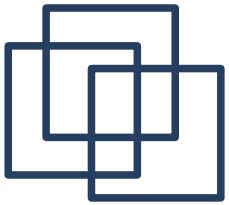
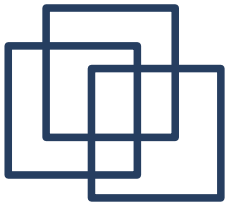


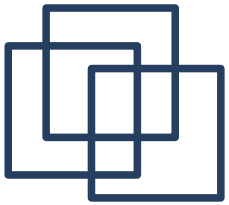
Verification by Abstraction Refinement Techniques

Emmanuel Fleury
LaBRI, CNRS UMR 5800
<emmanuel.fleury@labri.fr>

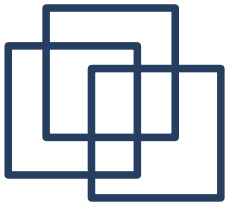


Outline

- Motivations
- Model-checking
- Predicate Abstraction
- Lazy Abstraction
- Lazy Abstraction at Work

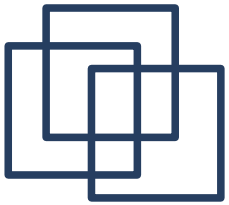


Motivations



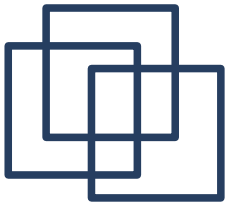
Our Goal ?


Rapid
A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue _



Our Goal ?

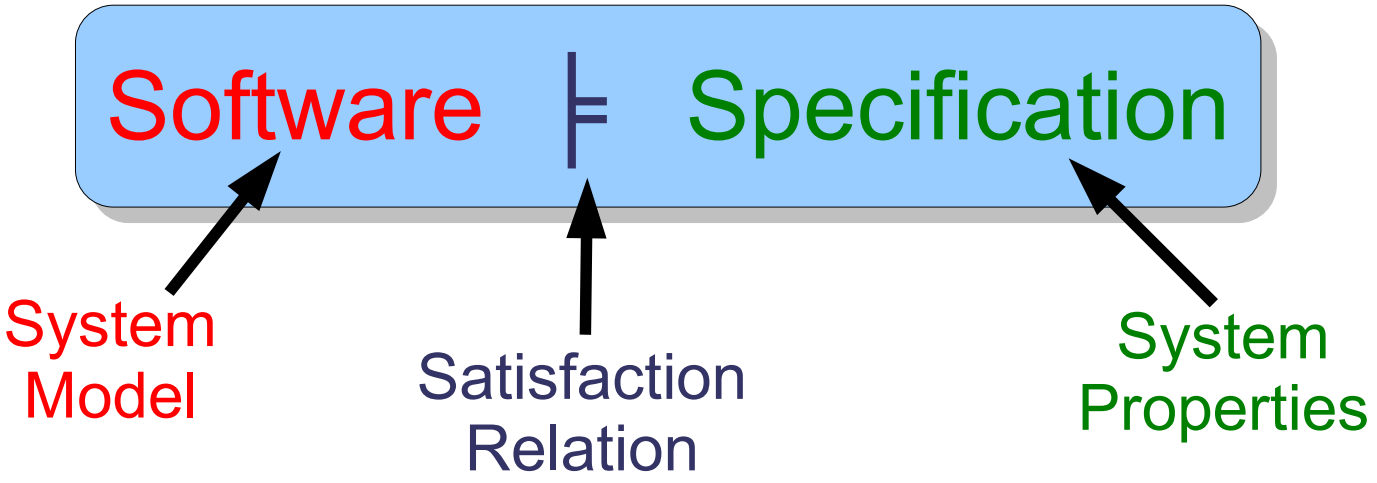
Hunting Program BUGS !

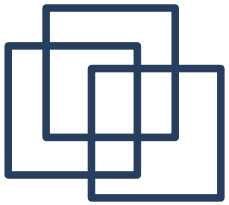


Our Goal

We do not aim at having bug free softwares!

We want to ensure some behaviours of the software





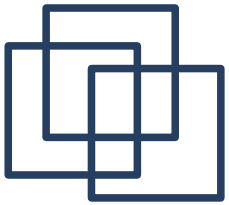
Our Goal ?

Software Certification

- Warranty error proneness of critical programs
- Requires precise and formal specifications
- Highly impact the development process

Software Quality

- Improve development process by catching programmers mistakes
- Requires formal specification of common annoying mistakes
- Automatically fill bug-reports for developpers



Verification Techniques

Static Analysis

Verify properties based on the source code of the software

Abstract Interpretation

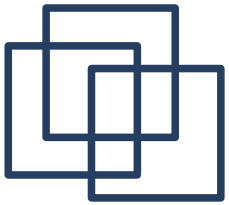
Verify properties based on an “safe” abstraction of the software (“safe” with respect to a *Galois connection*)

Theorem Proving

Verify properties on a mathematical representing the software with the help of a Proof Assistant

Model-checking

Verify properties on a transition system representing the software and through an exhaustive search through it



But...

Programming languages are Turing equivalent

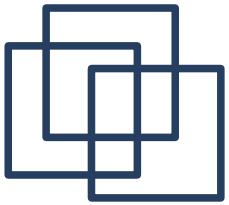
Most of the interesting properties are undecidable in a Turing-equivalent language (accessibility, liveness, ...)

Complexity of real programs is high

Size and complexity of the programs is growing exponentially with the time

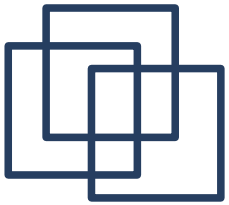
Verification techniques usually doesn't scale

Most of the techniques are used on hand-made abstract models and cannot cope with real ones.

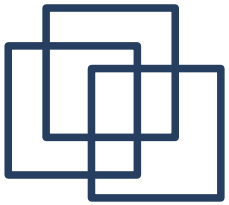


So… ?

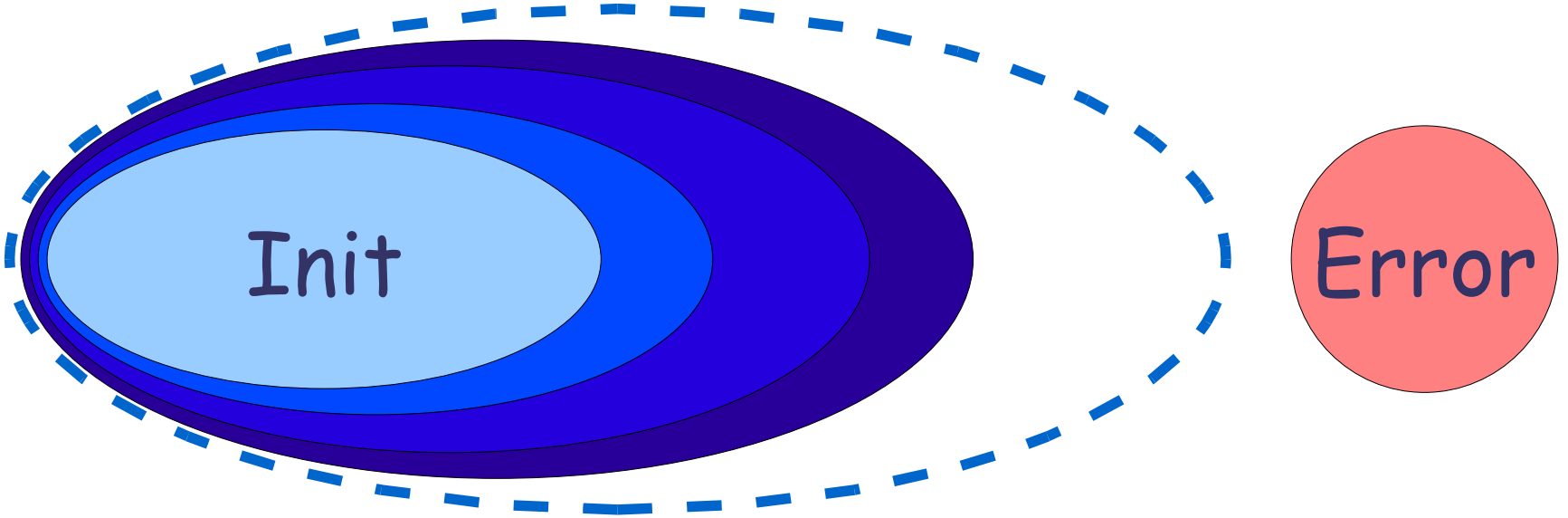
Is there still hope ?



