

Software Excellence via Program Analysis at Microsoft

Used by permission for 17-654/17-754:
Analysis of Software Artifacts
Jonathan Aldrich, Instructor

Manuvir Das

Center for Software Excellence
Microsoft Corporation

Static Analysis at Microsoft

- Program Analysis group in June 2005
 - Filed 7000+ bugs
 - Automatically added 10,000+ specifications
 - Answered hundreds of emails
(one future version of one product)
- We are program analysis researchers
 - but we live and breathe deployment & adoption
 - and we feel the pain of the customer

Context

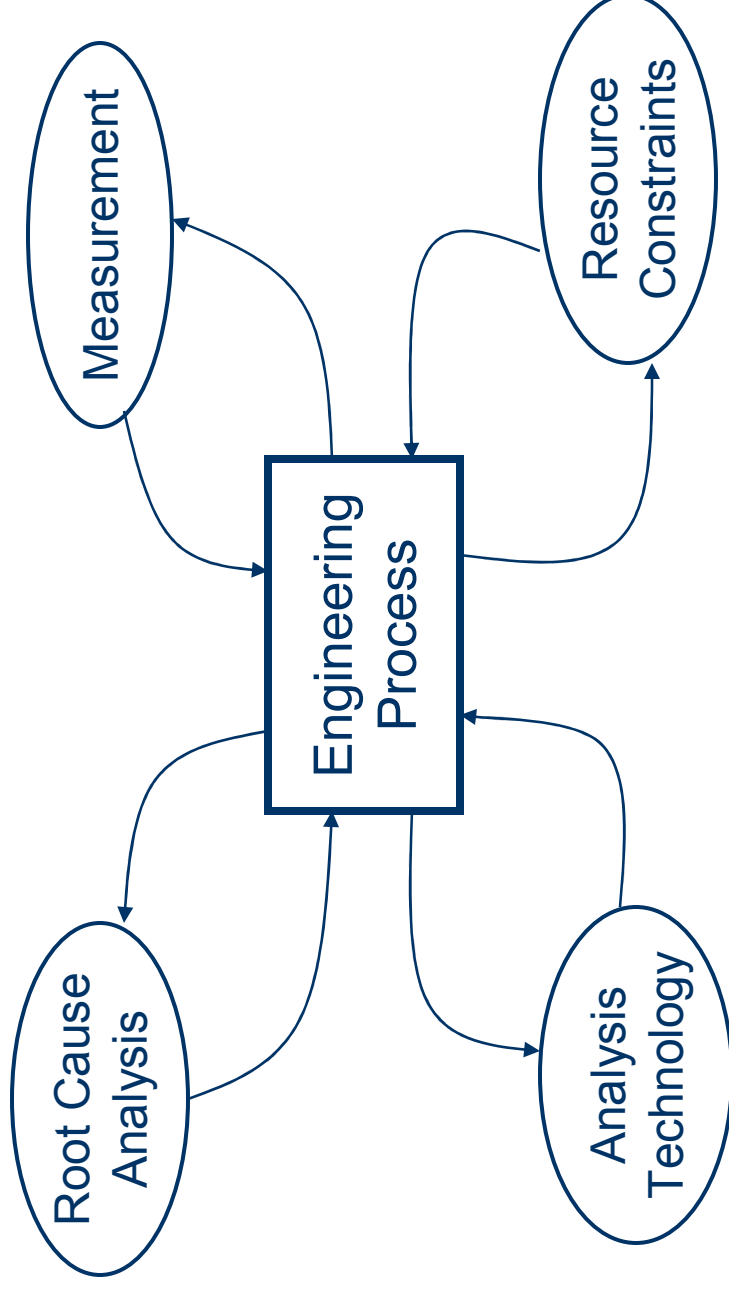
- The Nail (Windows)
 - Manual processes do not scale to “real” software
- The Hammer (Program Analysis)
 - Automated methods for “searching” programs
- The Carpenter (CSE)
 - A systematic, heavily automated, approach to improving the “quality” of software

Roadmap

- (part of) The engineering process today
- (some of) The tools that enable the process
- (a few) Program analyses behind the tools
- (too many) Lessons learned along the way
- (too few) Suggestions for future research

