Directions in Formal Verification of Software

Ishai Rabinovitz
Verification Technologies
IBM Haifa Labs

Outline

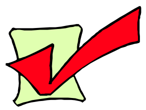
- What are formal verification and model checking?
- Why is formal verification for software so hard?
- Some basic techniques for software model checking
- The work here at IBM

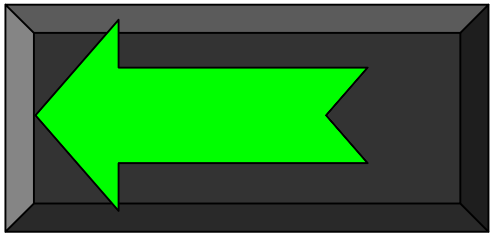
Verification

- Two main approaches to automatically find bugs in software and hardware
  - Testing (simulation)
  - Formal verification

Testing

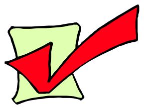
- Run on some inputs and examine the results
- Can measure some kind of coverage
- Advantages:
  - Relatively easy
  - Checks many aspects of the tested run (control as well as data)
- Disadvantages
  - Cannot prove correctness (falsification only)

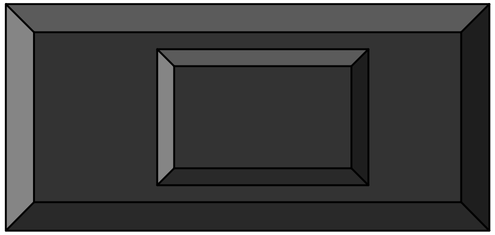




Formal verification

- Checks all possible runs
- Advantages:
  - Verification of the specification is possible (not only falsification)
- Disadvantages
  - Hard
  - Not always feasible
  - Good for control checking (not data)



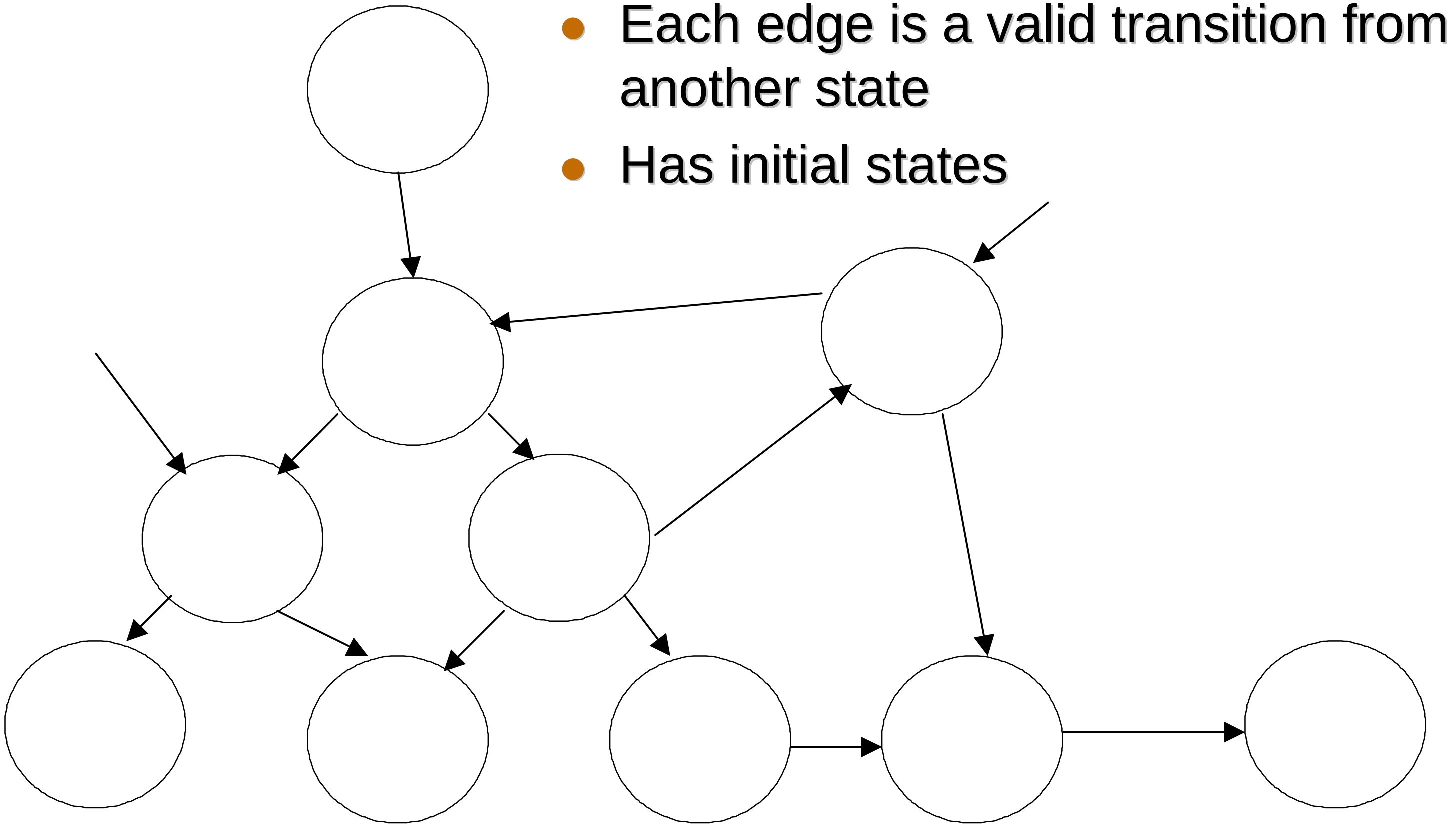


Formal verification techniques

- Theorem proving
- Model Checking
  - Explicit model checking
  - Symbolic model checking

