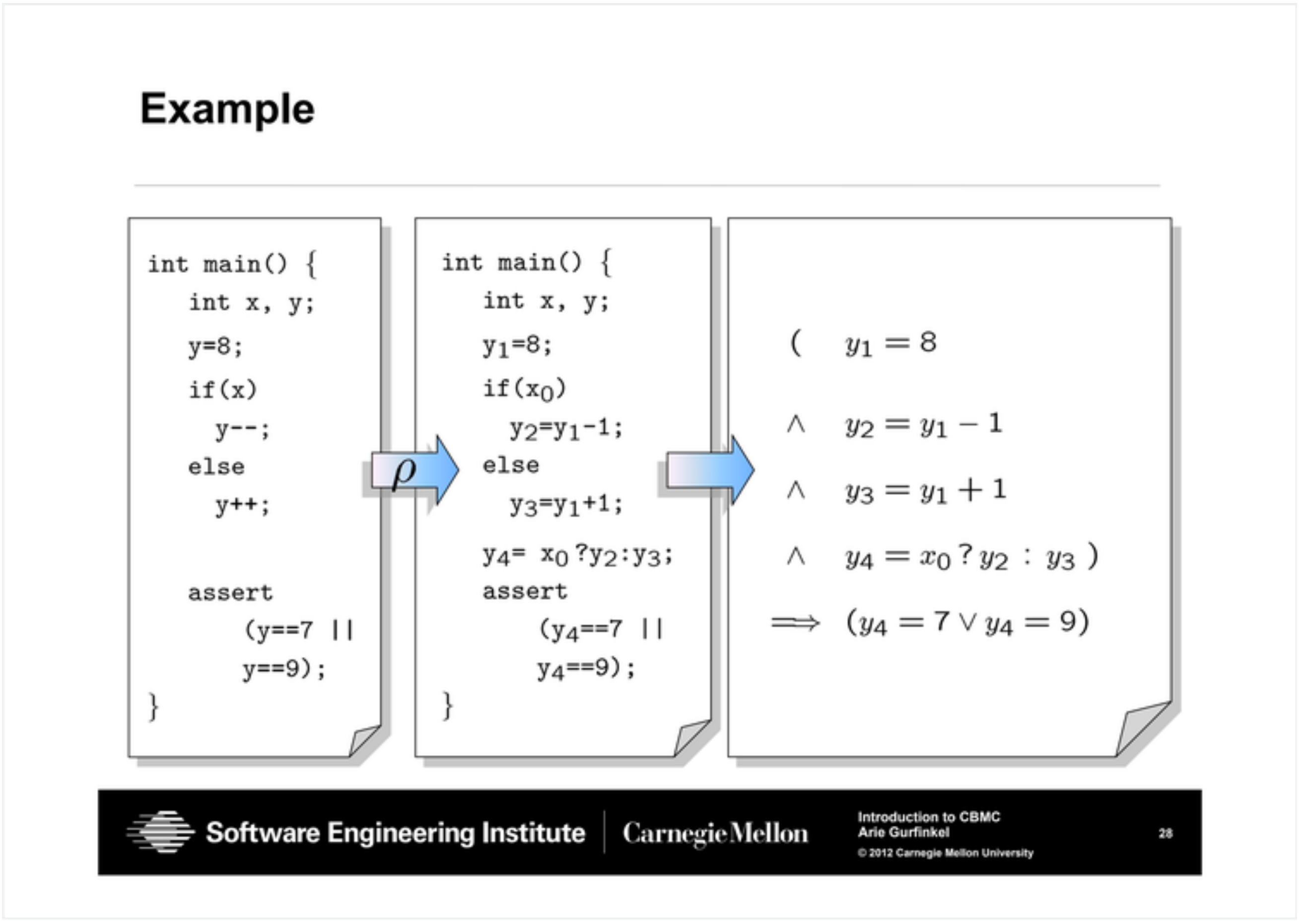
- Soundness
- Completeness, or
- Automation

A software verifier is **sound** if it reports every property violation (every error state that may be reached). In other words, if it says the program is correct, then it really is correct.

A verifier is **complete** if all the violations it reports are indeed errors. In other words, if it says the program is incorrect then there exists at least one property violation in its state space.