Engineering process

Methodology



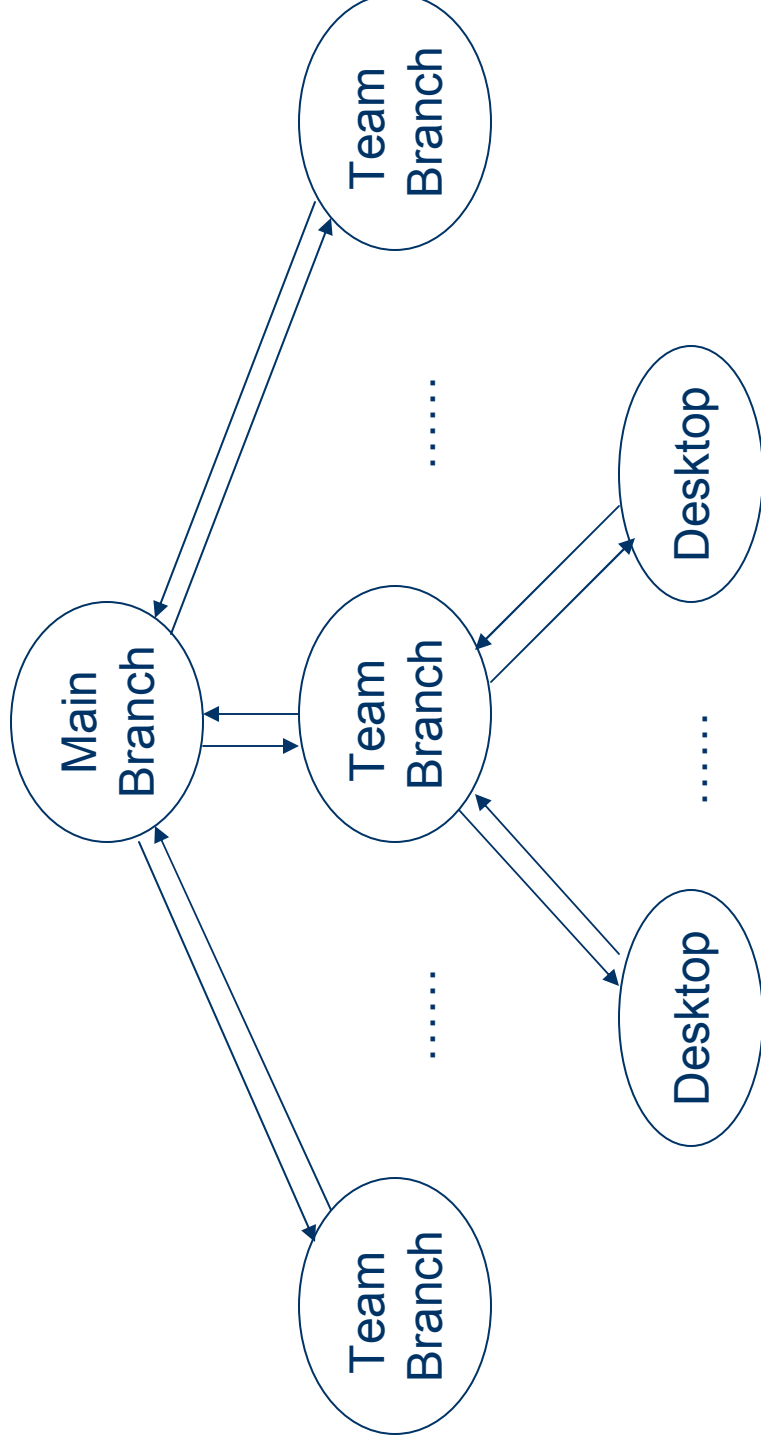
Root cause analysis

- Understand important failures in a deep way
 - Every MSRC bulletin
 - Beta release feedback
 - Watson crash reports
 - Self host
 - Bug databases
- Design and adjust the engineering process to ensure that these failures are prevented

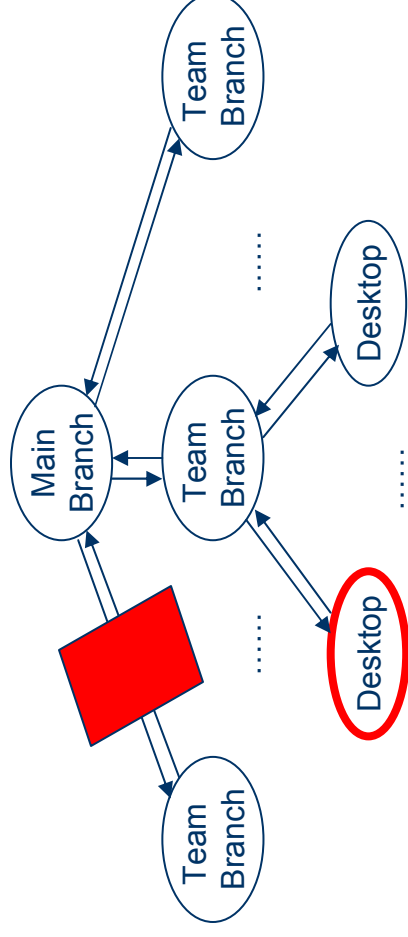
Measurement

- Measure everything about the process
 - Code quality
 - Code velocity
 - Tools effectiveness
 - Developer productivity
- Tweak the process accordingly

