Scalable Defect Detection

Manuvir Das, Zhe Yang, Daniel Wang
Center for Software Excellence
Microsoft Corporation

What this series covers

- Various techniques for static analysis of large, real imperative programs
- Lessons from our experience building static analyses in the "real world"
  - "real world" == Microsoft product teams
- A *pragmatic* methodology for mixing specifications with static analysis

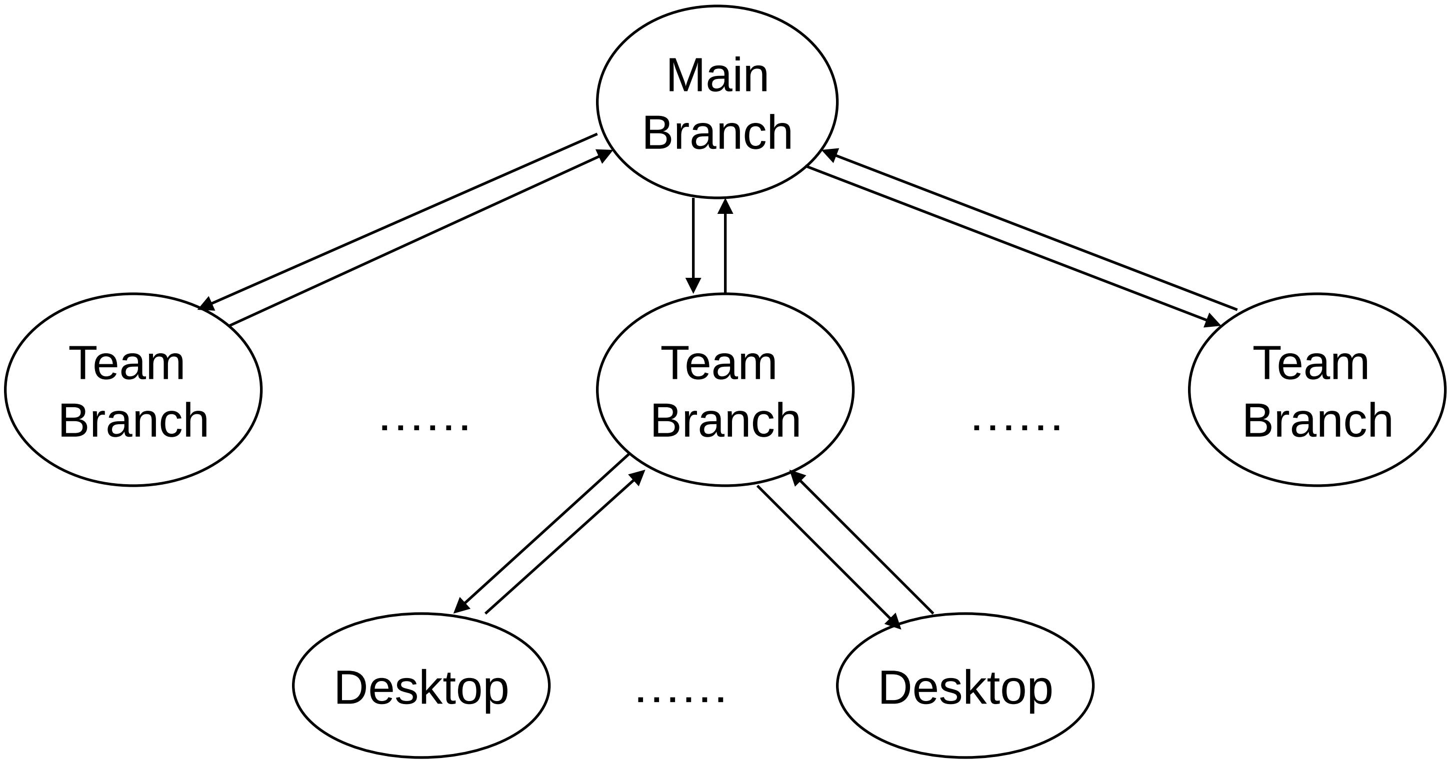
Who we are

- Microsoft Corporation
  - Center for Software Excellence
    - Program Analysis Group
      - 10 full time people including 7 PhDs
- We are program analysis researchers
  - But we measure our success by *impact*
  - CSE impact on Windows Vista
    - Developers *fixed* ~100K bugs that we found
    - Developers added ~500K specifications we designed
    - We answered thousands of developer emails