Bounded Model Checking

It is quite straightforward to formulate safety of an iteration-free imperative program as a satisfiability or validity problem, by encoding the program in logic using a Static Single Assignment intermediate form:



Example

```
int main() {
  int x, y;
  y=8;
  if(x)
    y--;
  else
    y++;

  assert
     (y==7 ||
     y==9);
}
```

ρ

```
int main() {
  int x, y;
  y1=8;
  if(x0)
    y2=y1-1;
  else
    y3=y1+1;

  y4= x0 ?y2:y3;
  assert
     (y4==7 ||
     y4==9);
}
```

$$
\begin{aligned}
(\quad & y_1 = 8 \\
\wedge \quad & y_2 = y_1 - 1 \\
\wedge \quad & y_3 = y_1 + 1 \\
\wedge \quad & y_4 = x_0 \, ? \, y_2 : y_3 \,) \\
\Longrightarrow \quad & (y_4 = 7 \vee y_4 = 9)
\end{aligned}
$$

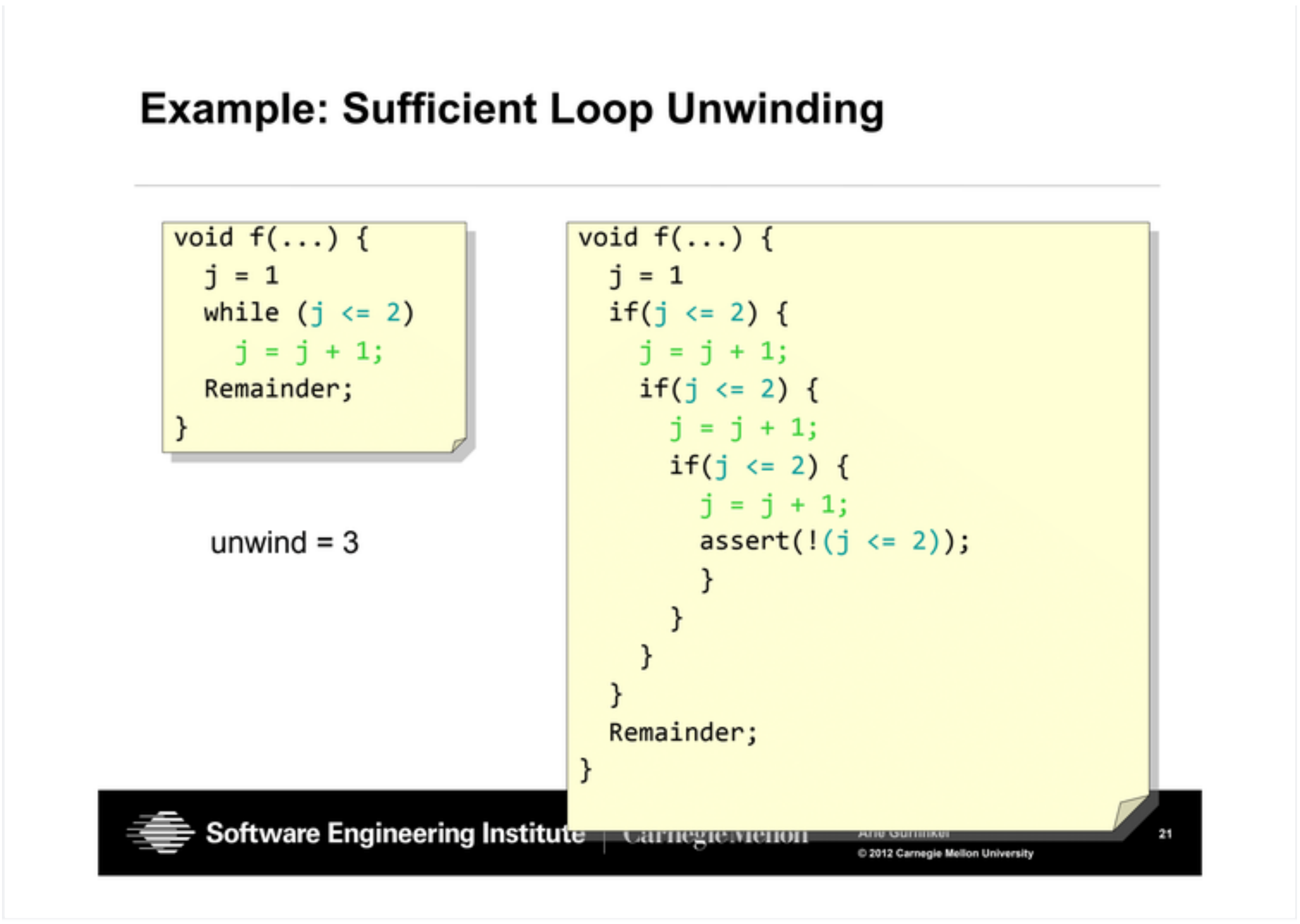
Software Engineering Institute | Carnegie Mellon

Introduction to CBMC
Arie Gurfinkel
© 2012 Carnegie Mellon University

28

PDF 6-cbmc(lec26) (dragged) (4) • PDF Document

In the final state of the above program, the value of y must be 7 or 9, otherwise an error state has been reached. If the proof obligation on the right is **valid** the program is correct.