Process – Build Architecture

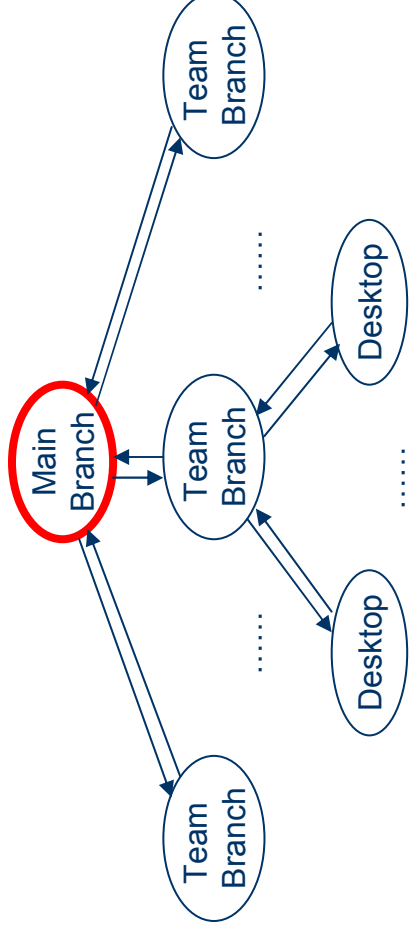


Process – Quality Gates



- Lightweight tools
 - run on developer desktop & team level branches
 - issues tracked within the program artifacts
- Enforced by rejection at gate

Process – Automated Bug Filing



- Heavyweight tools
 - run on main branch
 - issues tracked through a central bug database
- Enforced by bug cap

Tools

QG – Code Coverage via Testing

- Reject code that is not adequately tested
 - Maintain a minimum bar for code coverage
- Code coverage tool – Magellan
- Based on binary analysis - Vulcan

Magellan

- **BBCover**
 - low overhead instrumentation & collection
 - down to basic block level
- **Sleuth**
 - coverage visualization, reporting & analysis
- **Blender**
 - coverage migration
- **Scout**
 - test prioritization

QG – Component Integrity

- Reject code that breaks the componentized architecture of the product
 - Control all dependencies across components
- Dependency analysis tool – MaX
- Based on binary analysis - Vulcan

MaX

- Constructs a graph of dependencies between binaries (DLLs) in the system
 - Obvious : call graph
 - Subtle : registry, RPC, ...
- Compare policy graph and actual graph
- Some discrepancies are treated as errors

Vulcan

- Input – binary code
- Output – program abstractions
- Adapts to level of debug information
- Makes code instrumentation easy
 - think ATOM
- Makes code modification easy
 - link time, post link time, install time, run time

QG – Formal Specifications

- Reject code with poorly designed and/or insufficiently specified interfaces
- Lightweight specification language – SAL
 - initial focus on memory usage
- All functions must be SAL annotated
- Fully supported in Visual Studio (see MSDN)

SAL

- A language of contracts between functions
- preconditions
 - Statements that hold at entry to the callee
 - What does a callee expect from its callers?
- postconditions
 - Statements that hold at exit from the callee
 - What does a callee promise its callers?
- Usage example:
 - a_0 RT func ($a_1 \dots a_n$ T par)
- Buffer sizes, null pointers, memory usage, ...

SAL Example

- wcsncpy
 - precondition: destination must have enough allocated space

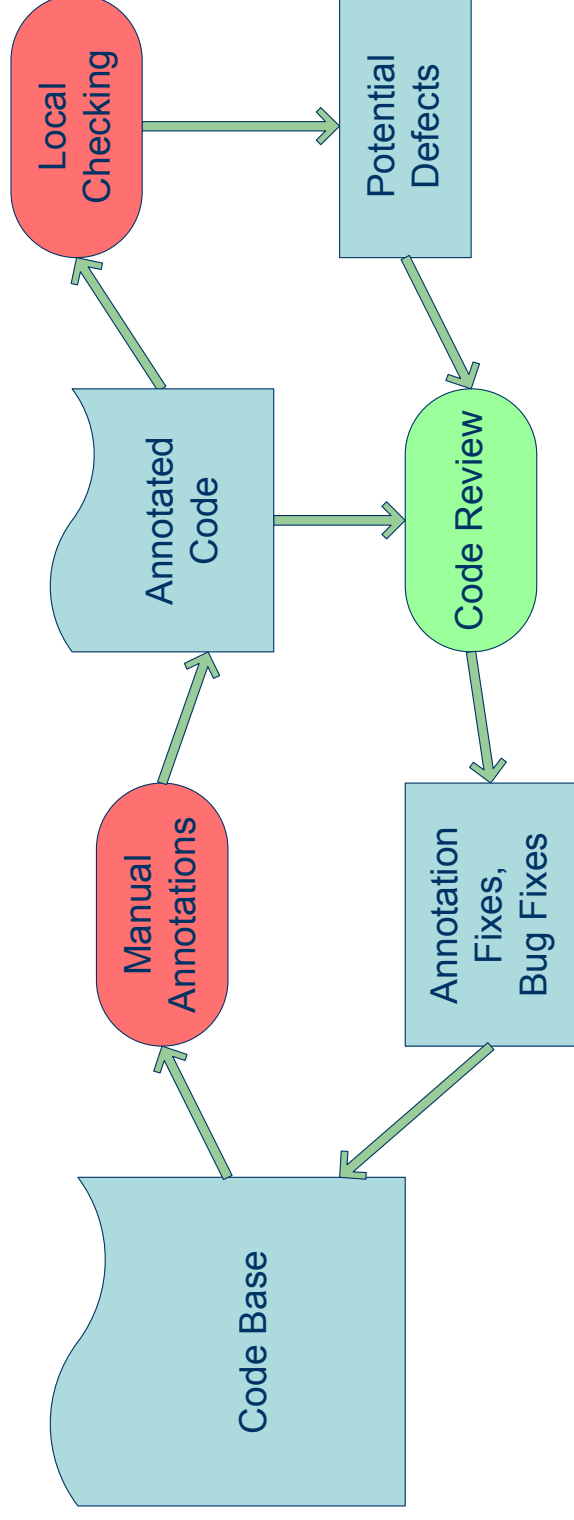
```
wchar_t wcsncpy (
    wchar_t *dest, wchar_t *src, size_t num );
```

```
wchar_t wcsncpy (
    __pre_writableTo(elementCount(num)) wchar_t *dest,
    wchar_t *src, size_t num );
```