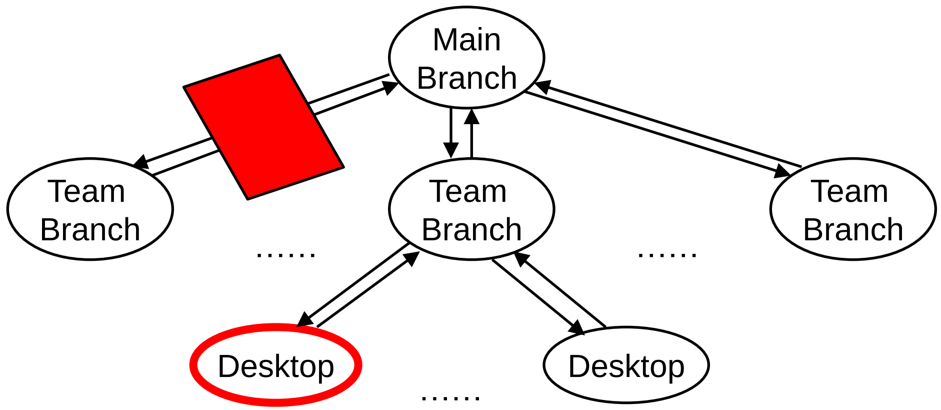
The real world

- Code on a massive scale
  - 10s of millions of lines of code
  - Many configurations & code branches
- Developers on a massive scale
  - Small mistakes in tools are magnified
  - Small developer overheads are magnified
- Defects on a massive scale
  - Bug databases and established processes rule
  - Defect classes repeat, both across code bases and across defect properties

Code in the real world



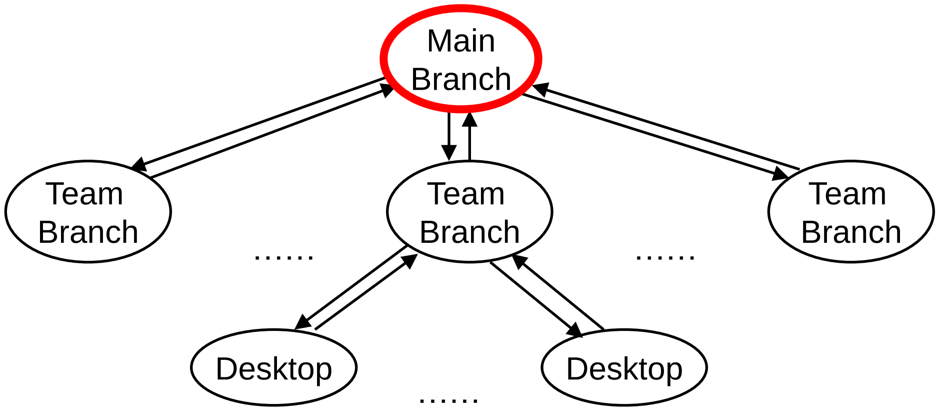
Process in the real world – 1



- An opportunity for lightweight tools
  - "always on" on every developer desktop
  - issues tracked within the program artifacts
  - enforcement by rejection at "quality gate"

    Speed, suppression, determinism

Process in the real world – 2



- An opportunity for heavyweight tools
  - run routinely after integration in main branch
  - issues tracked through a central bug database
  - enforcement by developer "bug cap"

Scale, uniqueness, defect management

Implications for analysis

- Scale, scale, scale
  - Should be run routinely on massive scale
- High accuracy
  - Ratio of bugs “worth fixing” should be high
- High clarity
  - Defect reports must be understandable
- Low startup cost
  - Developer effort to get results must be low
- High return on investment
  - More developer effort should reveal more bugs
- High agility
  - New defect detection tools should be easy to produce