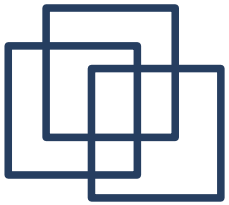
Model-Checking



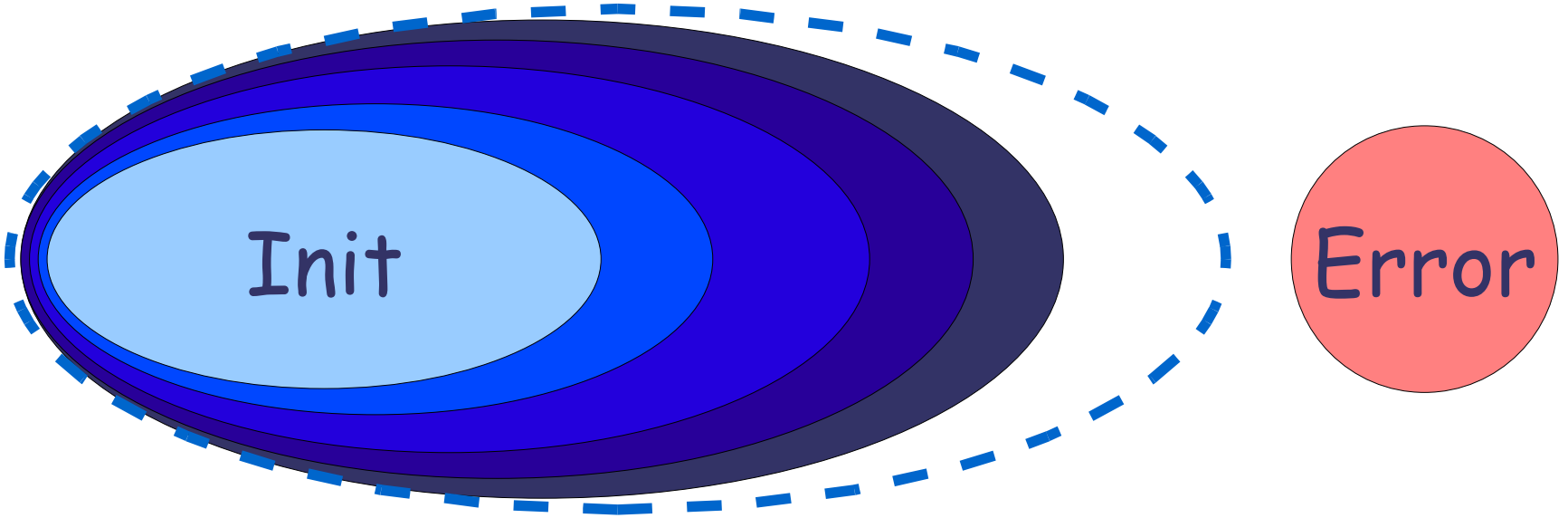
Model-Checking Basics



Iteration doesn't terminate ! (state explosion)



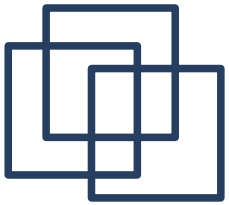
Model-Checking Basics



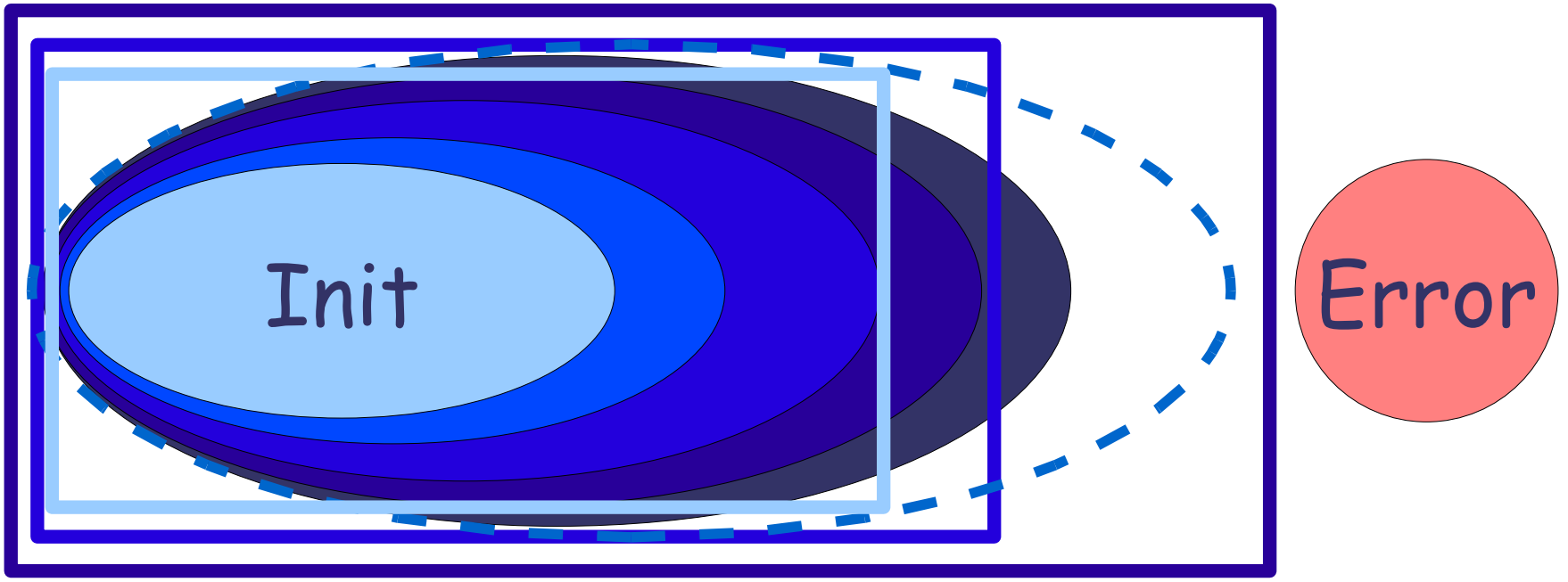
Iteration doesn't terminate ! (state explosion)

Solutions:

- Acceleration (make several steps at once)



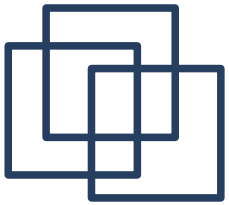
Model-Checking Basics



Iteration doesn't terminate ! (state explosion)

Solutions:

- Acceleration (make several steps at once)
- Abstraction (remove unnecessary details)



Can We Do Better ?

State explosion problem ! (doesn't scale)

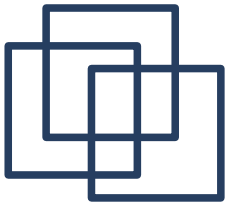
- Especially for complex C code programs
- Use Abstraction and Acceleration... How ?

Works only for finite transition systems

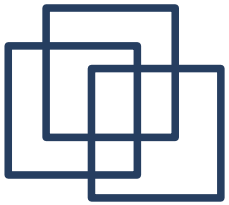
- Find a way to explore infinite systems
- Use Abstraction... How ?

Automatic Abstraction

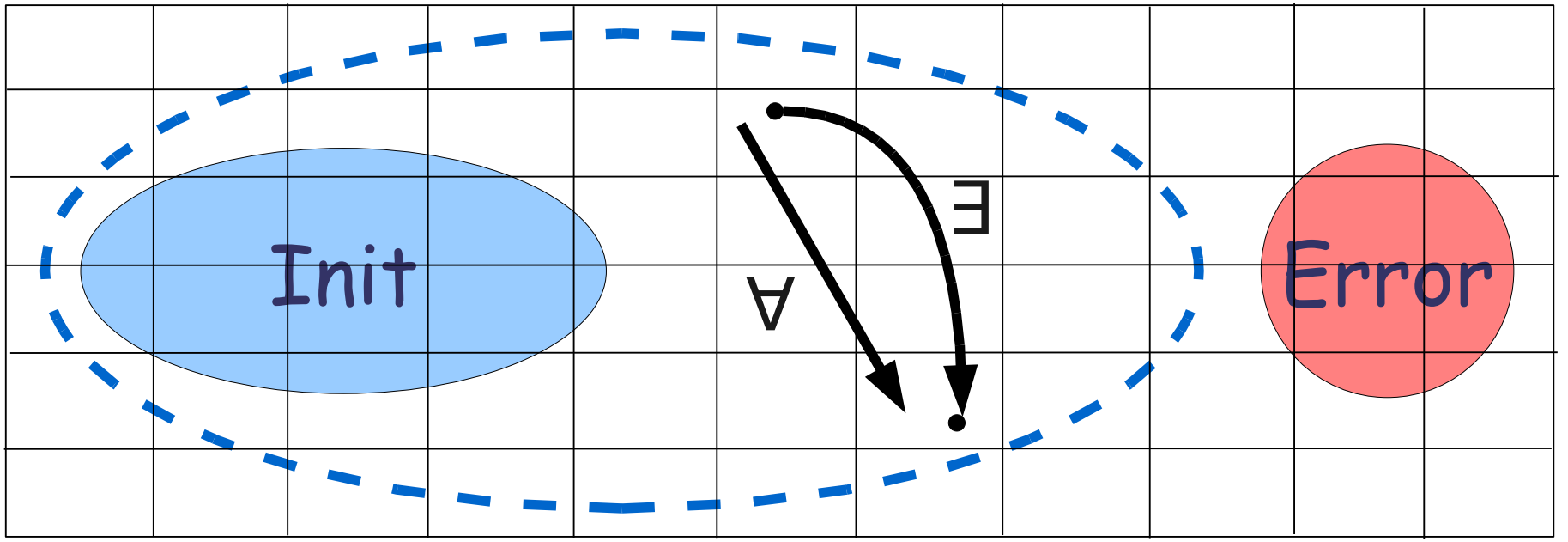
- Automatically derive the abstraction
- Use Predicate Abstraction ?