SAL Principle

- Control the power of the specifications:
 - Impractical solution: Rewrite code in a different language that is amenable to automated analysis
 - Practical solution: Formalize invariants that are implicit in the code in intuitive notations
 - These invariants often already appear in comments

Defect Detection Process – 1



QG – Integer Overflow

- Reject code with potential security holes due to unchecked integer arithmetic
- Range specifications + range checker – IO
- Local (intra-procedural) analysis
- Runs on developer desktop as part of regular compilation process

IO

- Enforces correct arithmetic for allocations

```
size1 = ...  
size2 = ...  
data = MyAlloc(size1+size2);  
for (i = 0; i < size1; i++)  
    data[i] = ...
```

- Construct an expression tree for every interesting expression in the code
- Ensure that every node in the tree is checked

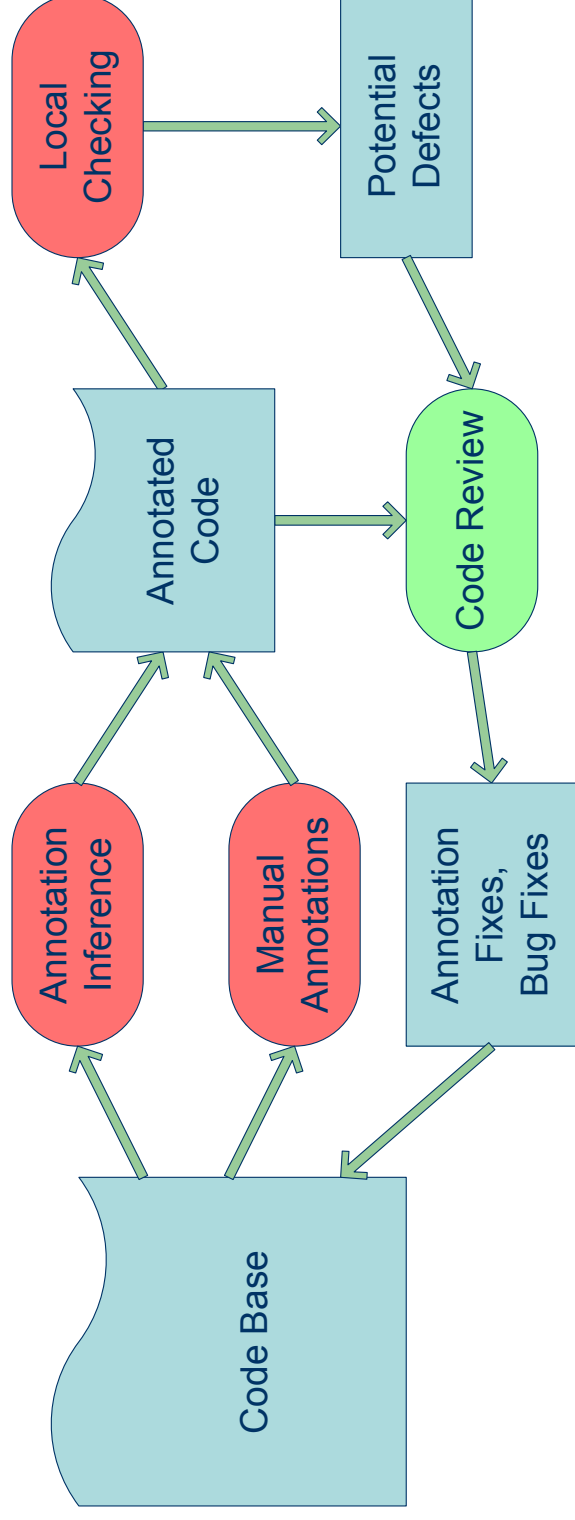
QG – Buffer Overruns

- Reject code with potential security holes due to out of bounds buffer accesses
- Buffer size specifications + buffer overrun checker – espX
- Local (intra-procedural) analysis
- Runs on developer desktop as part of regular compilation process

Bootstrap the process

- Combine global and local analysis:
 - Weak global analysis to infer (potentially inaccurate) interface annotations - SALInfer
 - Strong local analysis to identify incorrect code and/or annotations - espX