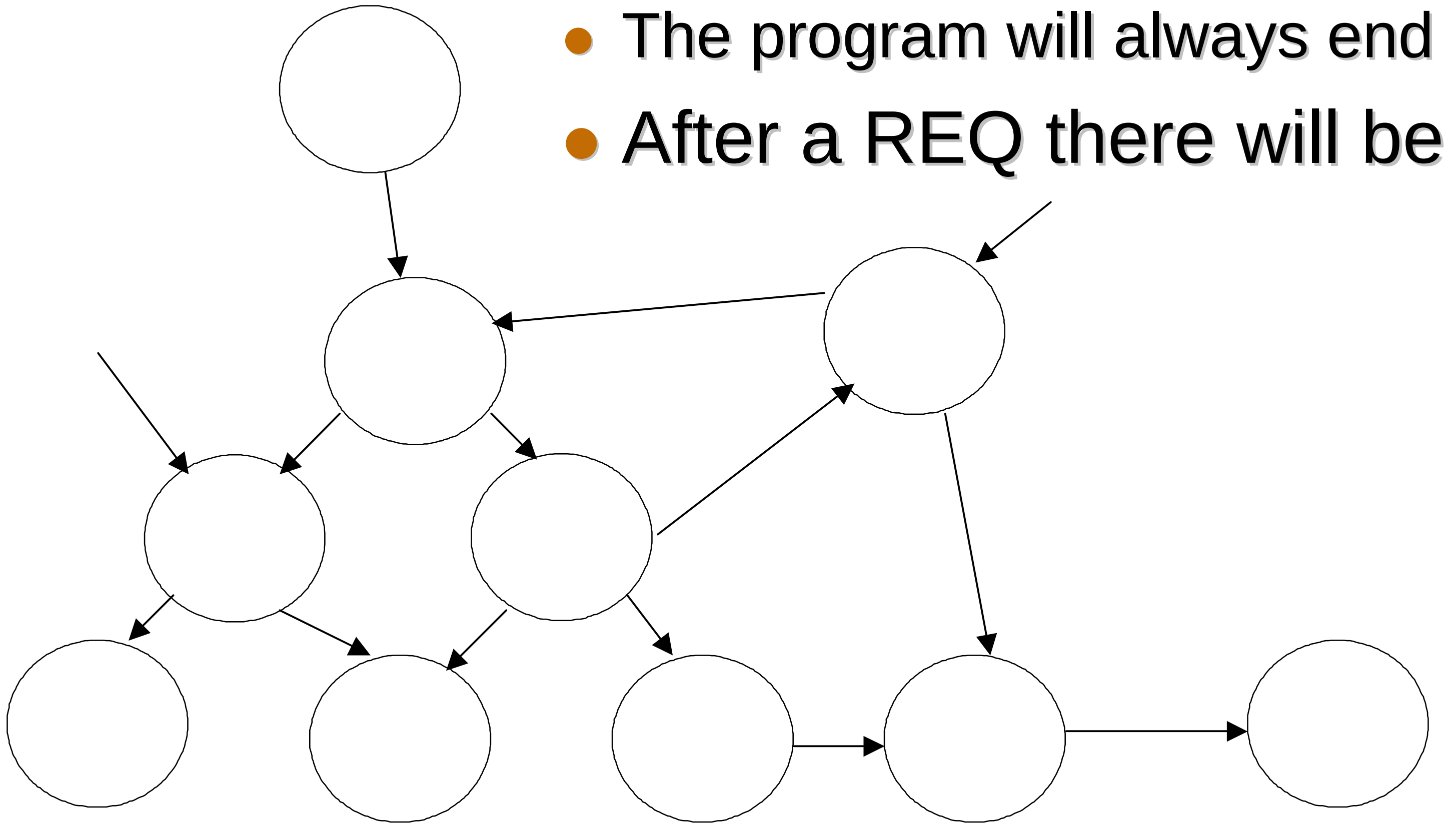
Model checking

- Build a model
- A model can be represented as a graph
  - Each vertex is a state of the system – value to all the variables (registers)
  - Each edge is a valid transition from state to another state
  - Has initial states



Model checking (2)

- We can check specifications like:
  - Always i<=j
  - The program will always end
  - After a REQ there will be an ACK



Formal verification of Hardware

- Success story
- Widely used in the industry
- Highly qualified users are needed
- Several successful techniques:
  - manual: divide and conquer, restrictions
  - automatic: abstraction refinement, and more

Verifying software is harder ！

Why?

Theoretical problems

- Software is undecidable. (Does the program end?)
- Software is unbounded (stack, dynamically allocated memory)

- Even if we restrict ourselves to finite implementations (the computer's memory is bounded) it is hard

Even finite software is hard

- Programming languages have complicated semantics (hard to model):
  - Functions
  - Recursion
  - Pointers
- Hardware techniques do not transfer:
  - Data manipulation (vs. control)
    - The control path is integrated with the data path
    - It is hard to express data relationship in symbolic model checker (huge BDDs)
  - Less modularity
    - It is difficult to use divide and conquer techniques

Harder type of parallelism

- While hardware designs use massive parallelism, a common clock is usually implemented (synchronous systems)
- In software no common clock is available (asynchronous systems)
- Asynchronous systems present more behaviors. – generates much bigger models

(an example will be given later)

Economic problems

- Bugs in HW are:
  - Not acceptable by the users
  - Very expensive to repair
- Bugs in SW are:
  - Tolerated by the user
  - Relatively cheap to fix
- HW vendors are willing to invest large resources (time, money and expert personnel) in verifying HW
- SW companies are not

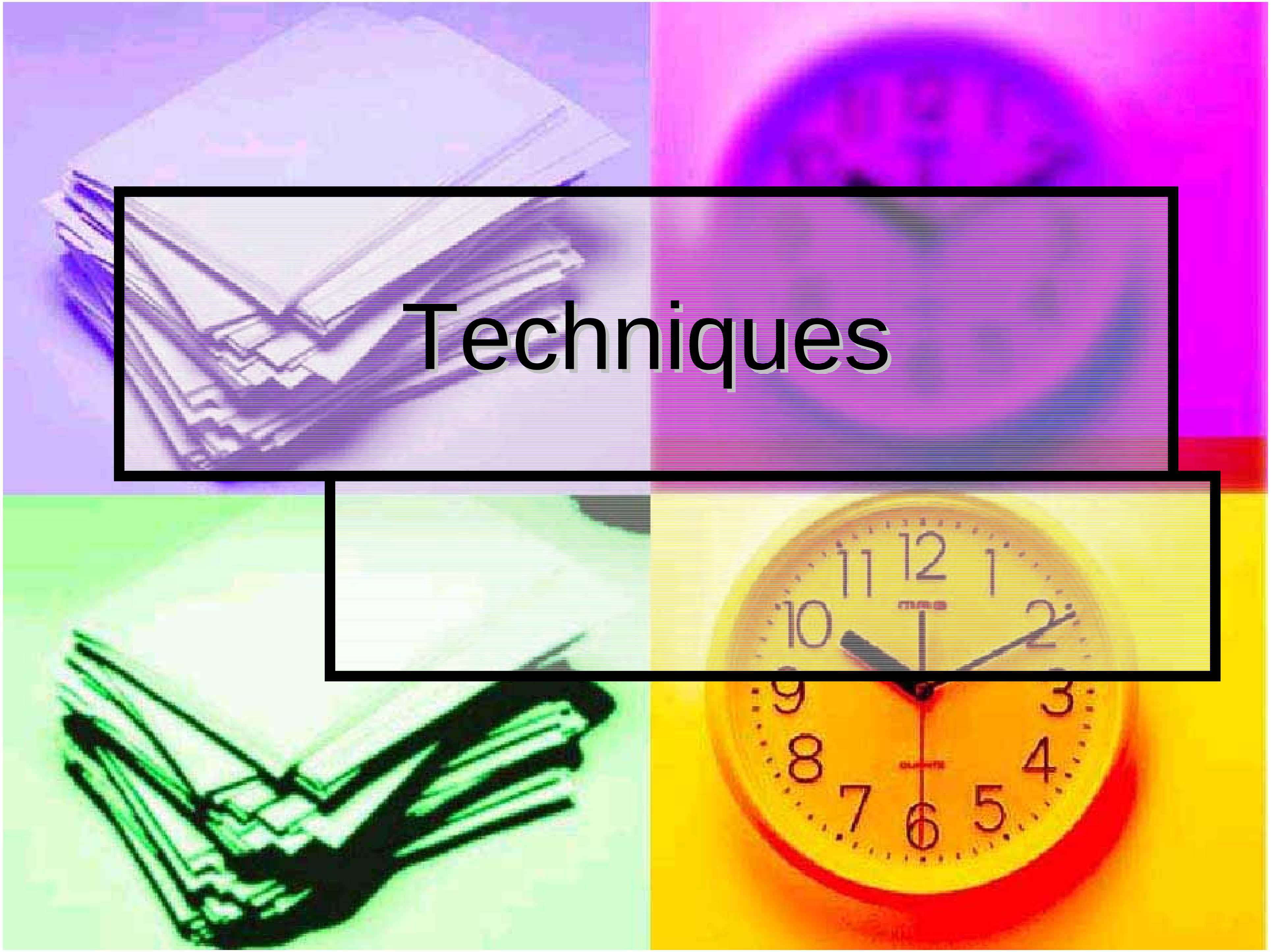
So why use formal verification on software?