Our solutions over time

- Gen 1: Manual Review
  - Too many code paths to think about
- Gen 2: Massive Testing
  - Inefficient detection of simple errors
- Gen 3: Global Program Analysis
  - Delayed results
- Gen 4: Local Program Analysis
  - Lack of calling context limits accuracy
- Gen 5: Formal interface specifications

Contrast this with …

- Build it into the language
  - e.g. memory management in Java/C#
- If not, then fix it with a type system
  - e.g. memory safety in Cyclone
- If not, then add formal specifications
  - e.g. memory defect detection in ESC/Java
- If not, then find bugs with static analysis
  - e.g. memory defect detection in PREfix
- If not, then find bugs with dynamic analysis

Our approach to scale

- Scalable whole program analysis
  - Combine lightweight analysis everywhere with heavyweight analysis in just the right places
- Accurate modular analysis
  - Assume availability of function-level pre-conditions and post-conditions
  - Powerful analysis + defect bucketing
- Programmer supplied specifications
  - Designed to be developer friendly
  - Automatically inferred via global analysis

… explained in 3 lectures

- Scalable whole program analysis
  - ESP
  - Manuvir Das

- Accurate modular analysis
  - espX, μSPACE
  - Zhe Yang

- Programmer supplied specifications
  - SAL, SALinfer
  - Daniel Wang

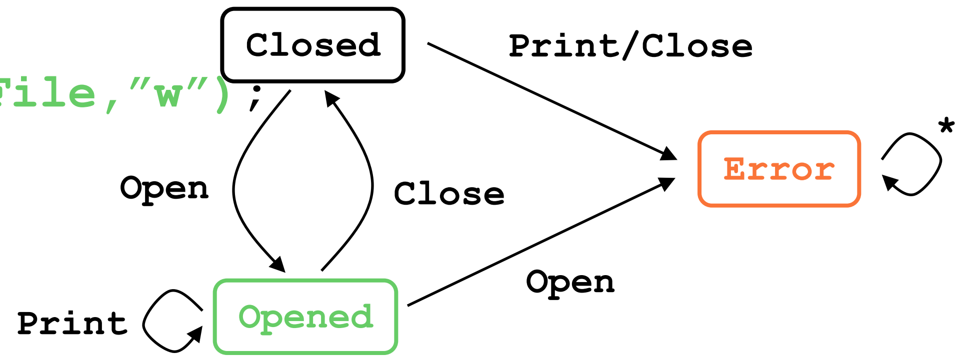
Scalable
Whole Program Analysis

Safety properties

- Dynamic checking
  - Instrument the program with a monitor
  - Fail if the monitor enters a bad state
- What is a safety property?
  - Anything that can be monitored
- Static checking
  - Simulate all possible executions of the instrumented program