Predicate Abstraction



Predicate Abstraction

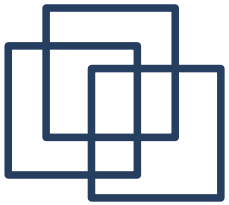


Predicates are boolean expressions:

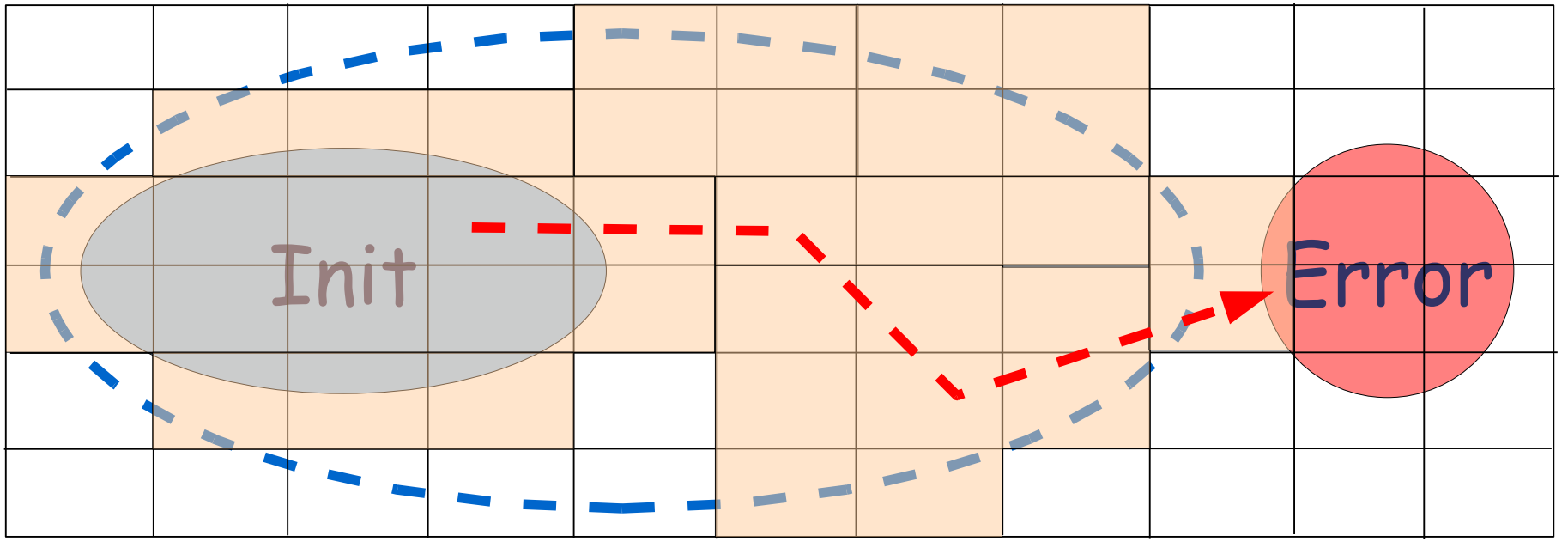
P_i **->** {true, false}, e.g. x=1, x>1, x<1 or z=y

Abstract States (boxes) are valuations:

$\{P_1, P_2, ..., P_n\}$ **->** $\{true, false\}^n$



Predicate Abstraction

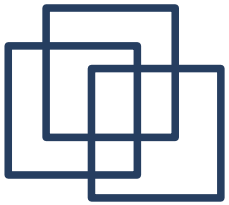


Predicates are boolean expressions:

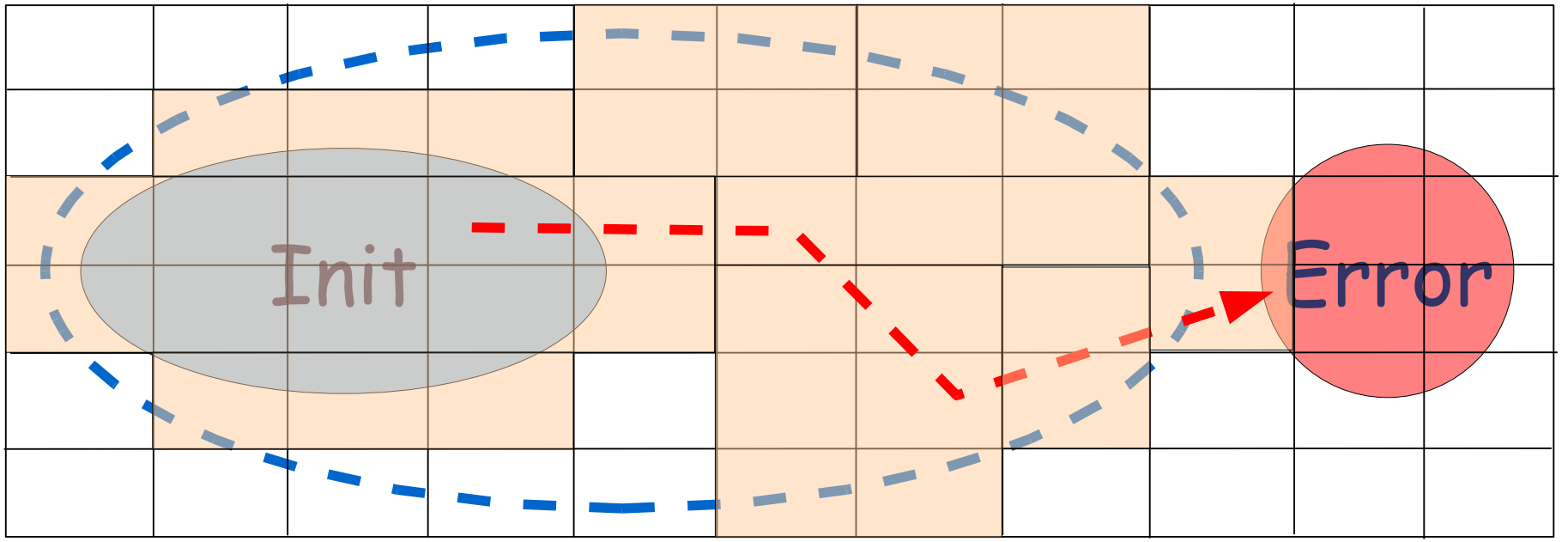
P_i **->** {true, false}, e.g. x=1, x>1, x<1 or z=y

Abstract States (boxes) are valuations:

$\{P_1, P_2, ..., P_n\}$ **->** $\{true, false\}^n$



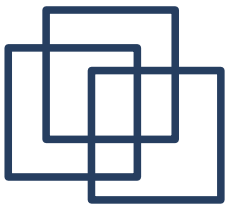
Predicate Abstraction



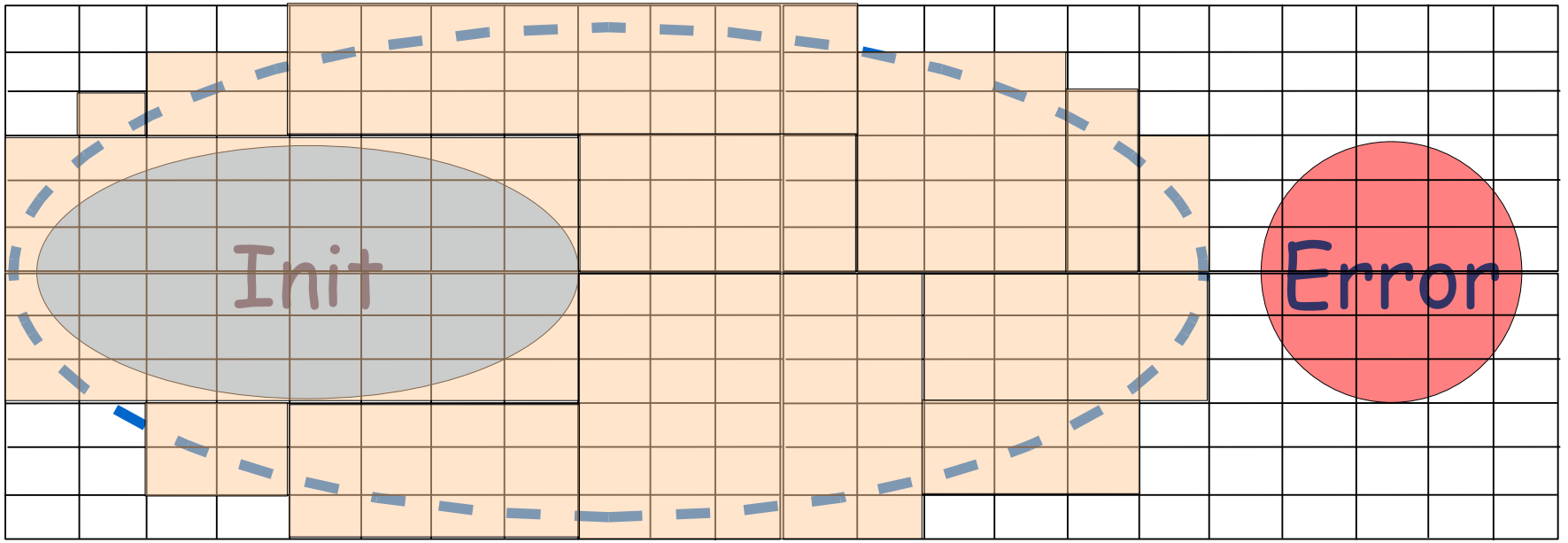
Abstraction too coarse !!!

Refinement is needed to avoid spurious error trace

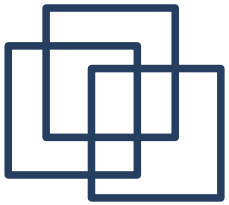
Split boxes into smaller ones (add new predicates)



Predicate Abstraction



No Error state reached.
The program is correct !



Can We Do Better ?

Abstract only where required

- Reachable state space is *very* sparse
- **Construct the abstraction on-the-fly**

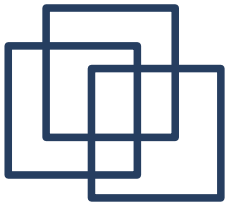
Refinement only when required

- Different precisions/abstractions for different regions
- **Refine locally**

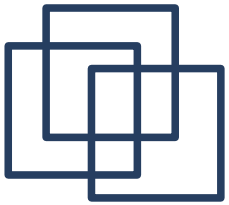
Reuse work from earlier phases

- Batch-oriented (lose work from previous runs)
- **Integrate the three phases**

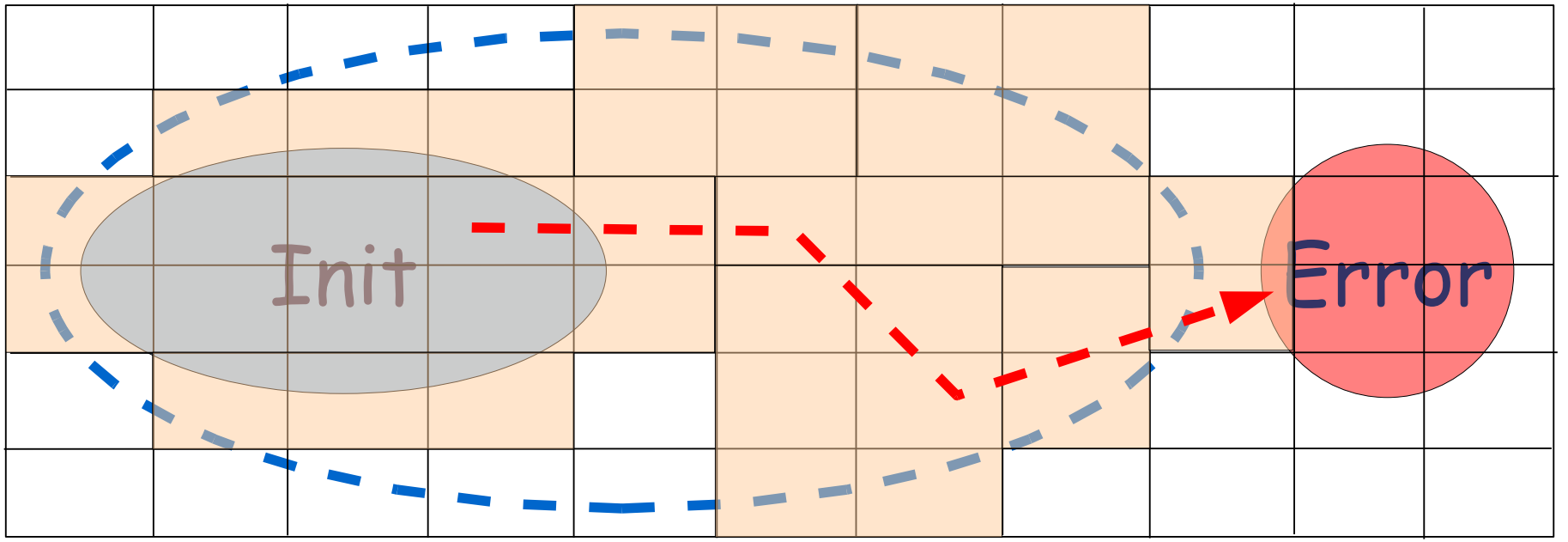
Exploit control flow structure



Lazy Abstraction



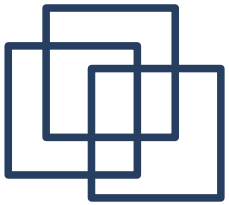
Lazy Abstraction



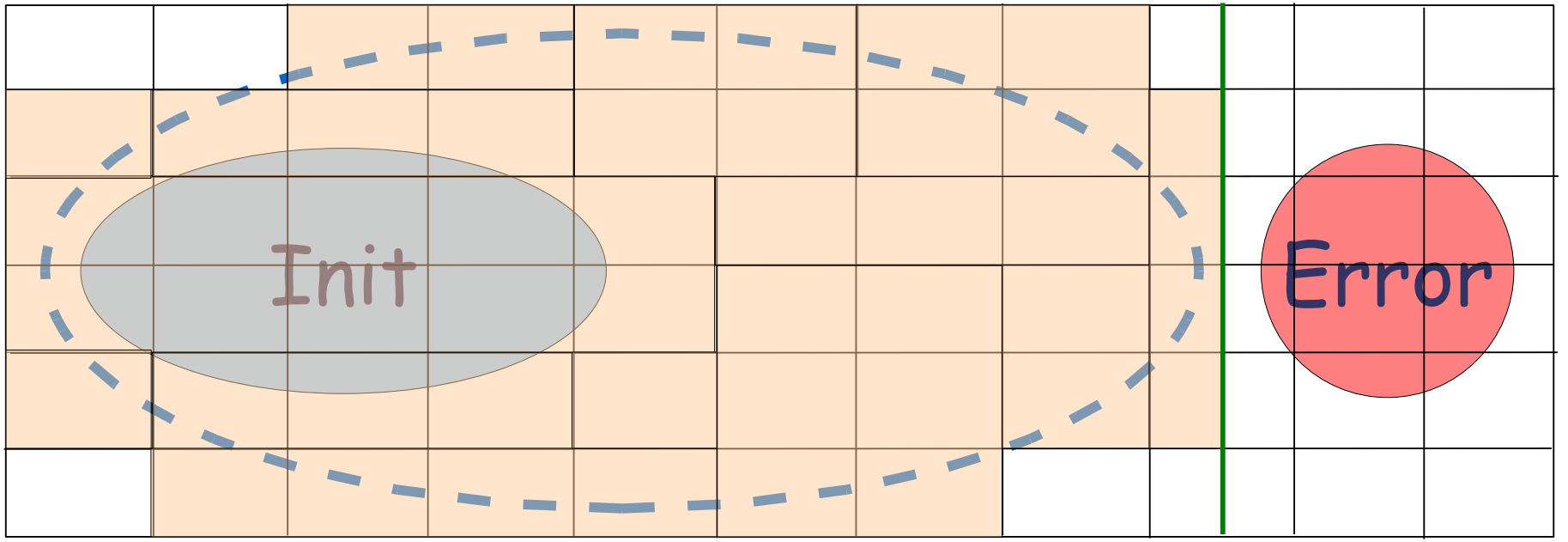
Abstraction too coarse !!!