Example: Sufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 2)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 2) {
    j = j + 1;
    if(j <= 2) {
      j = j + 1;
      if(j <= 2) {
        j = j + 1;
        assert(!(j <= 2));
      }
    }
  }
  Remainder;
}
```

Software Engineering Institute | Carnegie Mellon

© 2012 Carnegie Mellon University

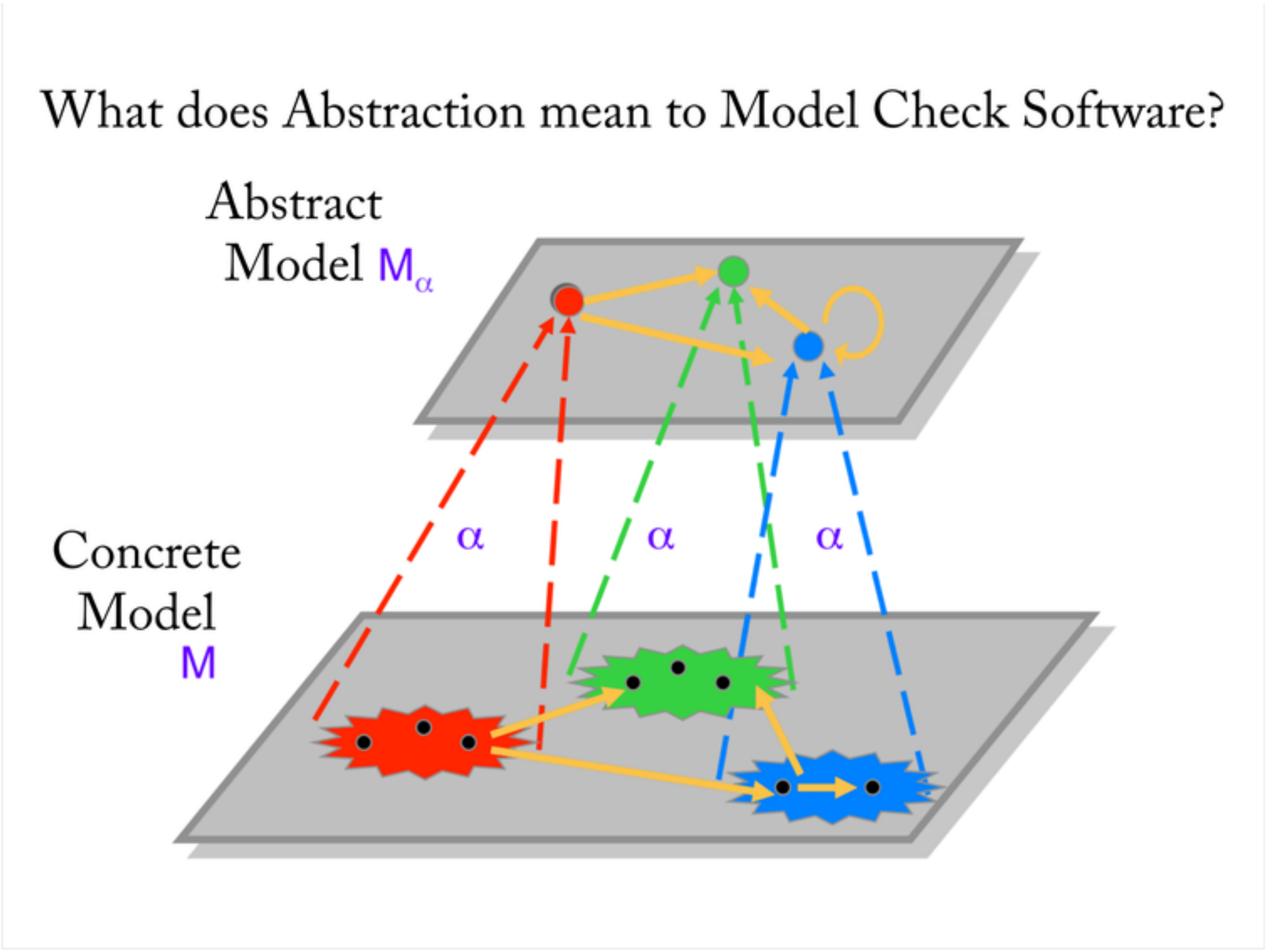
21

PDF 6-cbmc(lec26) (dragged) (2) • PDF Document

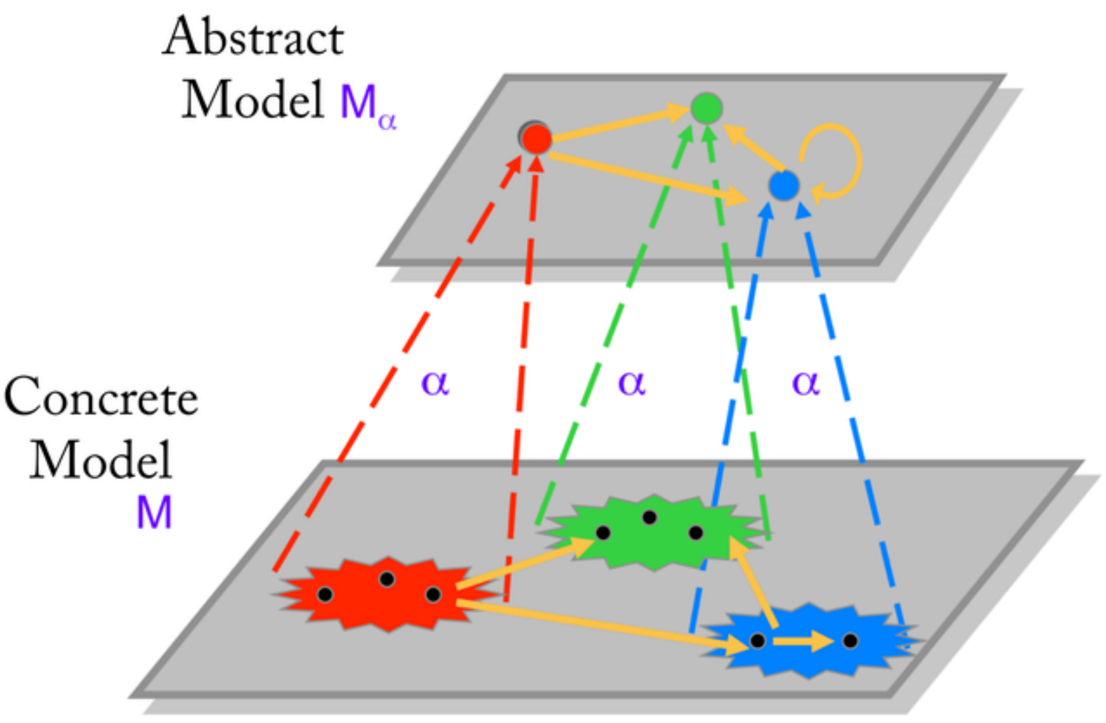
But of course it may not be known how many iterations are sufficient... not every loop corresponds to a defined iteration like the above. And no matter how many iterations are unfolded, this may not cover all possible executions of the program.