Example

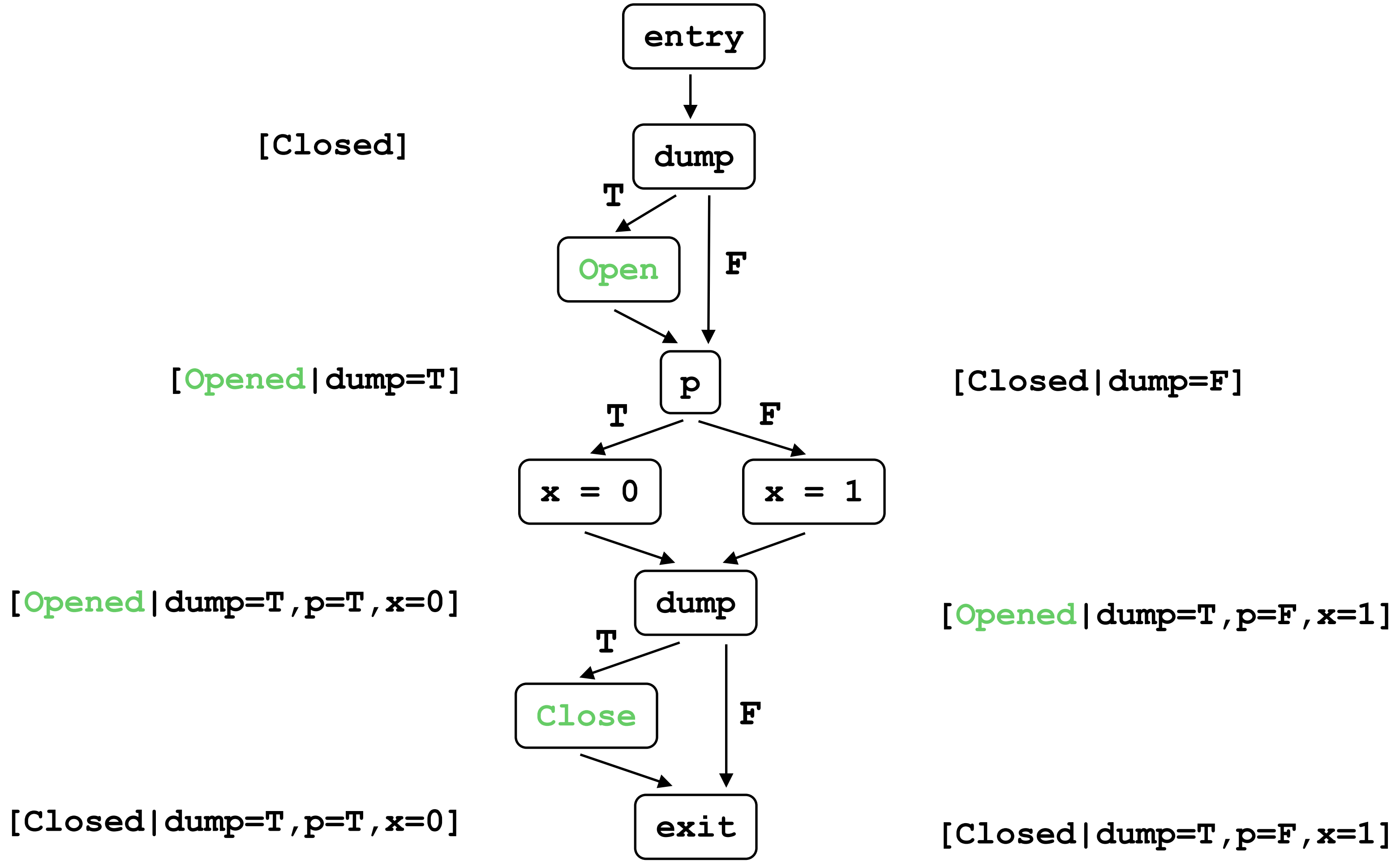
```
void main ()
{
  if (dump)
    fil = fopen(dumpFile,"w");

  if (p)
    x = 0;
  else
    x = 1;

  if (dump)
    fclose(fil);
}
```