Defect Detection Process - 2



SALinfer

```
void work()
{
    int elements[200];
    wrap(elements, 200);
}

void wrap(    int len)
            int *buf,

{
    int *buf2 = buf;
    int len2 = len;
    zero(buf2, len2);
}

void zero(    int len)
            int *buf,

{
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

SALinfer

```
void work()  
{  
  int elements[200];  
  wrap(elements, 200);  
}  
  
void wrap(  
    int *buf,  
    int len)  
{  
  int *buf2 = buf;  
  int len2 = len;  
  zero(buf2, len2);  
}  
  
void zero(  
    int *buf,  
    int len)  
{  
  int i;  
  for(i = 0; i <= len; i++)  
    buf[i] = 0;  
}
```

Track flow of values through the code

1. Finds stack buffer

SALinfer

```
void work()  
{  
    int elements[200];  
    wrap(elements, 200);  
}  
  
void wrap(pre elementCount(len) int *buf,  
         int len)  
{  
    int *buf2 = buf;  
    int len2 = len;  
    zero(buf2, len2);  
}  
  
void zero(  
         int len)  
{  
    int i;  
    for(i = 0; i <= len; i++)  
        buf[i] = 0;  
}
```

Track flow of values through the code

1. Finds stack buffer
2. Adds annotation

SALinfer

```
void work()  
{  
    int elements[200];  
    wrap(elements, 200);  
}  
  
void wrap(pre elementCount(len) int *buf,  
         int len)  
{  
    int *buf2 = buf;  
    int len2 = len;  
    zero(buf2, len2);  
}  
  
void zero(  
         int len)  
{  
    int i;  
    for(i = 0; i <= len; i++)  
        buf[i] = 0;  
}
```

Track flow of values through the code

1. Finds stack buffer
2. Adds annotation
3. Finds assignments

SALinfer

```
void work()  
{  
    int elements[200];  
    wrap(elements, 200);  
}  
  
void wrap(pre elementCount(len) int *buf,  
         int len)  
{  
    int *buf2 = buf;  
    int len2 = len;  
    zero(buf2, len2);  
}  
  
void zero(pre elementCount(len) int *buf,  
         int len)  
{  
    int i;  
    for(i = 0; i <= len; i++)  
        buf[i] = 0;  
}
```

Track flow of values through the code

1. Finds stack buffer
2. Adds annotation
3. Finds assignments
4. Adds annotation

espX

```
void work()
{
    int elements[200];
    wrap(elements, 200);
}

void wrap(pre elementCount(len) int *buf,
          int len)
{
    int *buf2 = buf;
    int len2 = len;
    zero(buf2, len2);
}

void zero(pre elementCount(len) int *buf,
          int len)
{
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

espX

```
void work()  
{  
  int elements[200];  
  wrap(elements, 200);  
}  
  
void wrap(pre elementCount(len) int *buf,  
         int len)  
{  
  int *buf2 = buf;  
  int len2 = len;  
  zero(buf2, len2);  
}  
  
void zero(pre elementCount(len) int *buf,  
         int len)  
{  
  int i;  
  for(i = 0; i <= len; i++)  
    buf[i] = 0;  
}
```

Building and solving constraints

1. Builds constraints
2. Verifies contract

espX

```
void work()  
{  
    int elements[200];  
    wrap(elements, 200);  
}  
  
void wrap(pre elementCount(len) int *buf,  
         int len)  
{  
    int *buf2 = buf;  
    int len2 = len;  
    zero(buf2, len2);  
}  
  
void zero(pre elementCount(len) int *buf,  
         int len)  
{  
    int i;  
    for(i = 0; i <= len; i++)  
        buf[i] = 0;  
}
```

Building and solving constraints

1. Builds constraints
2. Verifies contract
3. Builds constraints
len = length(buf); i ≤ len
4. Finds overrun
i < length(buf) ? NO!

QG – Code Correctness

- Reject code with potential crashes due to improper usage of memory
- Pointer usage specifications + memory usage checker – PREfast
- Managed code – PResharp
- Local (intra-procedural) analysis
- Runs on developer desktop as part of regular compilation process

ABF – Code Correctness

- Tease out hard to find inter-component bugs that lead to crashes
 - null dereference, un-initialized memory, leaks, ...
 - difficult to find accurately on the desktop
- Inter-procedural symbolic evaluation - PREFIX

PREFIX

- Bottom-up process on the call graph
- Symbolic evaluation of a fixed number of distinct paths through each function
 - use symbolic state to remove infeasible paths
 - report defects
 - build function models for use by callers
- Solved all the difficult engineering problems for the inter-procedural tools to follow

ABF – Security

- For every new security issue, map it to a coding defect and root out all other instances
 - Each coding defect is a different pattern, but most can be viewed as finite state properties
- Heavyweight, thorough, property-based inter-procedural analysis - ESP