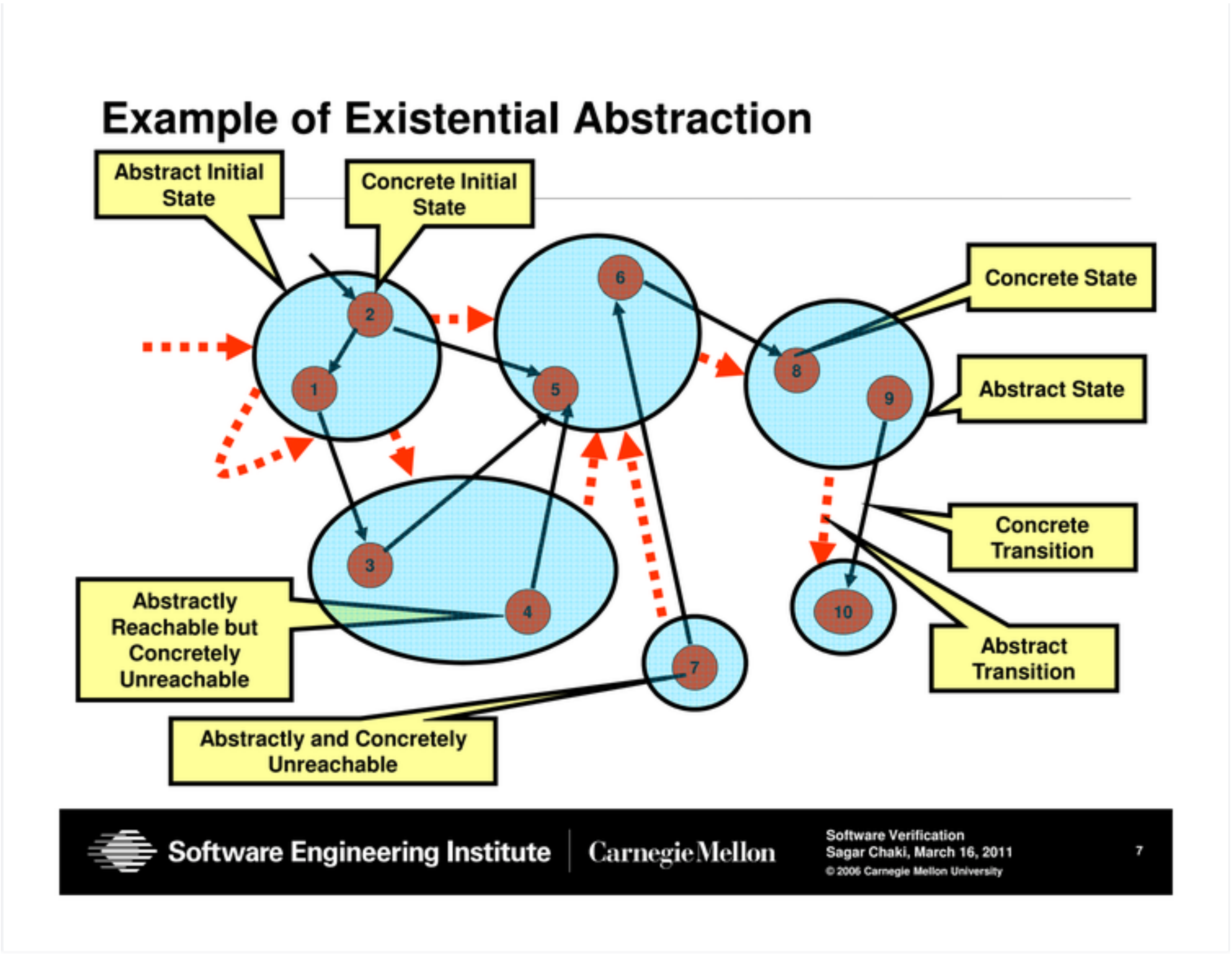
For this reason, bounded model checking is essentially *unsound,* although it can always be reported that the number of iterations was not sufficient (in this sense it is sometimes claimed to be sound).

Abstract Model Checking



What does Abstraction mean to Model Check Software?