Symbolic evaluation

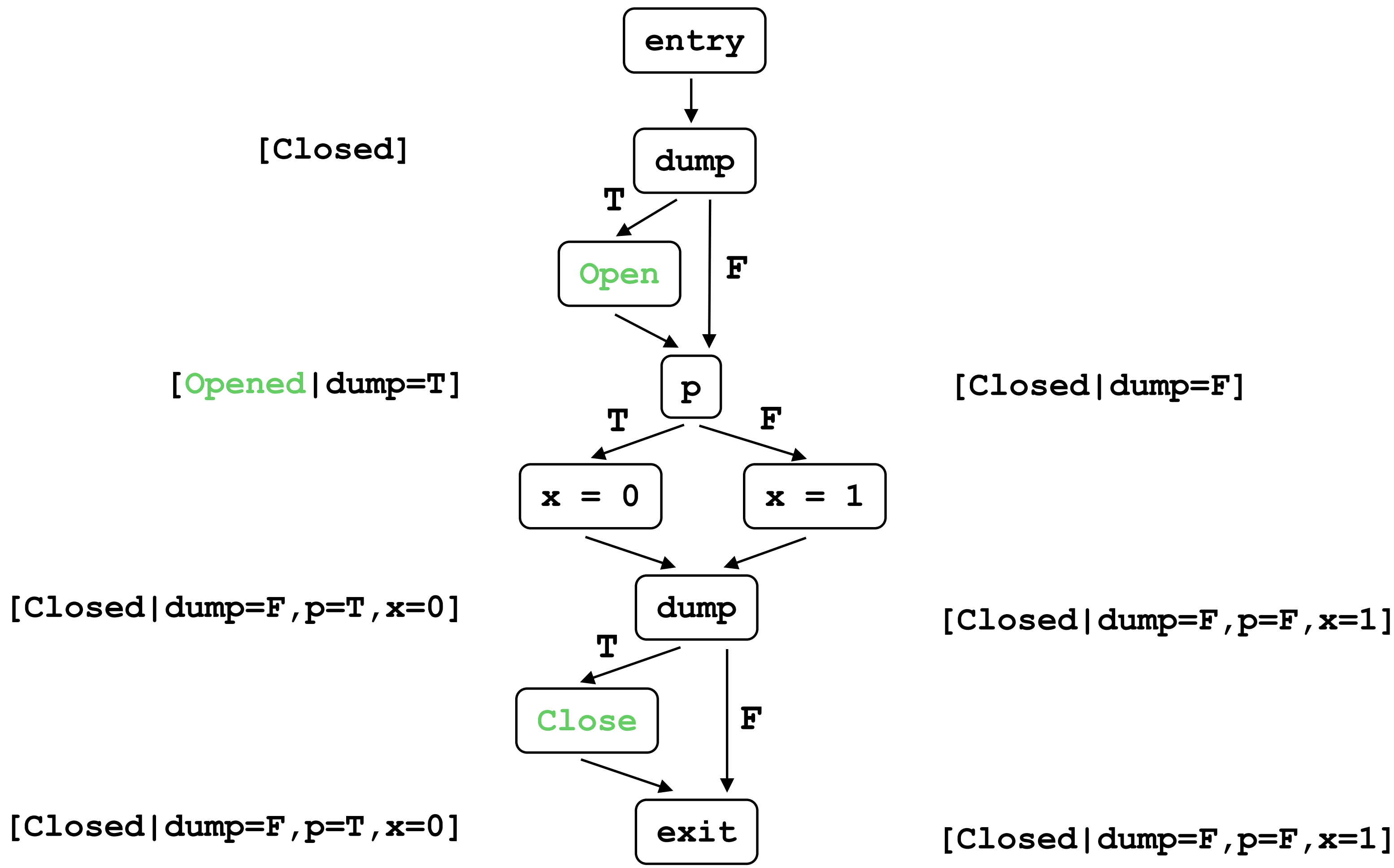
- Execute multiple paths in the program
  - Use symbolic values for variables
  - Execution state = Symbolic state + Monitor state
- Assignments & function calls:
  - Update execution state
- Branch points:
  - Does execution state imply branch direction?
  - Yes: process appropriate branch
  - No: split & update state, process branches
- Merge points:
  - Collapse *identical* states

Example



entry
[Closed]
dump
T
F
Open
[Opened|dump=T]
p
[Closed|dump=F]
T
F
x = 0
x = 1
[Opened|dump=T,p=T,x=0]
dump
[Opened|dump=T,p=F,x=1]
T
F
Close
[Closed|dump=T,p=T,x=0]
exit
[Closed|dump=T,p=F,x=1]

Example


entry
[Closed]
dump
T
F
Open
[Opened|dump=T]
p
[Closed|dump=F]
T
F
x = 0
x = 1
[Closed|dump=F,p=T,x=0]
dump
[Closed|dump=F,p=F,x=1]
T
F
Close
[Closed|dump=F,p=T,x=0]
exit
[Closed|dump=F,p=F,x=1]

Assessment

[+] can make this arbitrarily precise

[+] can show debugging traces

[-] may not scale (exponential paths)