Property-based analysis

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

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

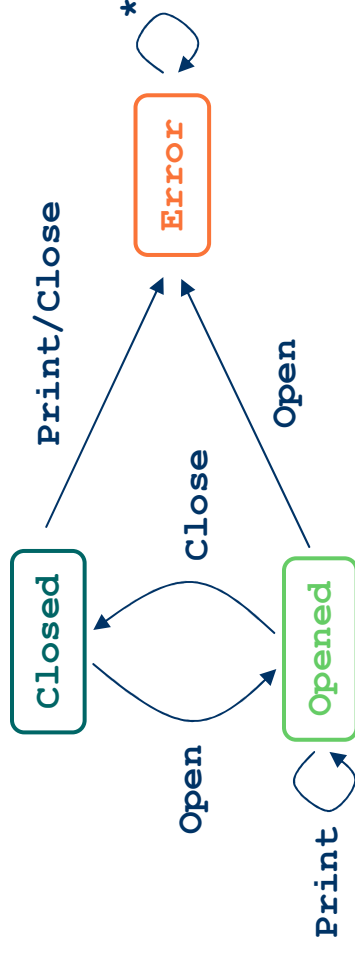
    if (dump)
        fclose (fil);
}
```


Property-based analysis

```
void main ()  
{  
  if (dump)  
    Open;  
  
  if (p)  
    x = 0;  
  else  
    x = 1;  
  
  if (dump)  
    Close;  
}
```

Property-based analysis

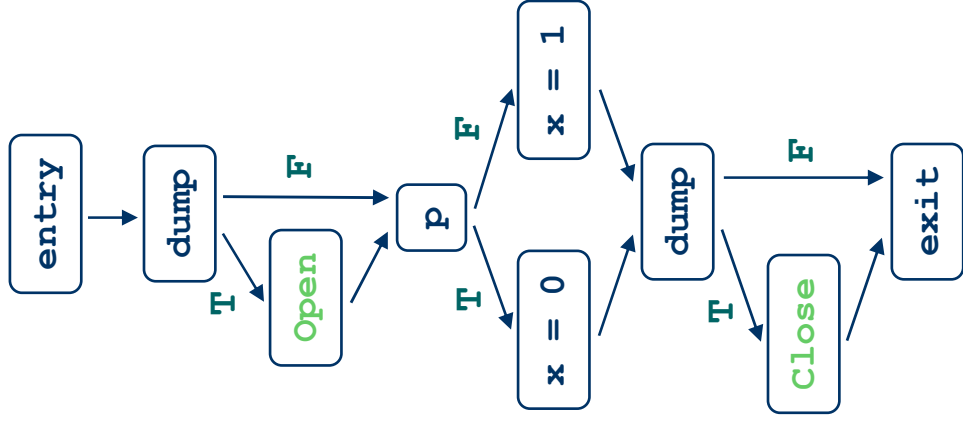
```
void main ()  
{  
  if (dump)  
    Open;  
  
  if (p)  
    x = 0;  
  else  
    x = 1;  
  