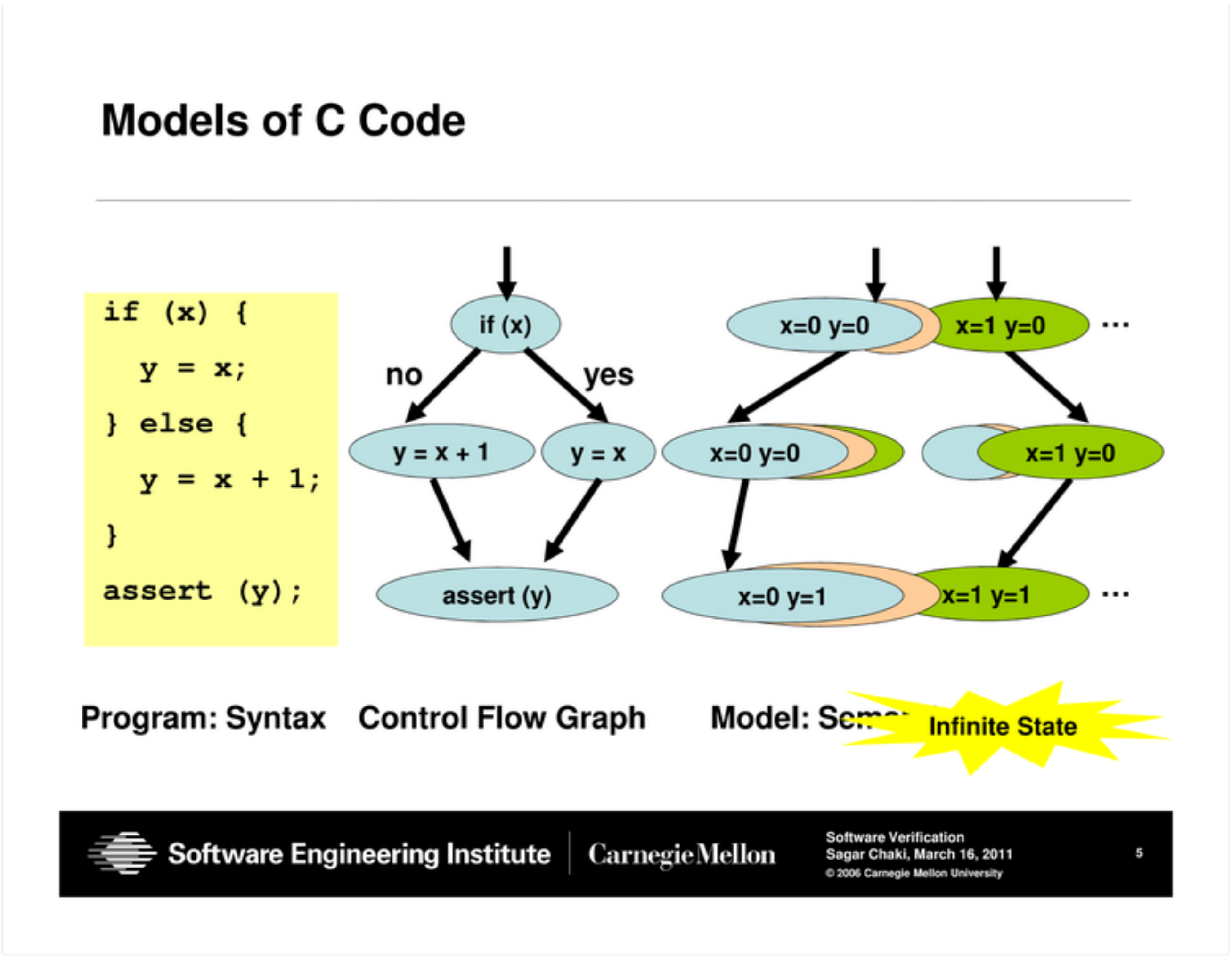



Example of Existential Abstraction
Abstract Initial State
Concrete Initial State
Concrete State
Abstract State
Concrete Transition
Abstract Transition
Abstractly Reachable but Concretely Unreachable
Abstractly and Concretely Unreachable
1
2
3
4
5
6
7
8
9
10
Software Engineering Institute
Carnegie Mellon
Software Verification
Sagar Chaki, March 16, 2011
© 2006 Carnegie Mellon University
7

PDF lec24_PA+IR (dragged) • PDF Document

Note that *abstraction is sound*: if a violation occurs in the concrete model, it also occurs in the abstract model.