[-] may not terminate (loops)

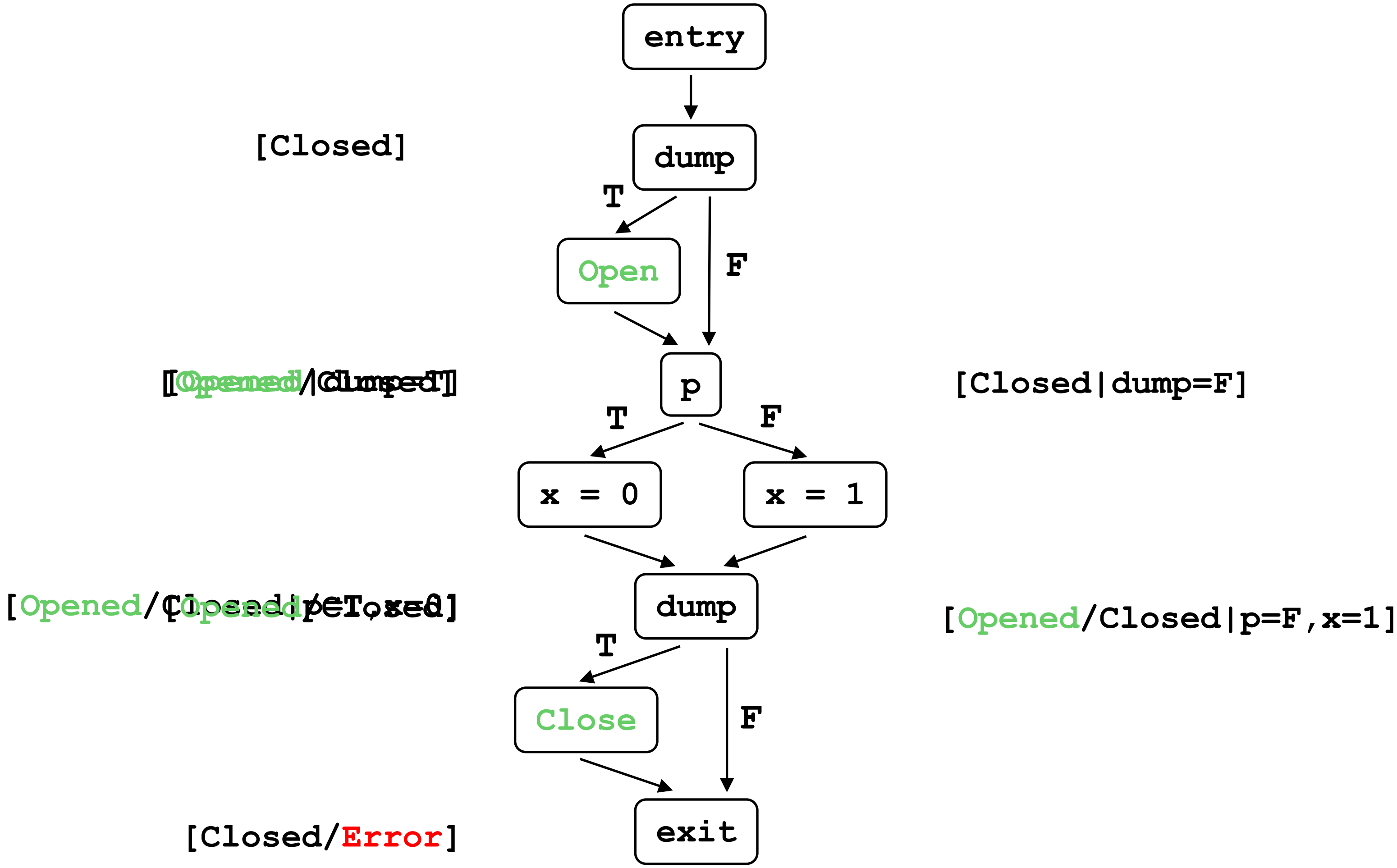
PREfix (SP&E 2000)