  if (dump)  
    Close;  
}
```



ESP

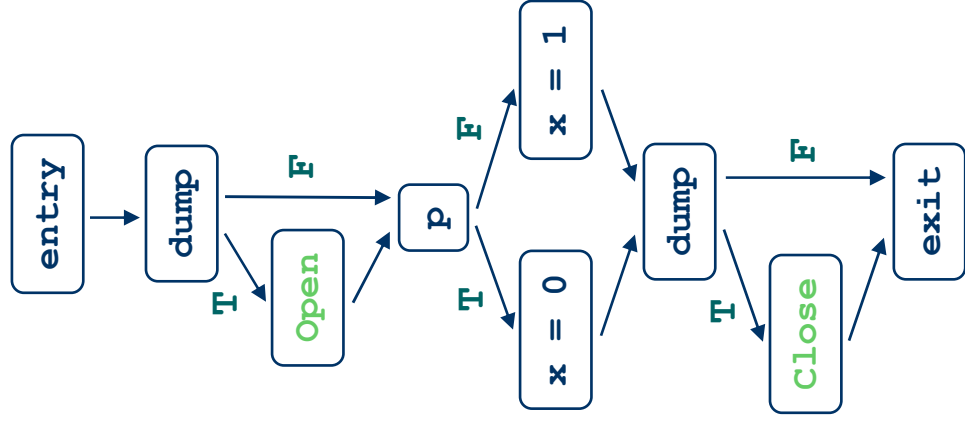
- Symbolically evaluate the program
 - track FSA state and execution state
- At control flow branch points:
 - Execution state implies branch direction?
 - Yes: process appropriate branch
 - No: split state and process both branches