- Parallel programs
  - Hard to test
  - Poor coverage
  - Programmers have less intuition
  - SW companies are willing to invest in skilled personnel
- Micro-code, smart-cards etc
  - Closer to hardware (in size and features)
  - Bugs are expensive to fix
- Critical software (intensive care systems, finance, security, anti-missiles systems etc.)

Simpler user interface is needed

- Write a specification in a simple way. (not all programmer familiar with temporal logic)
- Presenting the bug (counter example) in the program terms

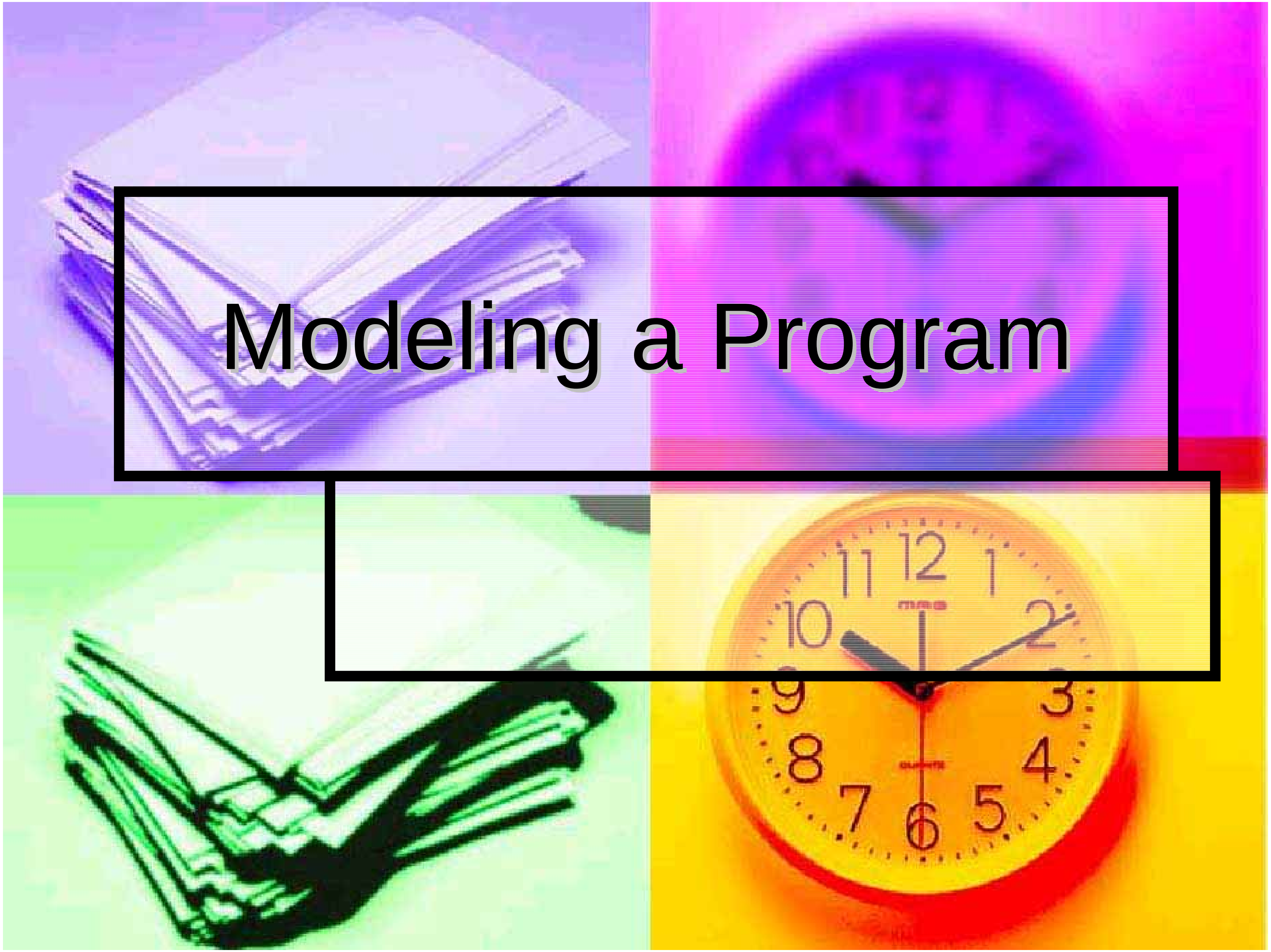
Techniques



Techniques

- Modeling a program
- Boolean programs (Microsoft’s SLAM)
- Abstraction refinement
- Parallel oriented model checker. (Lucent's VeriSoft, Bell Labs' SPIN)
- Framework (Kansas University’s Bandera)

Modeling a Program

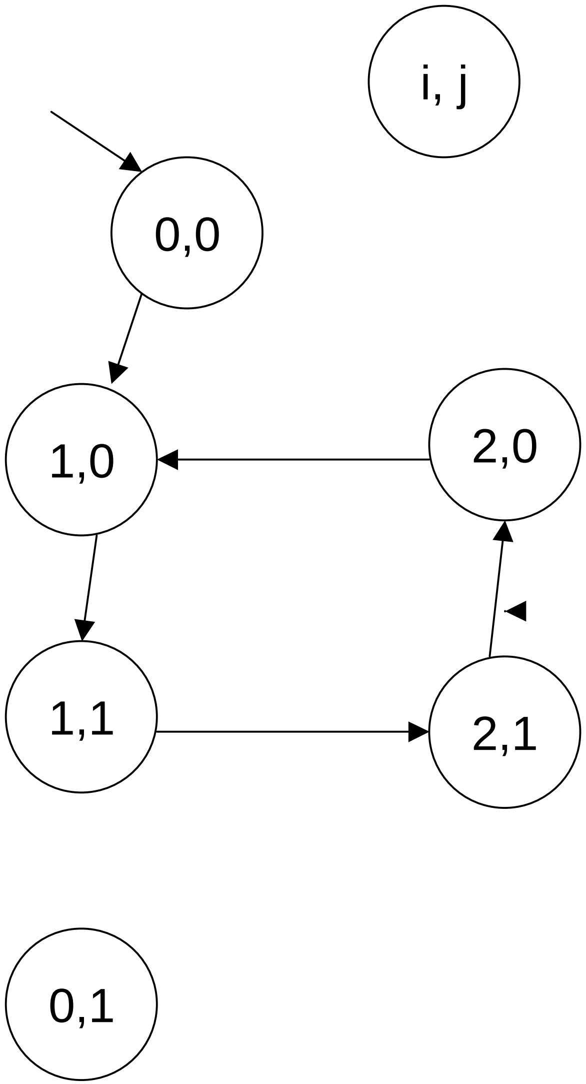


Modeling a program

- Model (like hardware) is synchronous – all variables change at once
- Software is sequential – one change at a time
- How can we translate a program to a model?