However, the approach is *not complete*: the abstract model may contain spurious counterexamples ("false positives") that need to be checked



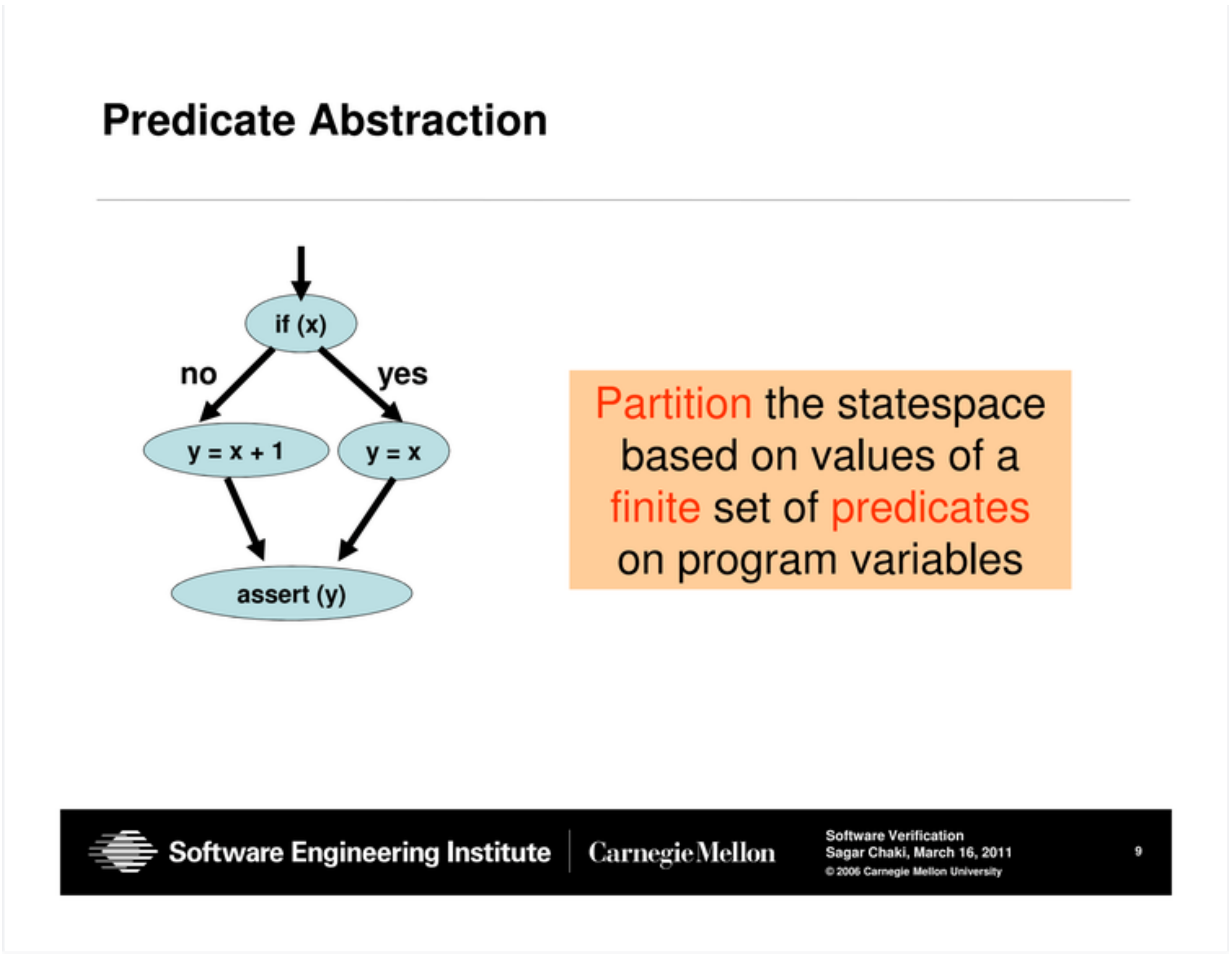
Models of C Code

```
if (x) {
  y = x;
} else {
  y = x + 1;
}
assert (y);
```

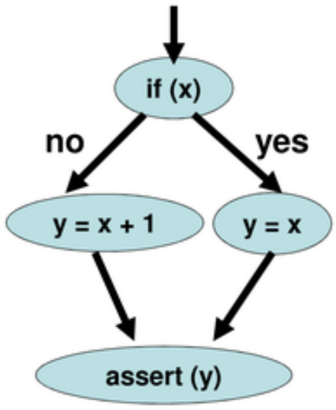
Program: Syntax Control Flow Graph Model: Sem…

PDF lec24_PA+IR (dragged) (4) • PDF Document

No execution leads to an error (y=0 in the final state), but the number of executions is not feasible to check, due to the nature of the data types. Note that no complicated control flow is required to make checking impossible!

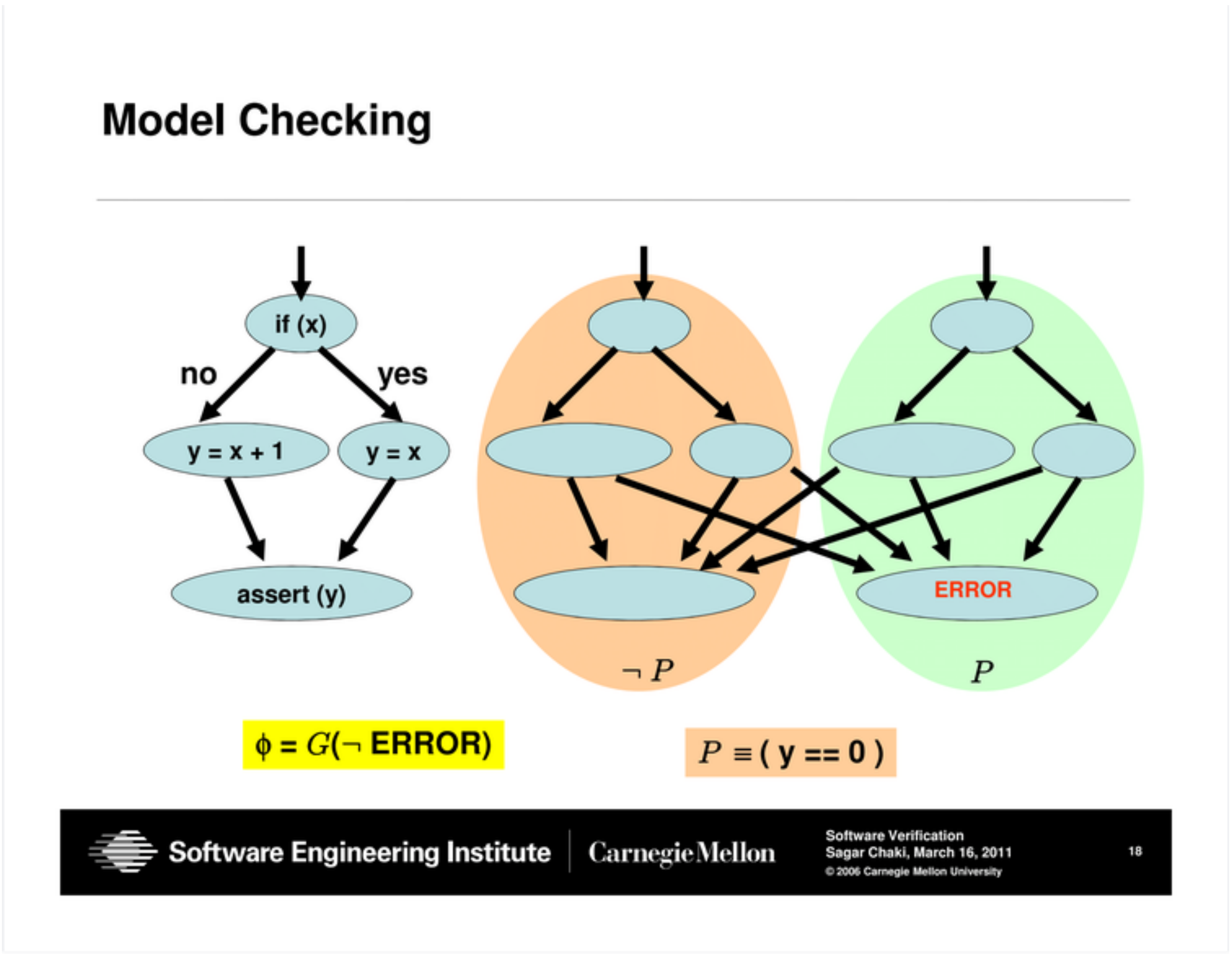


Predicate Abstraction



Partition the statespace based on values of a finite set of predicates on program variables

PDF lec24_PA+IR (dragged) (1) • PDF Document



PDF lec24_PA+IR (dragged) (2) • PDF Document

An obvious choice of predicate is the one that is used to define errors. In the leftmost part of the model the error never occurs.

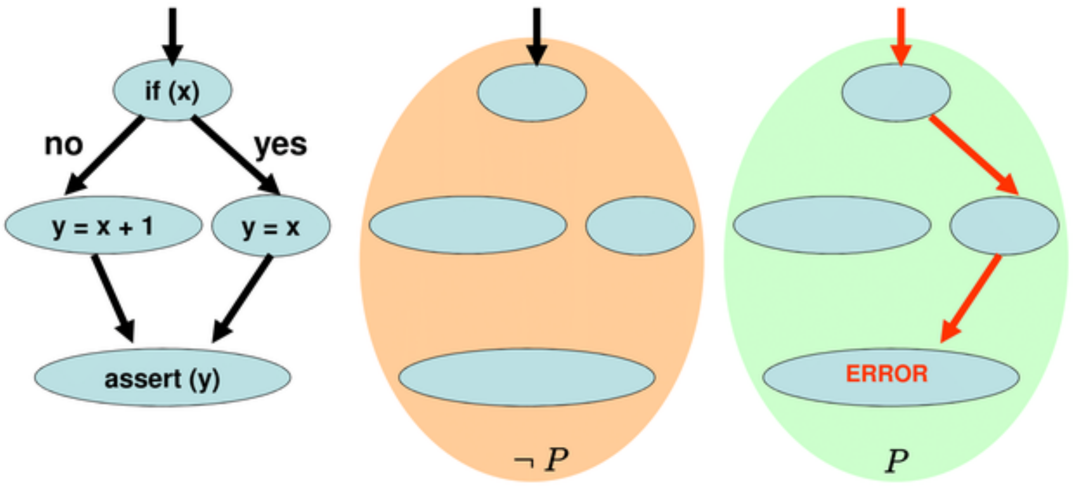
Instead of considering states for all the concrete values of the variables, only a pair of abstract values is considered: values satisfying / not satisfying the predicate