Refinement is needed to avoid spurious error trace

Split **only the relevant boxes** into smaller ones
(add new predicates)

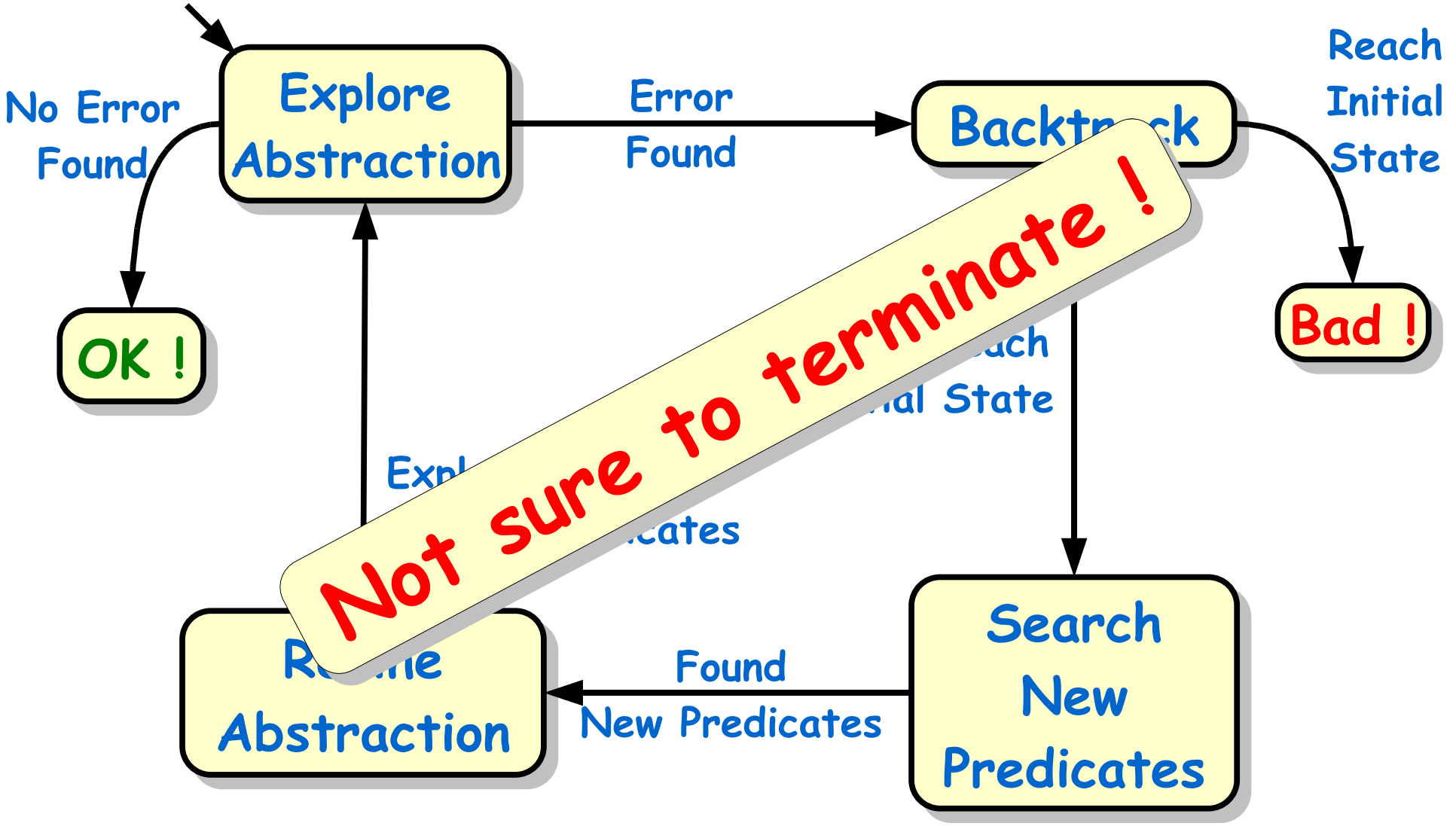


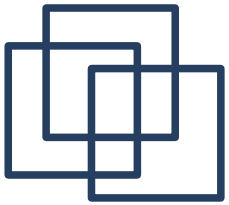
Lazy Abstraction



No Error state reached.
The program is correct !

The Algorithm

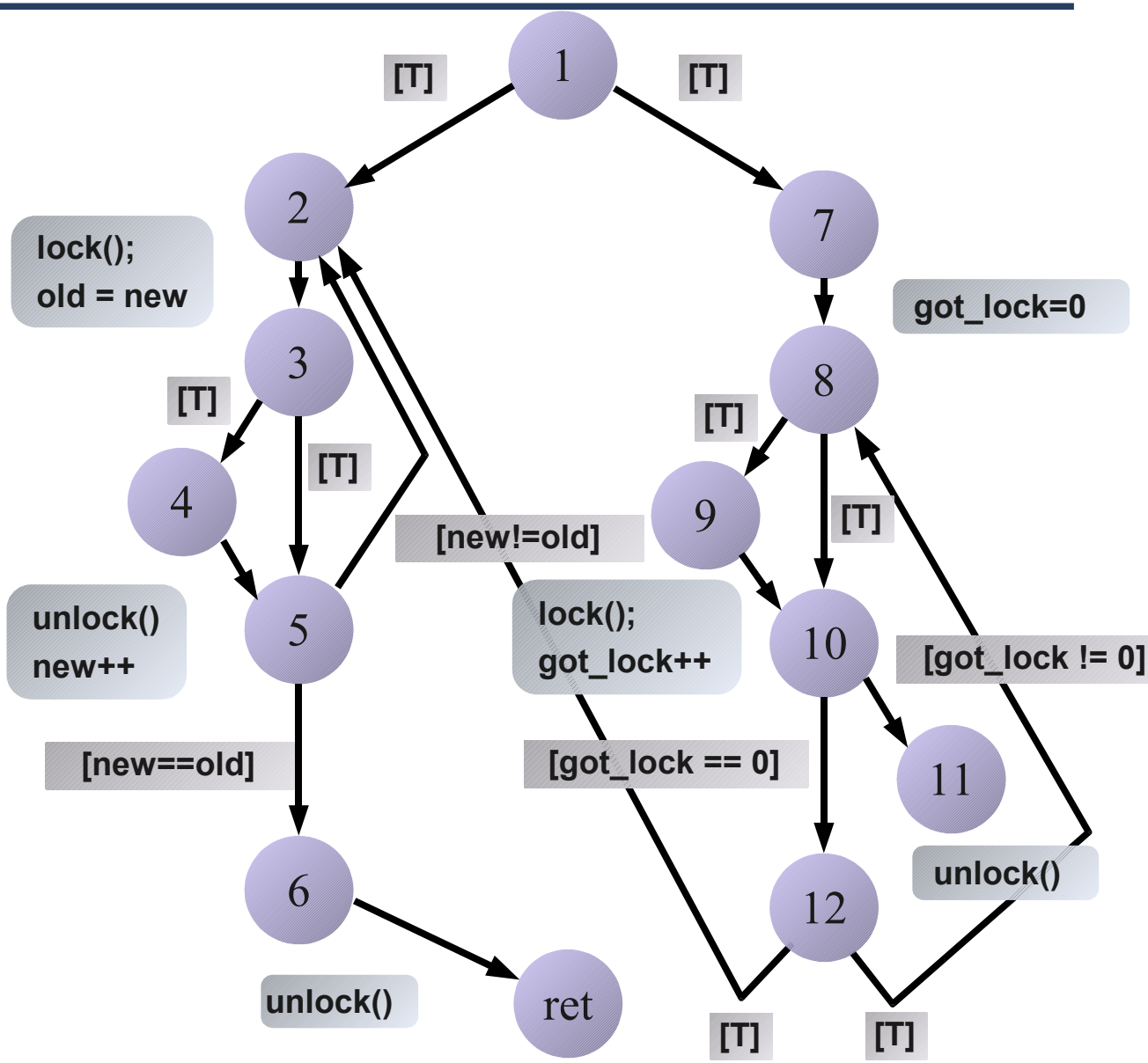




Lazy Abstraction at Work

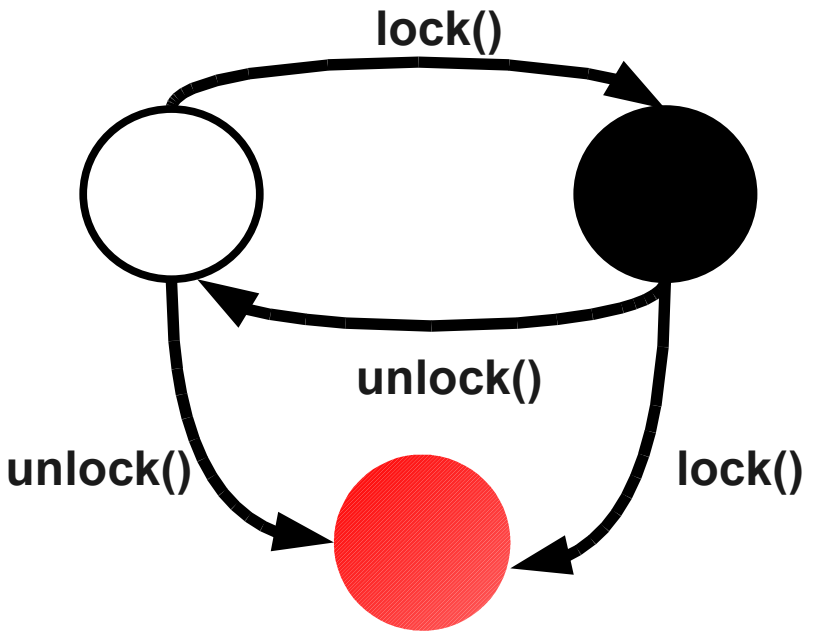
Control Flow Automata

```
Example ( ) {
1:    if (*) {
7:        do {
              got_lock = 0;
8:            if  (*) {
9:                lock();
                  got_lock ++;
              }
10:           if (got_lock) {
11:               unlock();
              }
12:       } while (*) ;
      }
2:    do {
          lock();
          old = new;
3:        if (*) {
4:            unlock();
              new ++;
          }
5:    } while ( new != old);
6:    unlock ();
      return;
}
```