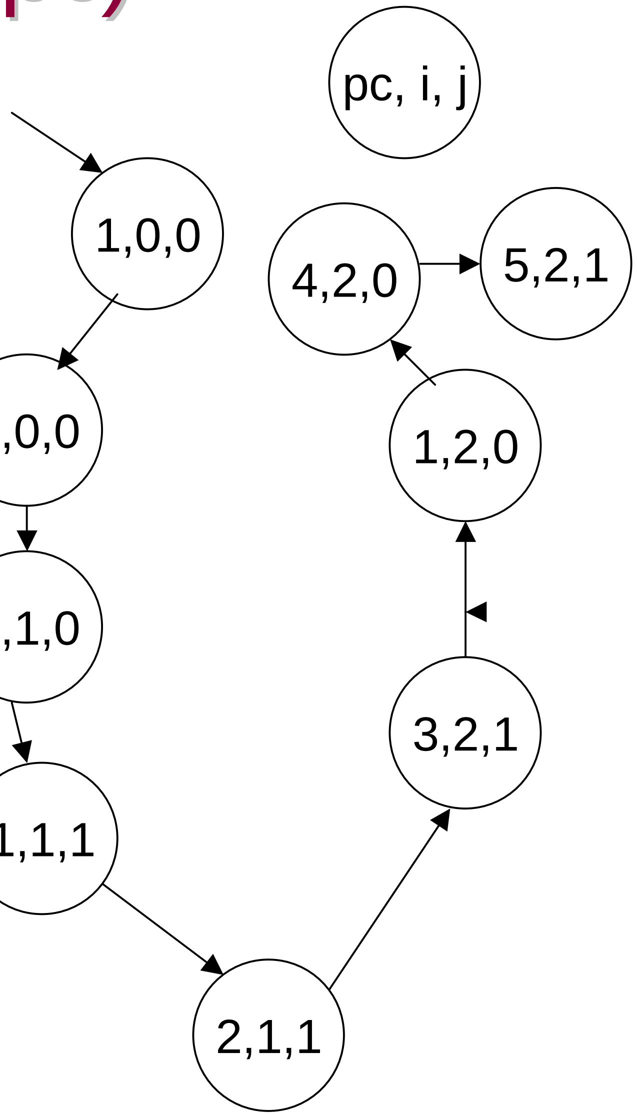
Example

```
bar () {
  int i=0, j=0;
   while (i<2) {
          i++;
          j=i%2;
   }
   i=1;
}
```



Example (Using pc)

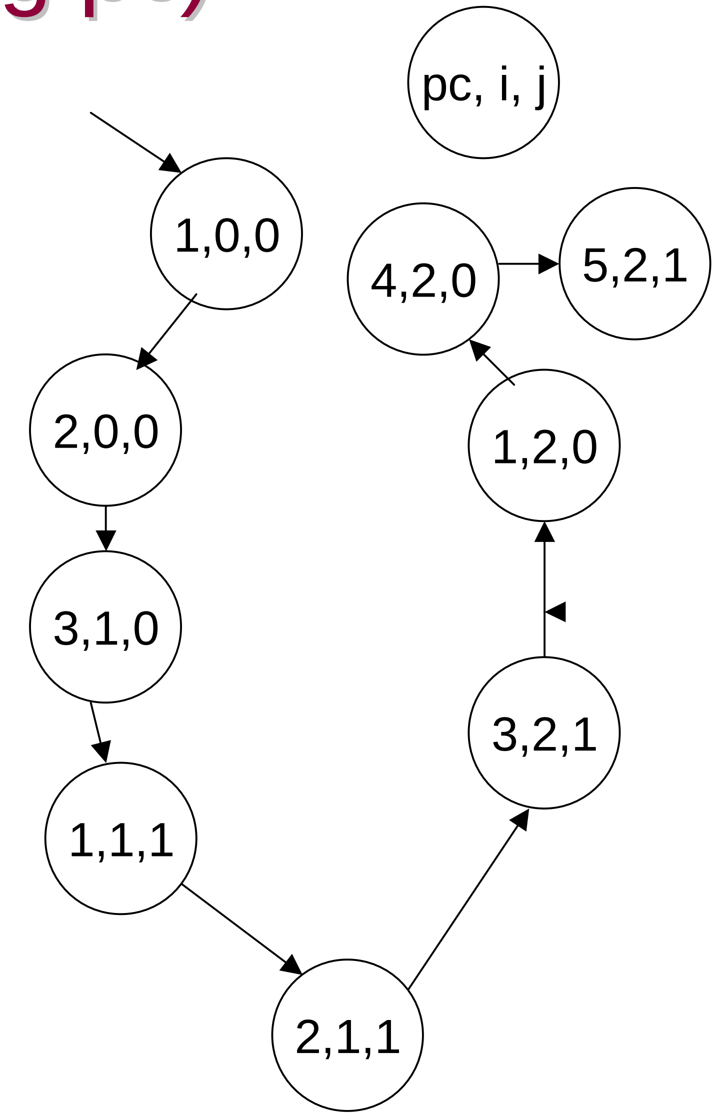
```
bar () {
  int i=0, j=0;
1 while (i<2) {
2         i++;
3         j=i%2;
  }
4 i=1;
}5
```



Example (Using pc)

```
next (i) =case
    pc=2 : i+1
    pc=4 : 1
    else  : i
next (j) =case
    pc=3 : i%2
    else  j
next (pc) = case
    pc=1  : if i<2 then 2
                       else 4
    pc=2   : 3
    pc=3   : 1
    pc=4   : 5
```

```
bar () {
  int i=0, j=0;
1  while (i<2) {
2        i++;
3        j=i%2;
   }
4  i=1;
}5
```



Boolean programs

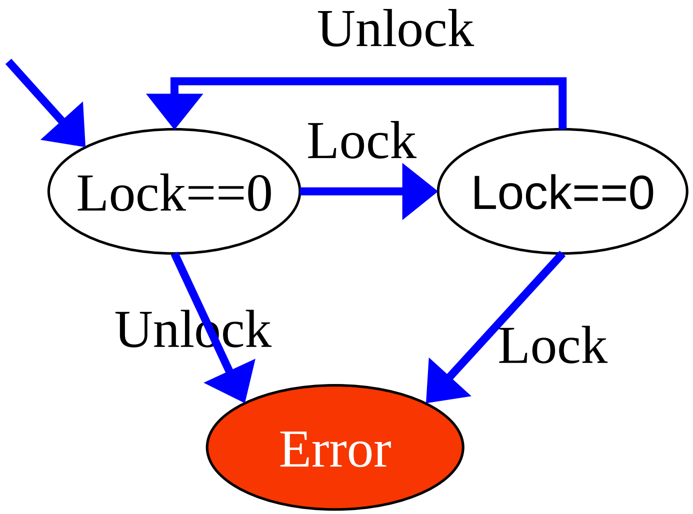
Microsoft’s SLAM

Boolean programs

- If we wanted to manually verify a program, we wouldn’t try to explore all of its states or run on all the inputs
- We would set some invariants and prove that they are kept throughout the program run
- Microsoft’s SLAM tries to do the same

Example – Lock mechanism

```
Lock {
  if (LOCK==1) error;
  else LOCK = 1;
}

Unlock {
  if (LOCK==0) error;
  else LOCK = 0;
}
```