Note that the model is constructed by checking the feasibility of individual transitions, not paths! The rightmost execution is not possible, although the individual transitions are:

- it is possible to have x<>0 (conditional) and y=0 (P), thus the top transition is possible
- it is possible to have y=0 and y=x, thus the bottom transition is possible
- the path is not feasible, but the model does not capture that

So the abstraction did not solve the problem: the abstract model contain a spurious counter-example

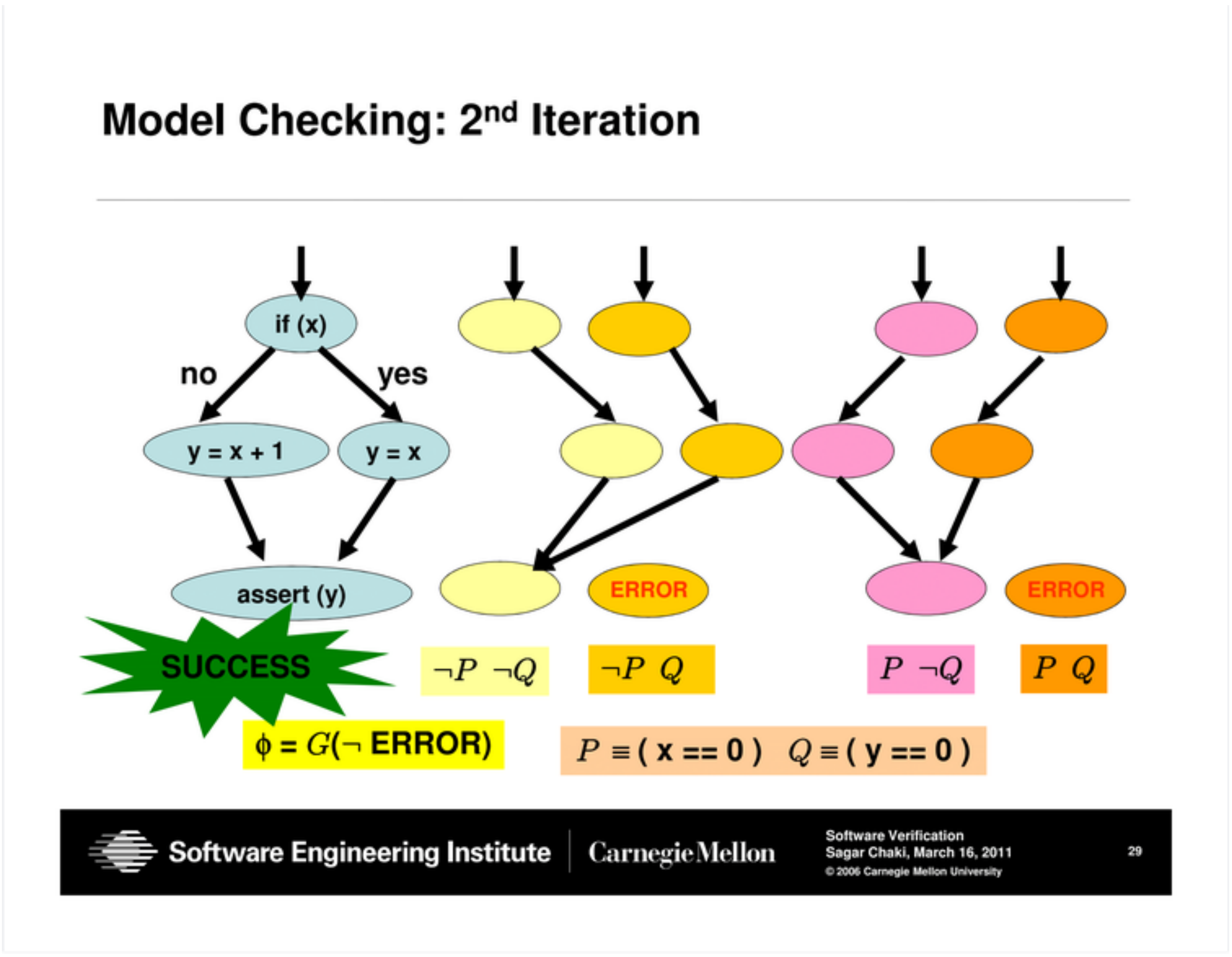
Model Checking



$\phi = G(\neg \text{ERROR})$

$P \equiv (\, y == 0 \,)$

PDF lec24_PA+IR (dragged) (3) • PDF Document


Model Checking: 2nd Iteration
if (x)
no
yes
y = x + 1
y = x
assert (y)
SUCCESS
ERROR
ERROR
¬P ¬Q
¬P Q
P ¬Q
P Q
ϕ = G(¬ ERROR)
P ≡ (x == 0)   Q ≡ (y == 0)
Software Engineering Institute
Carnegie Mellon
Software Verification
Sagar Chaki, March 16, 2011
© 2006 Carnegie Mellon University
29

PDF lec24_PA+IR (dragged) (5) • PDF Document

The model can be refined by including an additional predicate. In practice, tools do this by examining the spurious counterexamples.

In the above example, considering the additional predicate testing the value of x for 0 leads to a successful verification: no error occurs in any combination of the predicate values.

Deductive Verification

These methods involve reasoning about programs using a program logic, together with First-Order Logic (FOL) and specific theories for data types (numeric, arrays, and others).

The verifier typically generates from the program a set of FOL formulas called *verification conditions* (VCs). The program is correct if and only if every VC is valid.

These formulas are sent to a proof tool to be discharged. The tool can be internal or external to the verifier, and it can be automated or interactive.

So what is the magic? What about Rice's theorem?

Well, deductive verification is *not an automated method*, because it requires input from users. In particular, it may require annotating the program with loop invariants and other elements.

There is another sense in which deductive verification is not automated:
it is not guaranteed that the validity of verification conditions can be decided automatically, since FOL is not decidable. This means that if an automated theorem prover is used to discharge VCs it may time out or return "unknown", in which case it is necessary to use a (non-automated!) interactive proof assistant.