Observer (Specification)

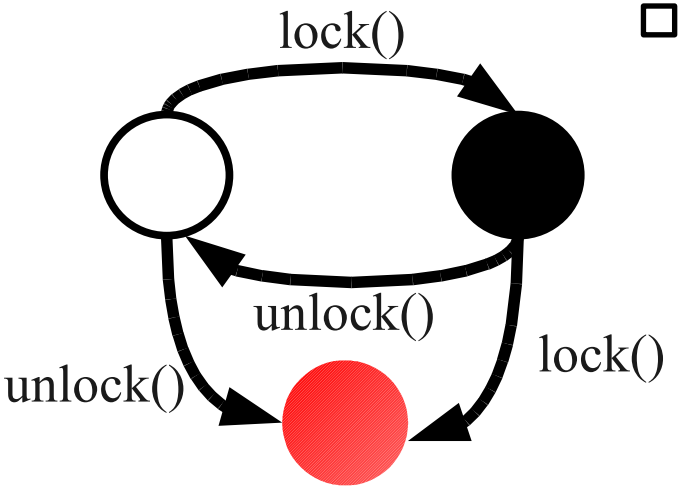
```
Example ( ) {
1:    if (*) {
7:        do {
                got_lock = 0;
8:              if  (*) {
9:                  lock();
                    got_lock ++;
                }
10:             if (got_lock) {
11:                 unlock();
                }
12:       } while (*) ;
      }
2:    do {
            lock();
            old = new;
3:          if (*) {
4:              unlock();
                new ++;
            }
5:    } while ( new != old);
6:    unlock ();
      return;
}
```

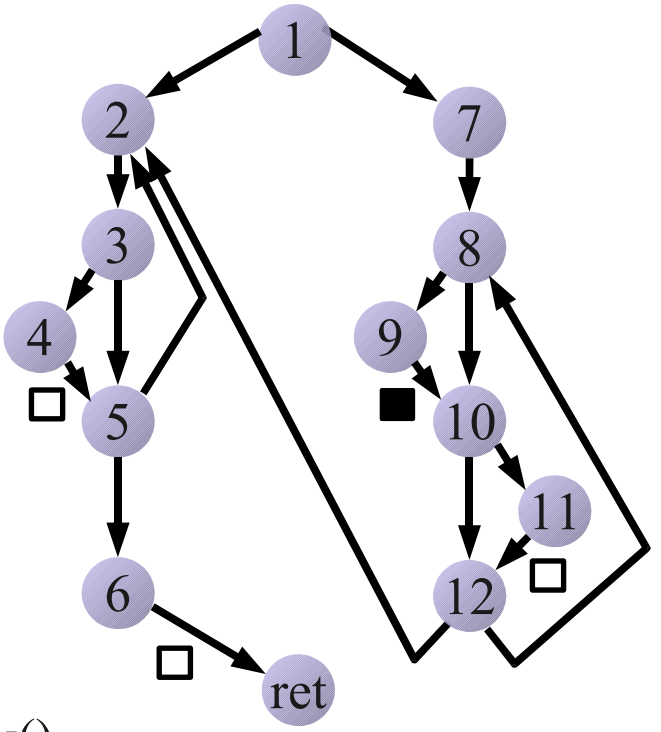


Q: Is Error Reachable ?

Example of Observer

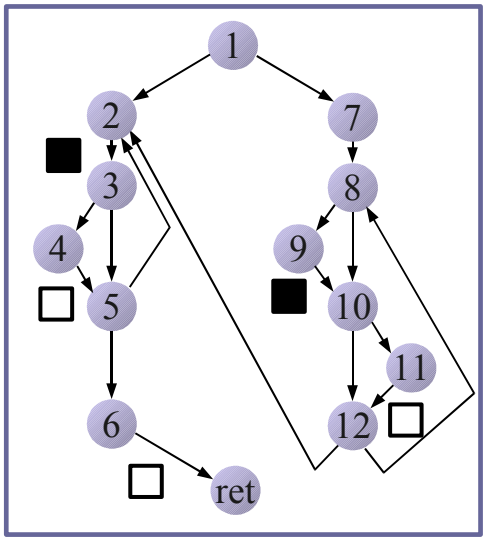
```
Example ( ) {
1:    if (*) {
7:        do {
                got_lock = 0;
8:              if  (*) {
9:                  lock();
                    got_lock ++;
                }
10:             if (got_lock) {
11:                 unlock();
                }
12:       } while (*) ;
      }
2:    do {
            lock();
            old = new;
3:          if (*) {
4:              unlock();
                new ++;
            }
5:    } while ( new != old);
6:    unlock ();
      return;
}
```





Q: Is Error Reachable ?