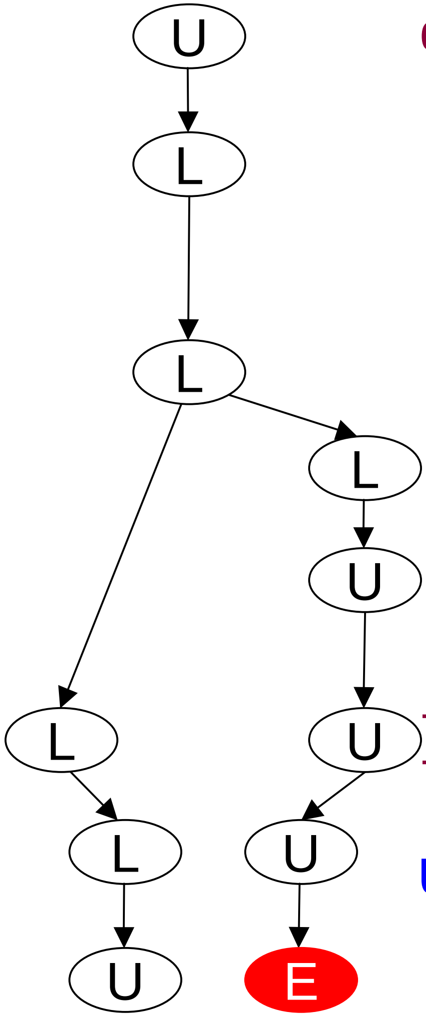


Example - Program

Does this code obey the locking rule?

```
do {
   Lock();

   nPacketsOld = nPackets;

   if(request){
       request = request->Next;
       Unlock();
       nPackets++;
   }
} while (nPackets != nPacketsOld);

Unlock();
```

Example – Boolean program

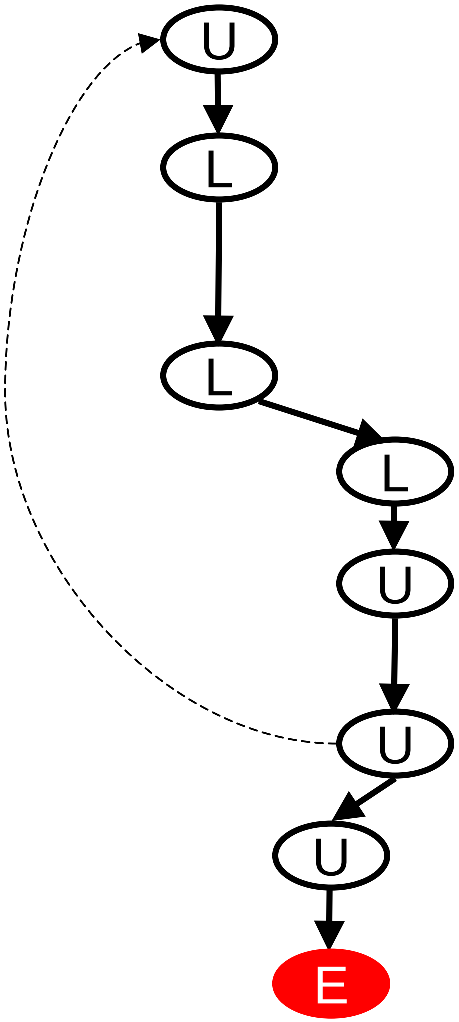



```
do {
   Lock();

   if(*){

      Unlock();

   }
} while (*);

Unlock();
```

Model checking
boolean program
(bebop)

Example – Reconstruction

Is error path feasible in C program? (newton)




```
do {
   Lock();

   nPacketsOld = nPackets;

   if(request){
      request = request->Next;
      Unlock();
      nPackets++;
   }
} while (nPackets != nPacketsOld);

Unlock();
```

nPackets = C
nPacketsOld = C

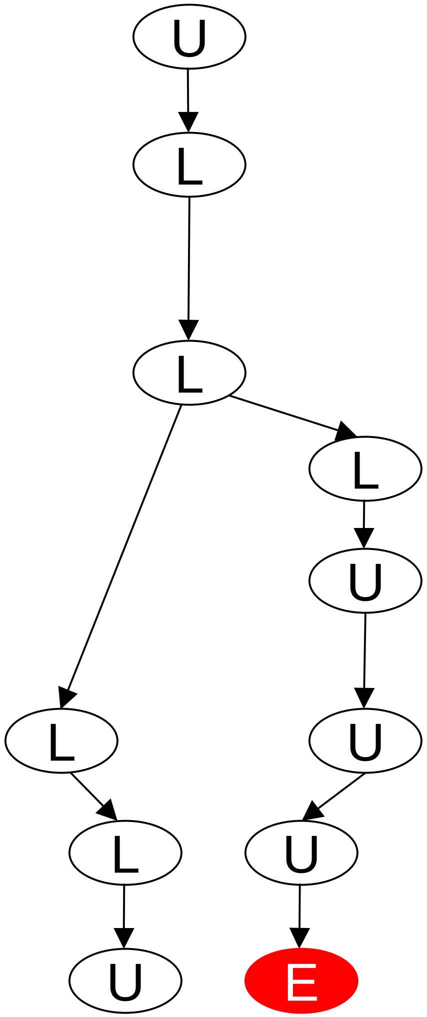
nPackets = C+1

Assume: C+1 == C

Example – Refinement

b : (nPacketsOld == nPackets)

Add new predicate to boolean program (c2bp)

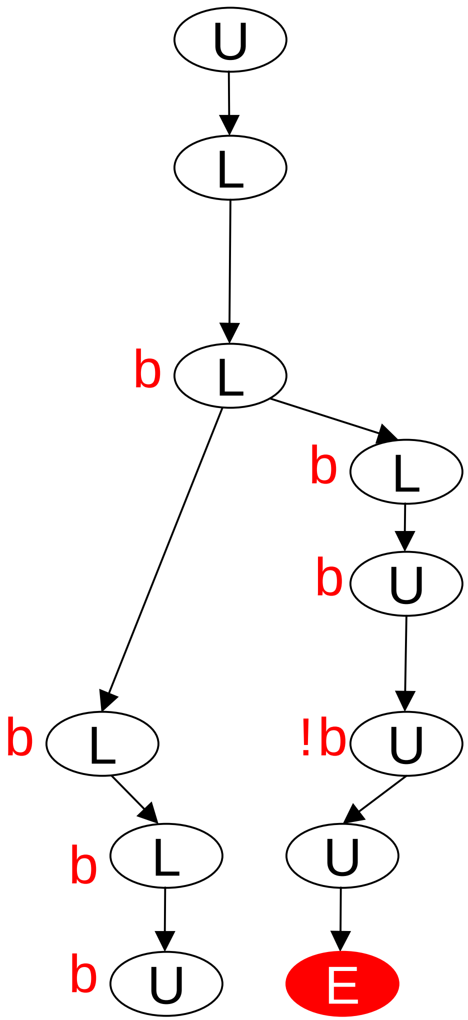



```
do {
   Lock();

   nPacketsOld = nPackets; b := true;

   if(request){
       request = request->Next;
       Unlock();
       nPackets++; b := b? false : *;
   }
} while (nPackets != nPacketsOld);// !b

Unlock();
```

Example

b : (nPacketsOld == nPackets)

Model checking
refined
boolean program
(bebop)