- explore a subset of all paths

Dataflow analysis

- Merge points:
  - Collapse all states
  - One execution state per program point

Example



entry
[Closed]
dump
T
Open
F
p
[Closed|dump=F]
T
F
x = 0
x = 1
dump
[Opened/Closed|p=F,x=1]
T
Close
F
exit
[Closed/Error]

Assessment

[-] precision is limited

[-] cannot show debugging traces

[+] scales well

[+] terminates

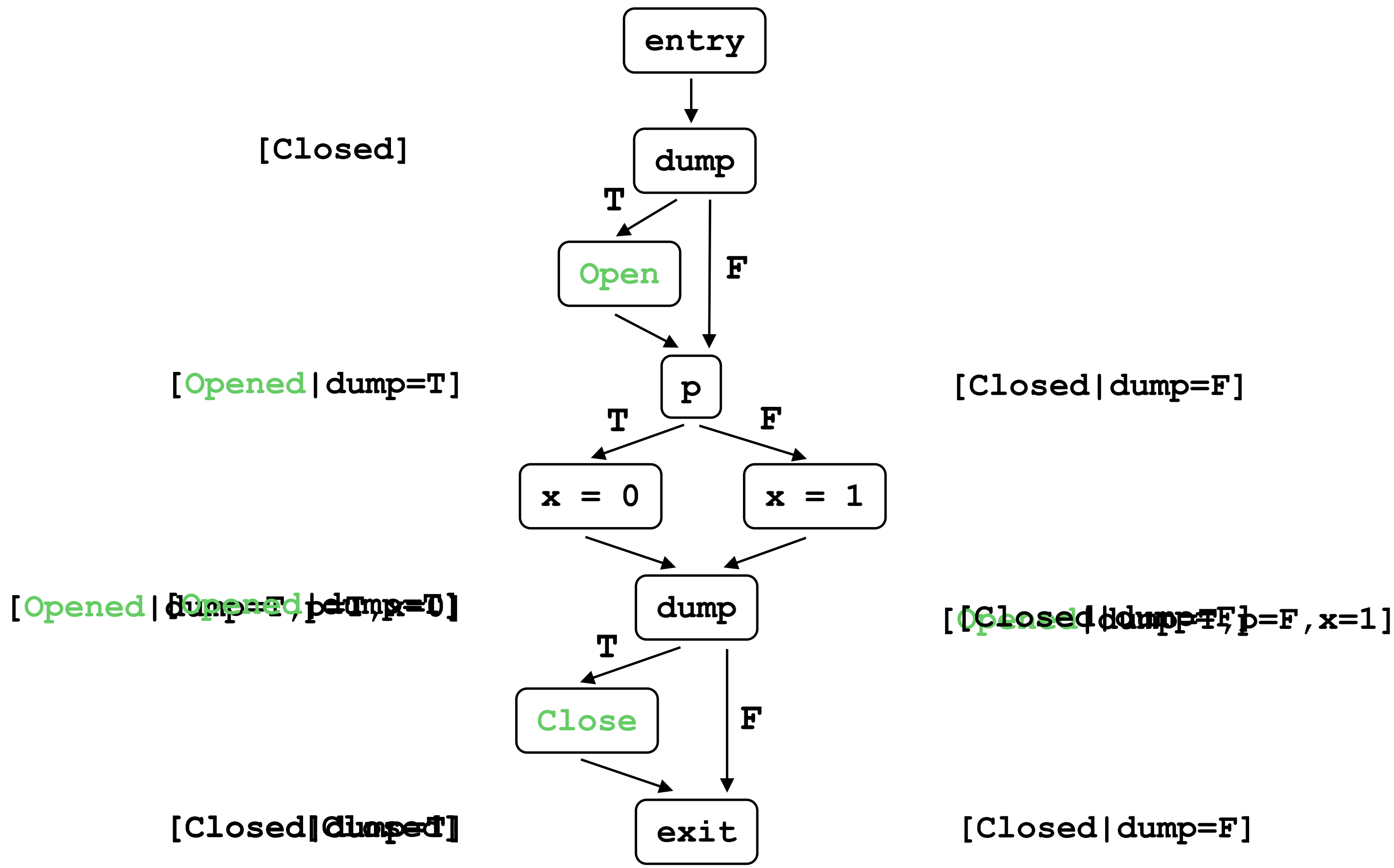
CQual (PLDI 2002)