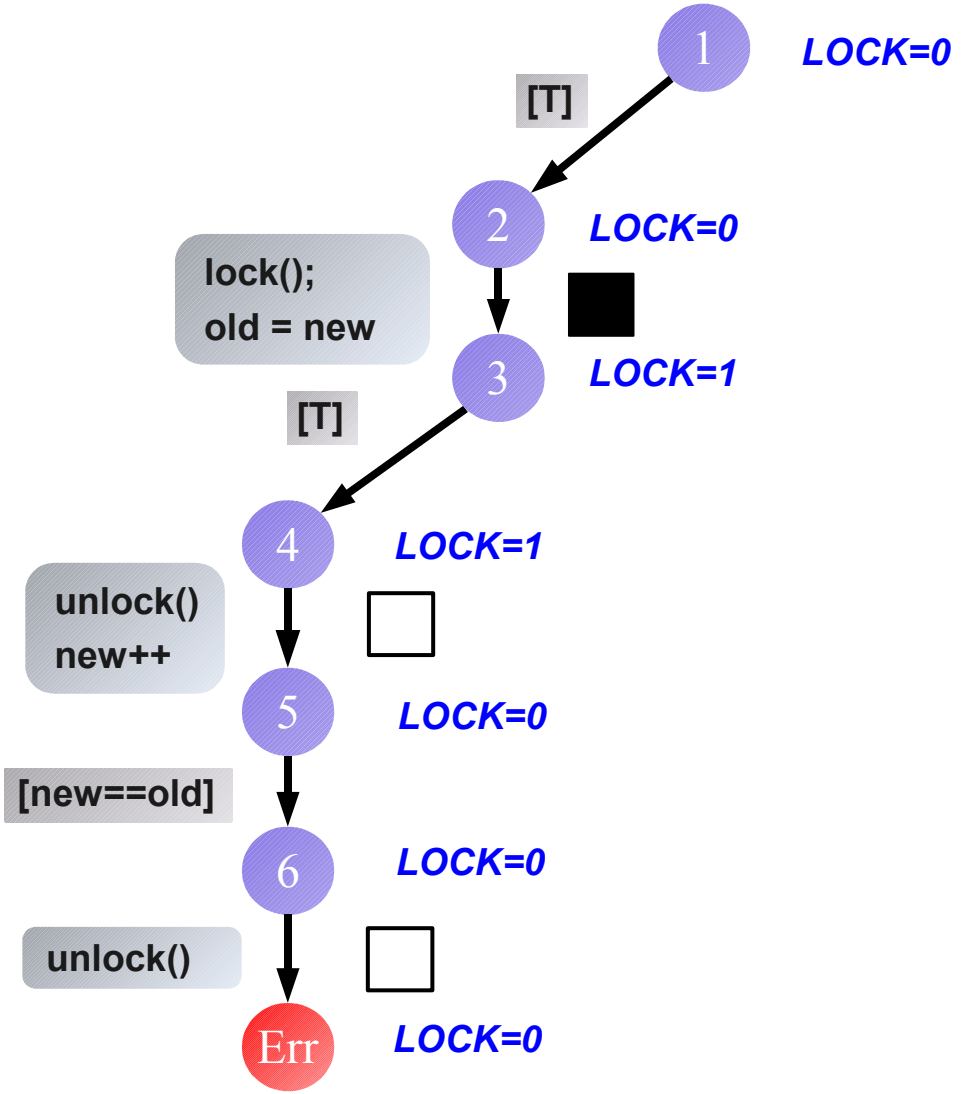
Step 1: Forward Search



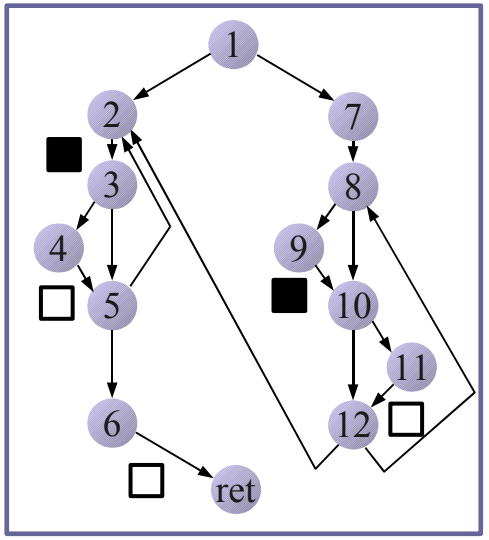
Q: Is the error trace spurious ?

Set of predicates:

LOCK=0, LOCK=1

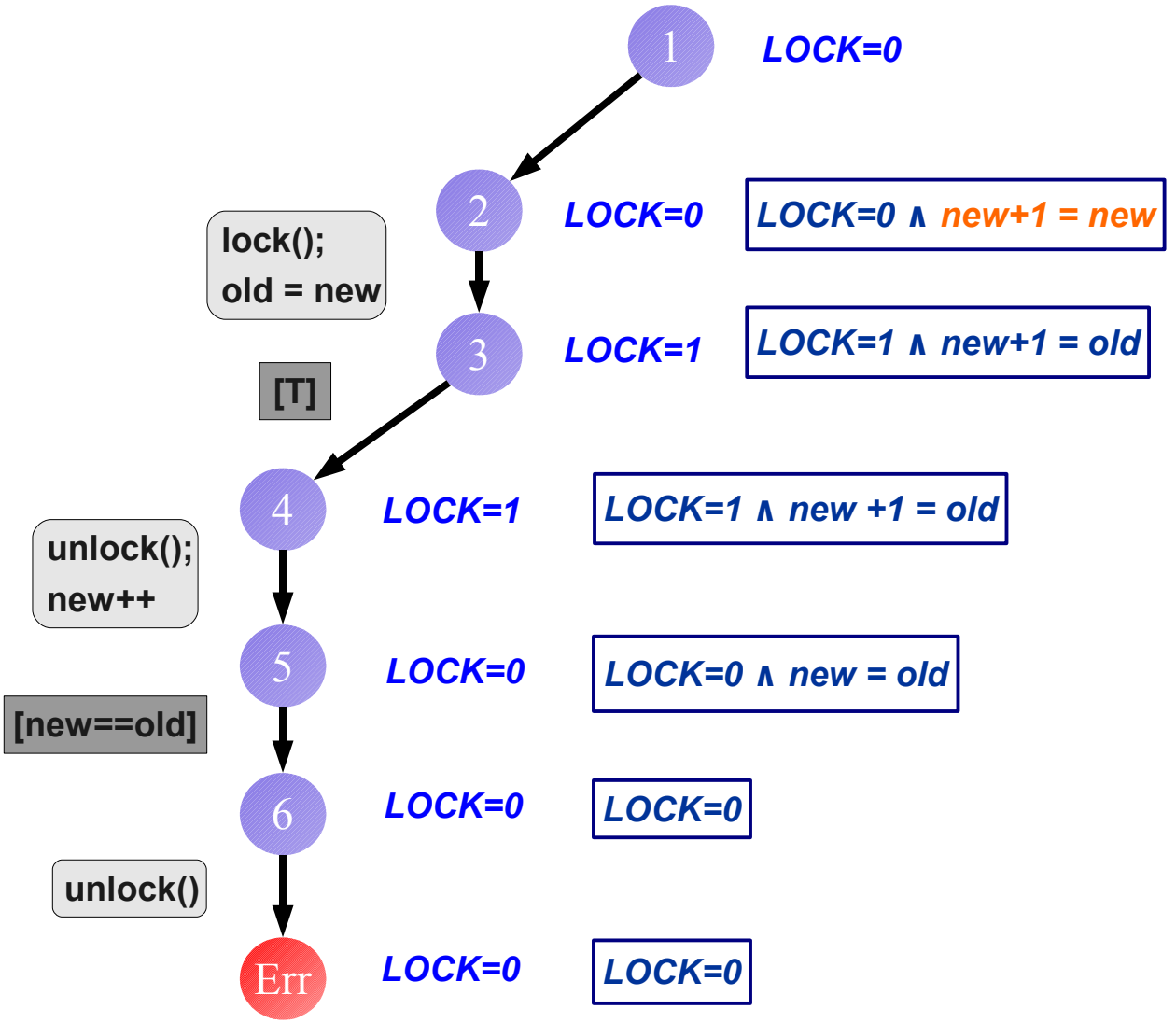


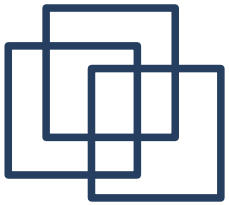
Step 2: Counter Example Analysis



Track the predicate:

new = old





Craig's Interpolation Theorem [Craig '57]

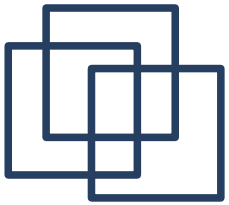
Given formulas Ψ^-, Ψ^+ s.t. $\Psi^- \wedge \Psi^+$ is unsatisfiable

There exists an interpolant Φ to Ψ^-, Ψ^+ s.t.

- Ψ^- implies Φ
- $\Phi \wedge \Psi^+$ is unsatisfiable
- Φ has common symbols from Ψ^- and Ψ^+

Φ is computable from the proof of unsatisfiability of $\Psi^- \wedge \Psi^+$

[Krajicek'97][Pudlak'97]
(boolean) SAT-based Model Checking [McMillan'03]



Finding the new Predicate

Trace	Trace Formula
pc_1: x = ctr	$x_1 = ctr_0$
pc_2: ctr = ctr + 1	$\wedge \quad ctr_1 = ctr_0 + 1$
pc_3: y = ctr	$\wedge \quad y_1 = ctr_1$
	$\wedge \quad x_1 = i_0 - 1$
pc_4: assume(x = i-1)	$\wedge \quad y_1 \neq i_0$
pc_5: assume(y ≠ i)	

ψ^-

ψ^+

Interpolate → Φ

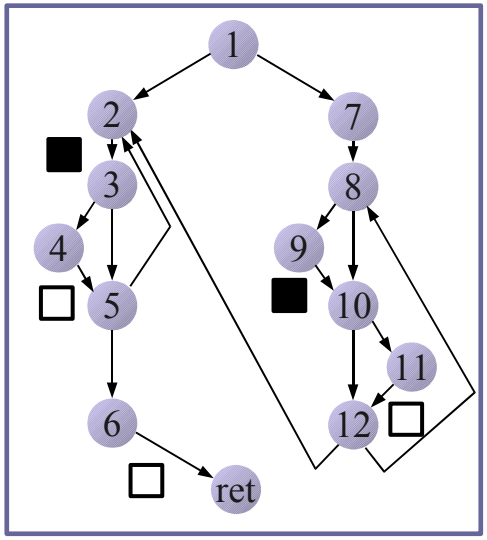
Require:

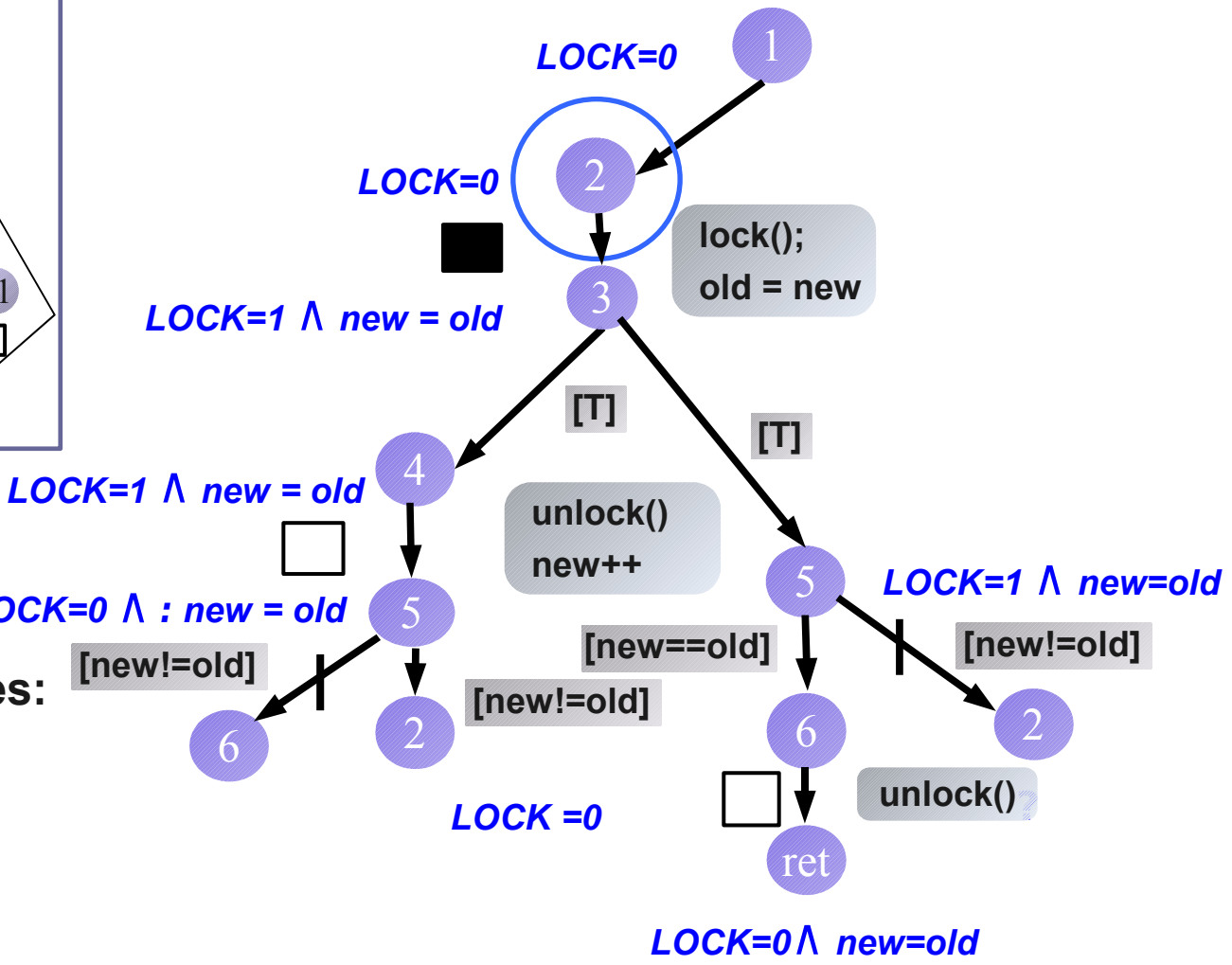
1. Predicate ***implied*** by trace ***prefix***
2. Predicate on ***common*** variables
   common = ***current*** value
3. Predicate & ***suffix*** yields a ***contradiction***

Interpolant:

1. ψ^- ***implies*** Φ
2. Φ has symbols ***common*** to ψ^-, ψ^+
3. $\Phi \wedge \psi^+$ is ***unsatisfiable***

Step 3: Resume Search





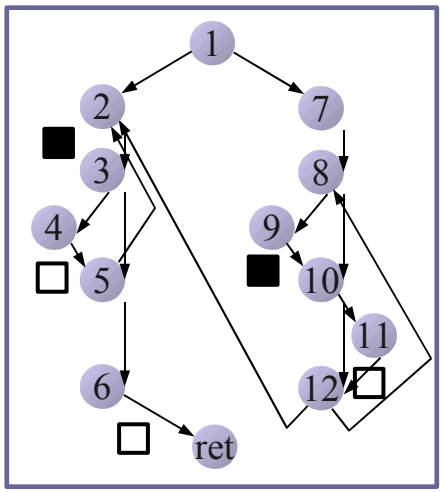
Set of predicates:

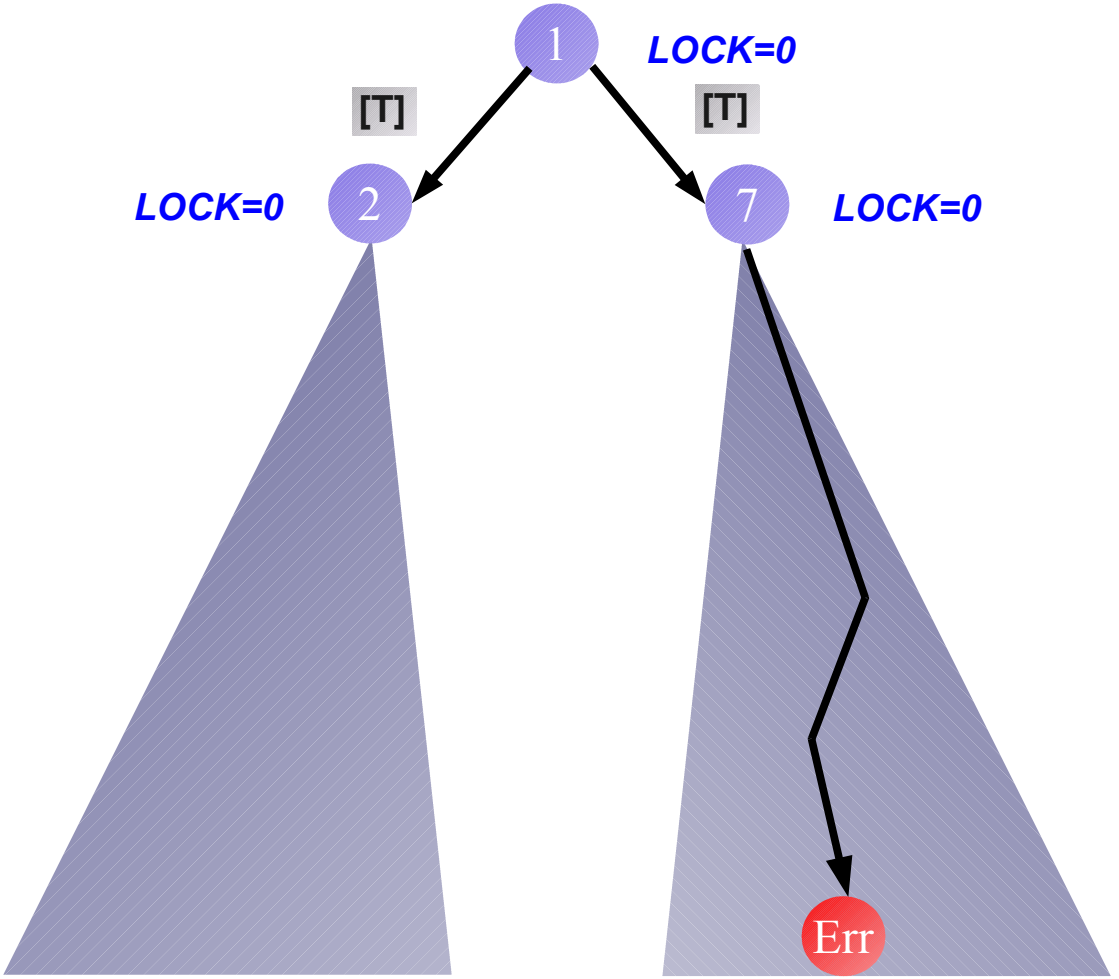
LOCK=0, LOCK=1

New predicate:

new = old,

Step 4: Search the Right Branch





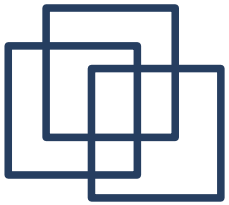
Set of predicates:

LOCK=0, LOCK=1

New predicate: (from trace)

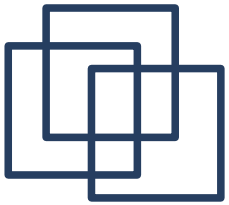
got_lock = 0

The program is correct...



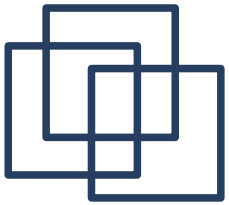
Conclusion

- Used in several tools:
  - Blast (Berkeley/UCLA)
  - Slam (Microsoft)
- Method has been proven to work on real code (device drivers, sendmail, ...).
- A lot of active research in this topic
- The key technology is to look for nice new predicates
- Infinite states systems are still badly handled
- Still need some tuning



Further Work (in our team)

- Check for more properties
  (liveness, shape-analysis, ...)
- Combine with other methods
  (theorem proving, satisfiable modulo theories, acceleration techniques)
- Program new tools
- Challenge real softwares or projects
- Apply the method for industry
  (CNES, AEDS, Airbus, CEA, ...)
- ... plenty of other tracks to follow ...



Questions ?