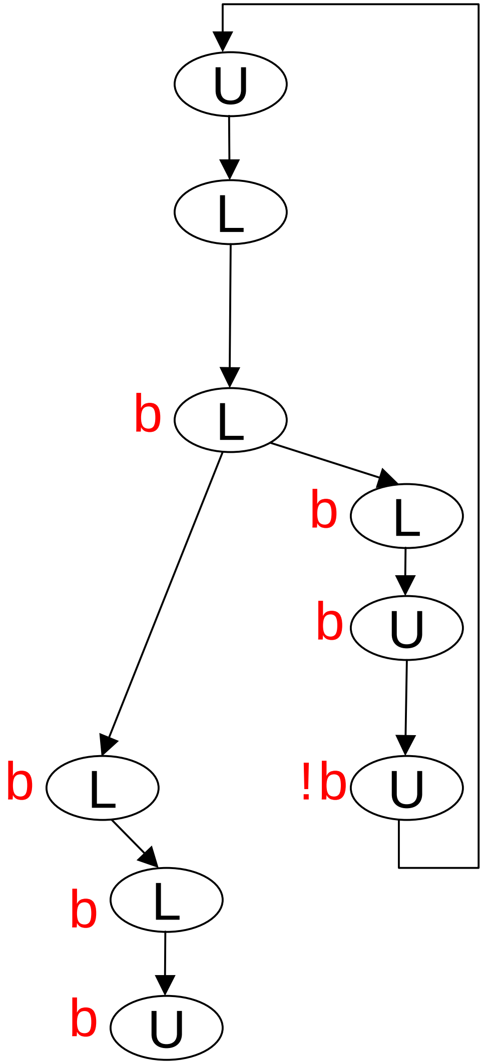



```
do {
   Lock();

   b := true;

   if(*){

      Unlock();
      b := b? false : *;
   }
} while ( !b );

Unlock();
```

Example

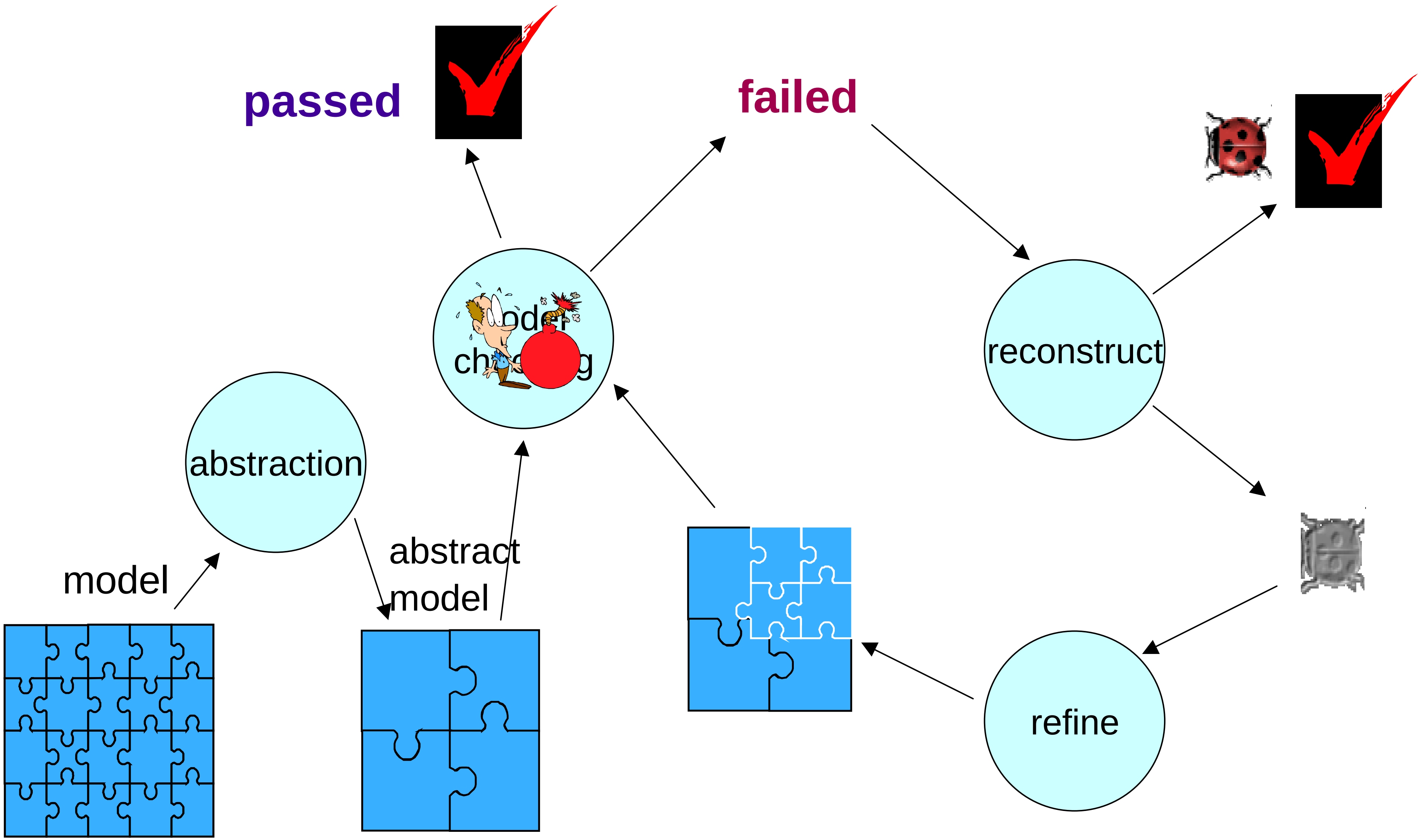
b : (nPacketsOld == nPackets)

Model checking refined boolean program (bebop)




```
do {
   Lock();

   b := true;

   if(*){

        Unlock();
      b := b? false : *;
   }
} while ( !b );

Unlock();
```

Abstraction refinement



Parallel oriented

Bell Labs' SPIN, Lucent's VeriSoft

Parallel oriented (VeriSoft)

- Parallel programs force us to encounter all possible interleavings – generates large models
- One of the common heuristics to reduce the model is partial-order reductions
- Mainly useful for explicit model checking

interleaving

Global vars : G,Z

a1: x=G

a2: x=0

a3: y=1

a4: Z=2

a5:

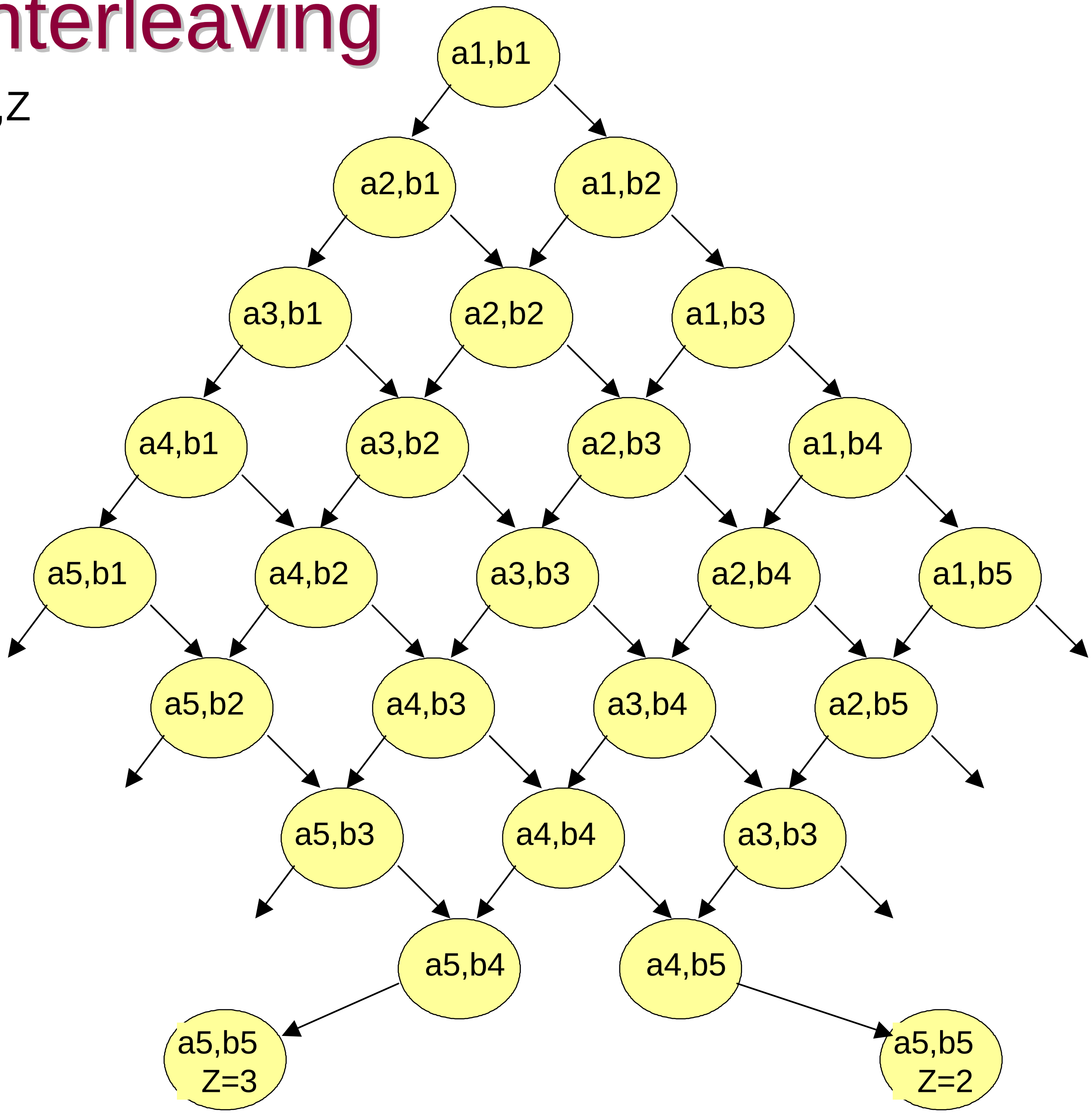
b1: p=G

b2: p=0

b3: q=1

b4: Z=3

b5:



Visible instructions

a1: x=G

a2: x=0

a3: y=1

a4: Z=2

a5:

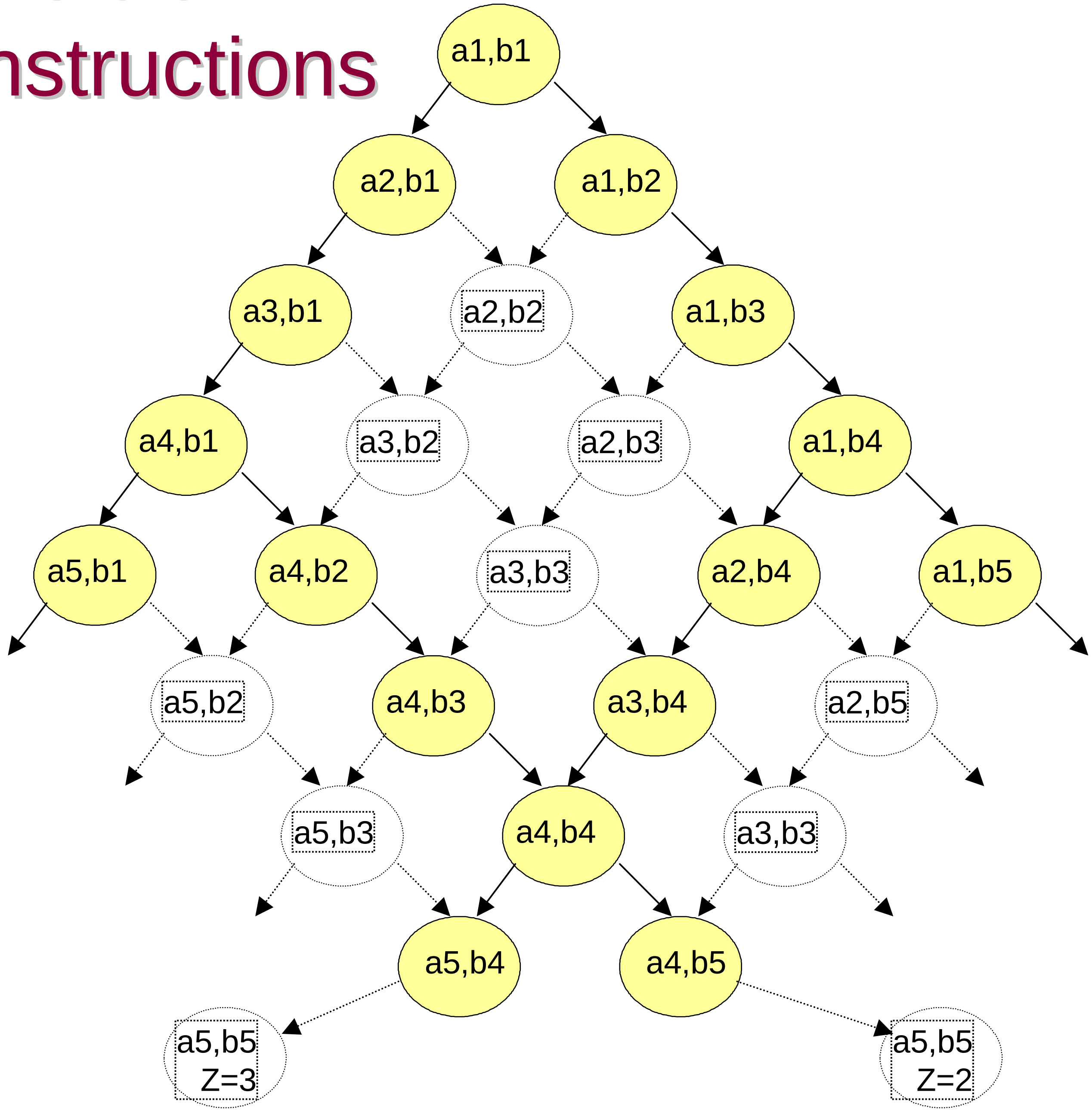
b1: p=G

b2: p=0

b3: q=1

b4: Z=3

b5:



Partial Order Reduction

a1: x=G

a2: x=0

a3: y=1

a4: Z=2

a5:

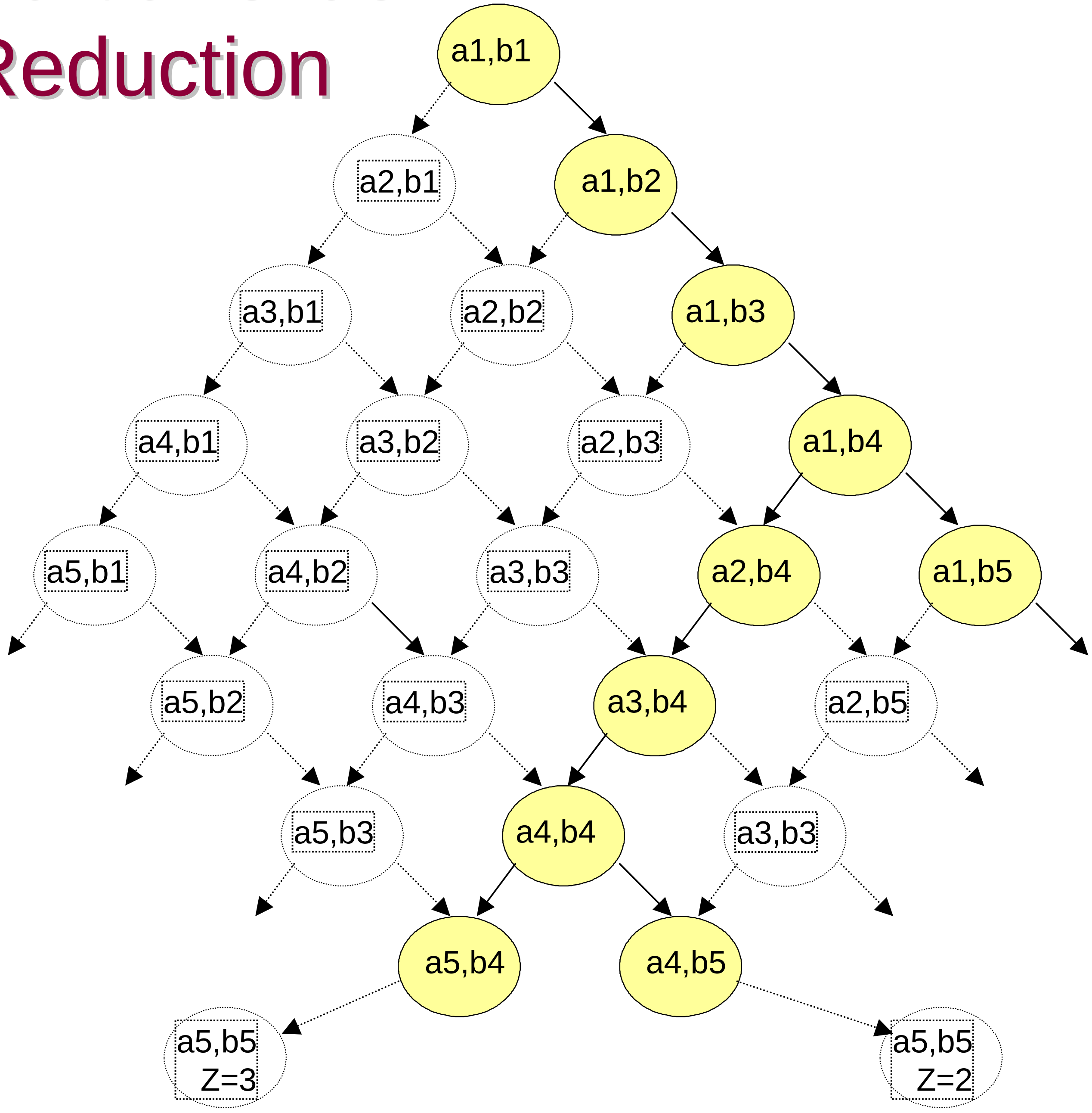
b1: p=G

b2: p=0

b3: q=1

b4: Z=3

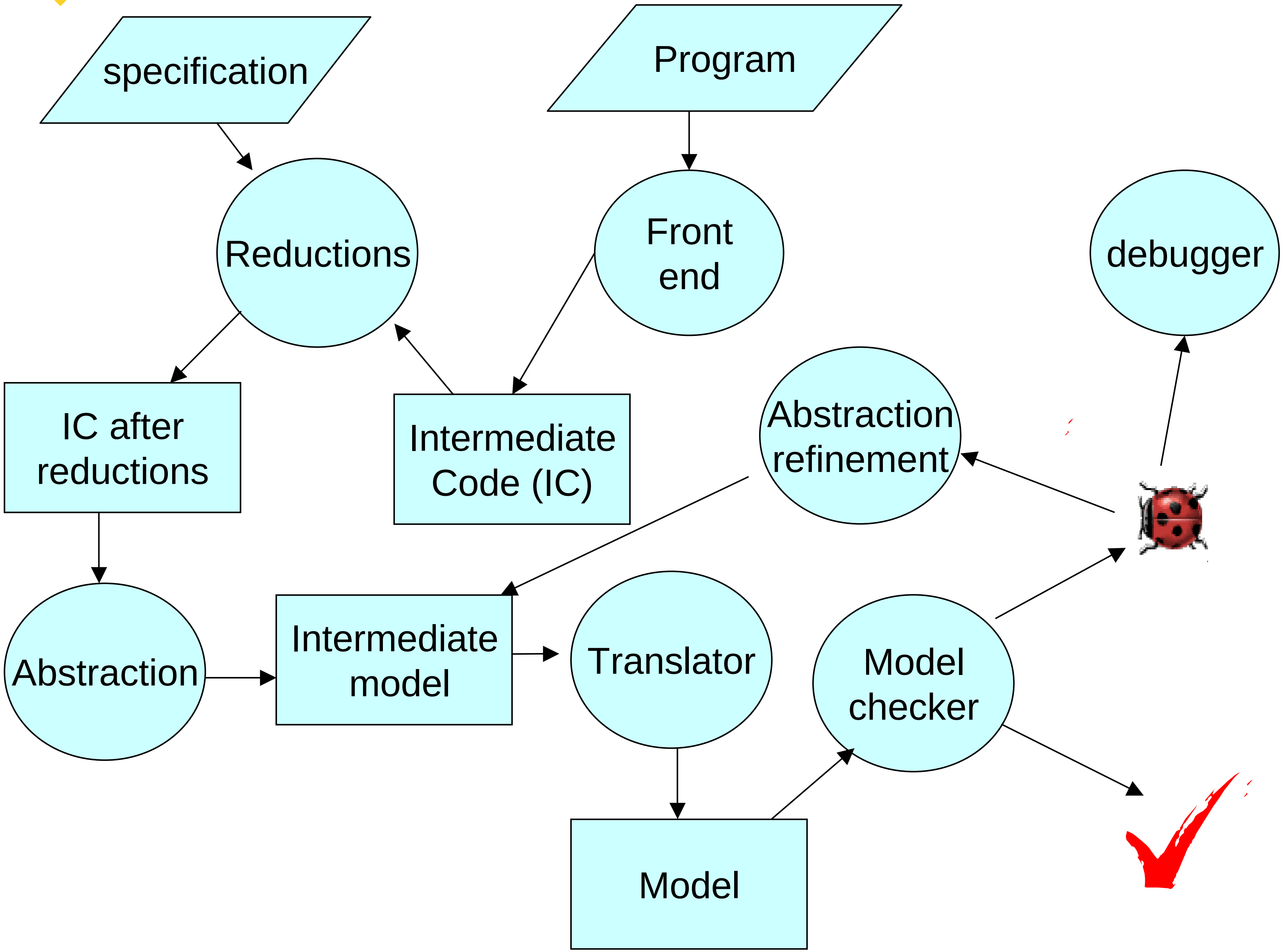
b5:



Framework

Kansas University Bandera-like

Framework



specification
Program
Reductions
Front end
debugger
IC after reductions
Intermediate Code (IC)
Abstraction refinement
Abstraction
Intermediate model
Translator
Model checker
Model

Here at IBM

- Using the power of RuleBase
- Translate C to EDL
- Support
  - Function + recursion
  - Pointers (no pointer arithmetic)
- Automatic specifications:
  - No infinite loops
  - No assert violations
  - No memory leaks
  - No access to dangling pointers
  - No out of bound access to arrays