- apply to type-based properties

Property simulation

- Merge points:
  - Do states agree on monitor state?
  - Yes: merge states
  - No: process states separately
- ESP
  - PLDI 2002, ISSTA 2004

Example


entry
dump
T
F
Open
p
x = 0
x = 1
dump
Close
exit
[Closed]
[Opened|dump=T]
[Closed|dump=F]
[Closed|dump=F]

Assessment

- [=] is usually precise
- [=] can usually show debugging traces

- [=] usually scales well
- [=] usually terminates

ESP

- a pragmatic compromise

Multiple state machines

```
void main ()
{
  if (dump1)
    fil1 = fopen(dumpFile1,"w");

  if (dump2)
    fil2 = fopen(dumpFile2,"w");

  if (dump1)
    fclose(fil1);

  if (dump2)
    fclose(fil2);
}
```

Code Pattern	Monitor Transition
e = fopen(_)	Open(e)
fclose(e)	Close(e)

Multiple state machines

```
void main ()
{
  if (dump1)
    Open(fil1);

  if (dump2)
    Open(fil2);

  if (dump1)
    Close(fil1);

  if (dump2)
    Close(fil2);
}
```

Code Pattern	Monitor Transition
`e = fopen(_)`	`Open(e)`
`fclose(e)`	`Close(e)`

Bit vector analysis

```
void main ()
{
  if (dump1)
    Open;

  if (dump2)
    ID;

  if (dump1)
    Close;

  if (dump2)
    ID;
}
```

```
void main ()
{
  if (dump1)
    ID;

  if (dump2)
    Open;

  if (dump1)
    ID;

  if (dump2)
    Close;
}
```

Bit vector analysis

- Source to sink safety properties
  - Sources: Object creation points or function/component entry points
  - Sinks: Transitions to error state
- Analyze every source independently
  - Requires (exponentially) less memory
  - Spans smaller segments of code
  - Parallelizes easily

Memory aliasing

```
void main ()
{
  if (dump)
    fil = fopen(dumpFile,"w");

  pfil = &fil;

  if (dump)
    fclose(*pfil);
}
```

- Does Event(`exp`) invoke a transition on the monitor for location `l`?

Value alias analysis

- Precise alias analysis is expensive
- Solution: value alias sets (ISSTA 04)
  - For a given execution state, which syntactic expressions refer to location `l`?
    - Must and May sets for accuracy
    - Transfer functions to update these
- Observation: We can make value alias analysis path-sensitive by tracking value alias sets as part of monitor state

Selective merging

- Property simulation is really an instance of a more general analysis approach
- Selective merging
  - Define a projection on symbolic states
  - Define equality on projections
  - Ensure that domain of projections is finite
- Merge points:
  - Do states agree on projection?
  - Yes: merge states
  - No: process states separately
- Examples
  - Value flow analysis, call graph analysis

ESP at Microsoft

Windows Vista		
Issue	*Fixed*	*Noise*
Security – RELOJ	386	4%
Security – Impersonation Token	135	10%
Security – OpenView	54	2%
Leaks – RegCloseHandle	63	0%
In Progress		
Issue	*Found*	
Localization – Constant strings	1214	
Security – ClientID	282	
…		

Summary

- Scalable whole program analysis
  - Combine lightweight analysis everywhere with heavyweight analysis in just the right places
- Accurate modular analysis
  - Assume availability of function-level pre-conditions and post-conditions
  - Powerful analysis + defect bucketing
- Programmer supplied specifications
  - Designed to be developer friendly
  - Automatically inferred via global analysis

www.microsoft.com/cse
www.microsoft.com/cse/pa
research.microsoft.com/manuvir