- At control flow merge points:
 - States agree on property FSA state?
 - **Yes: merge states**
 - **No: process states separately**

Example

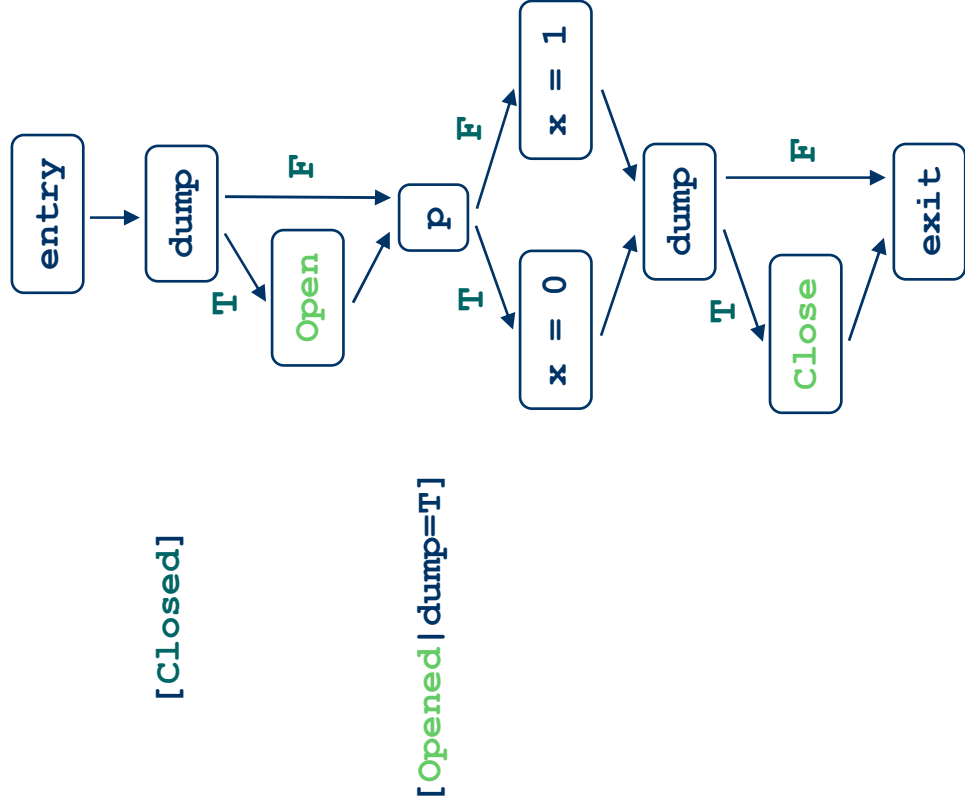


Example

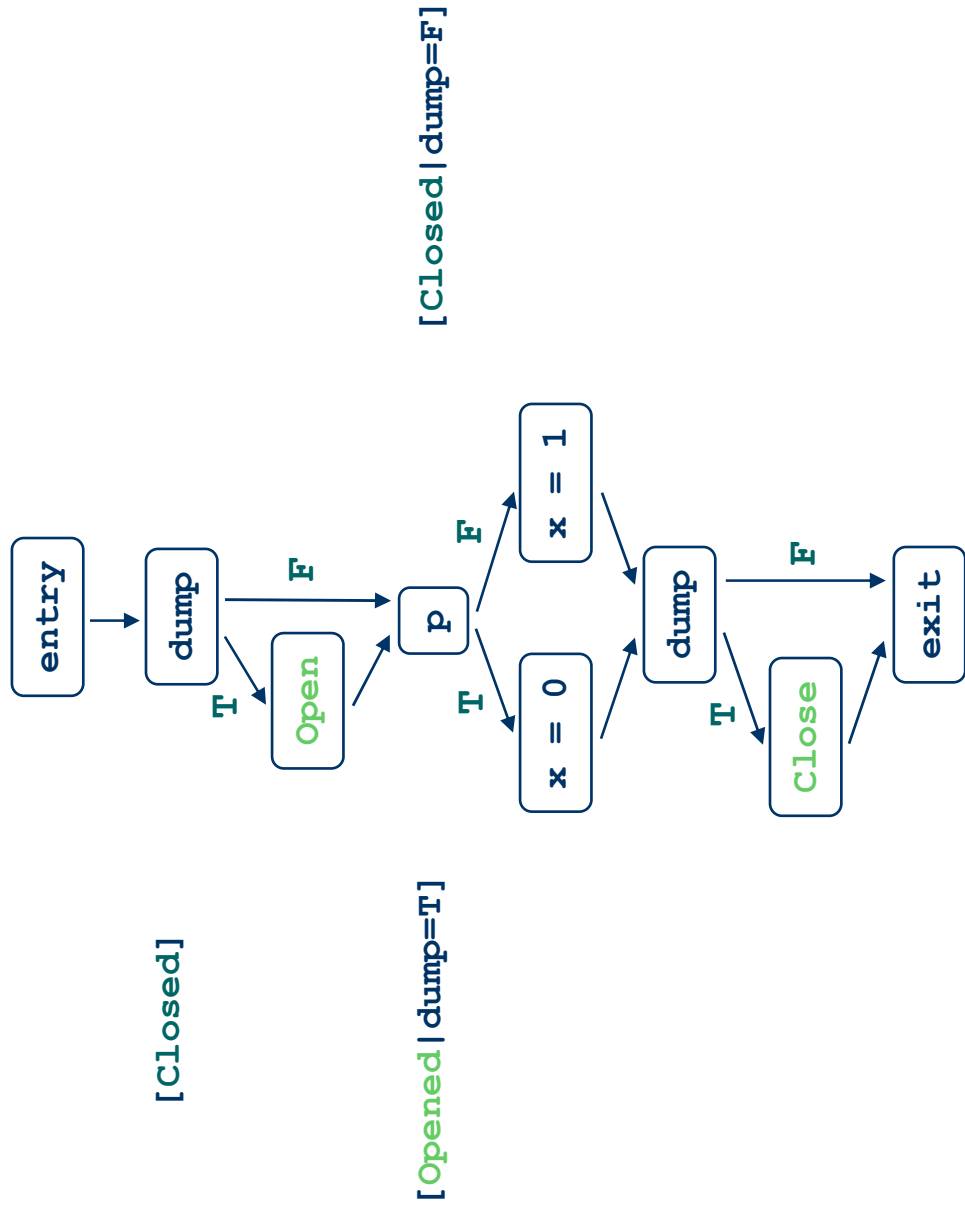
[Closed]



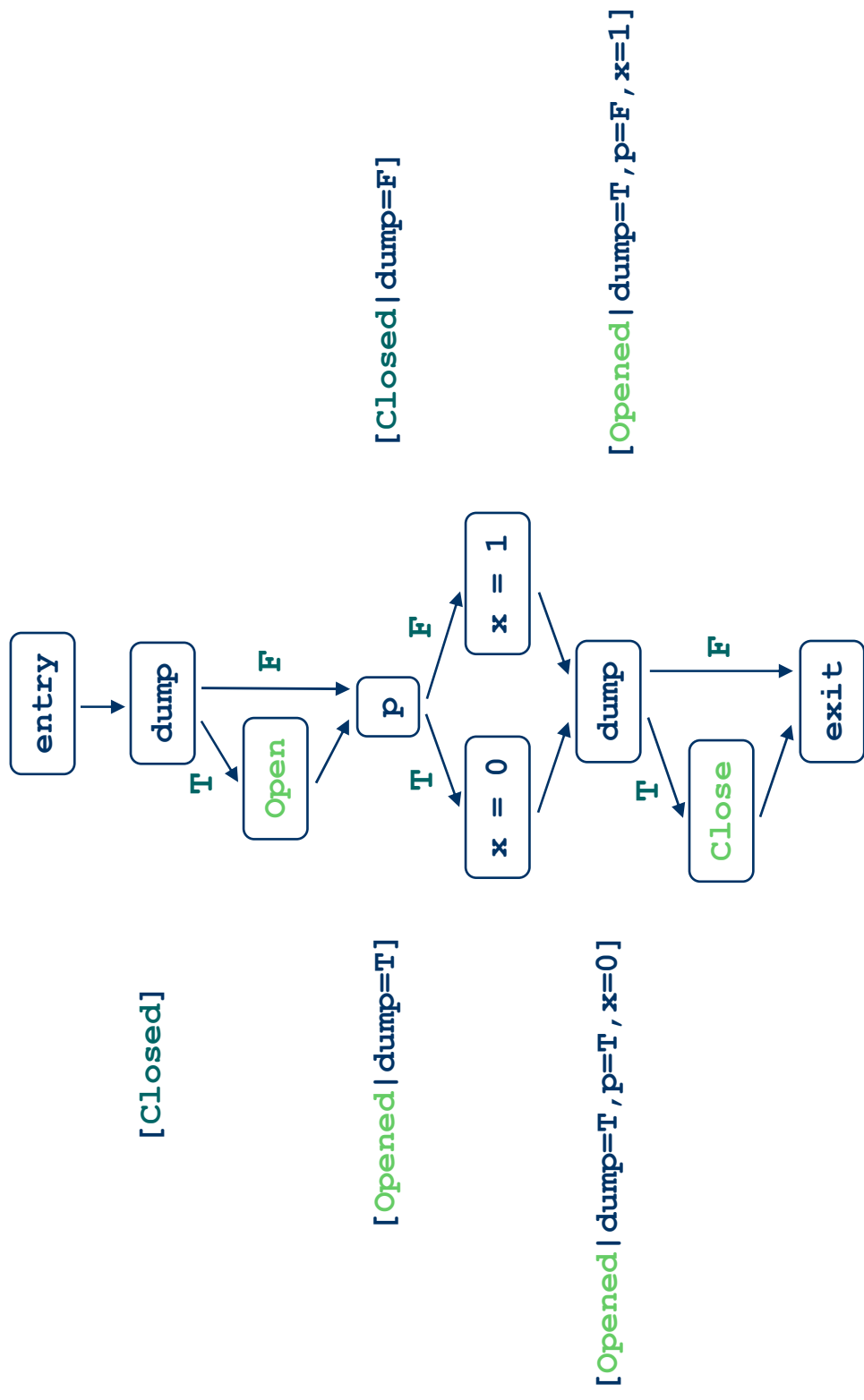
Example



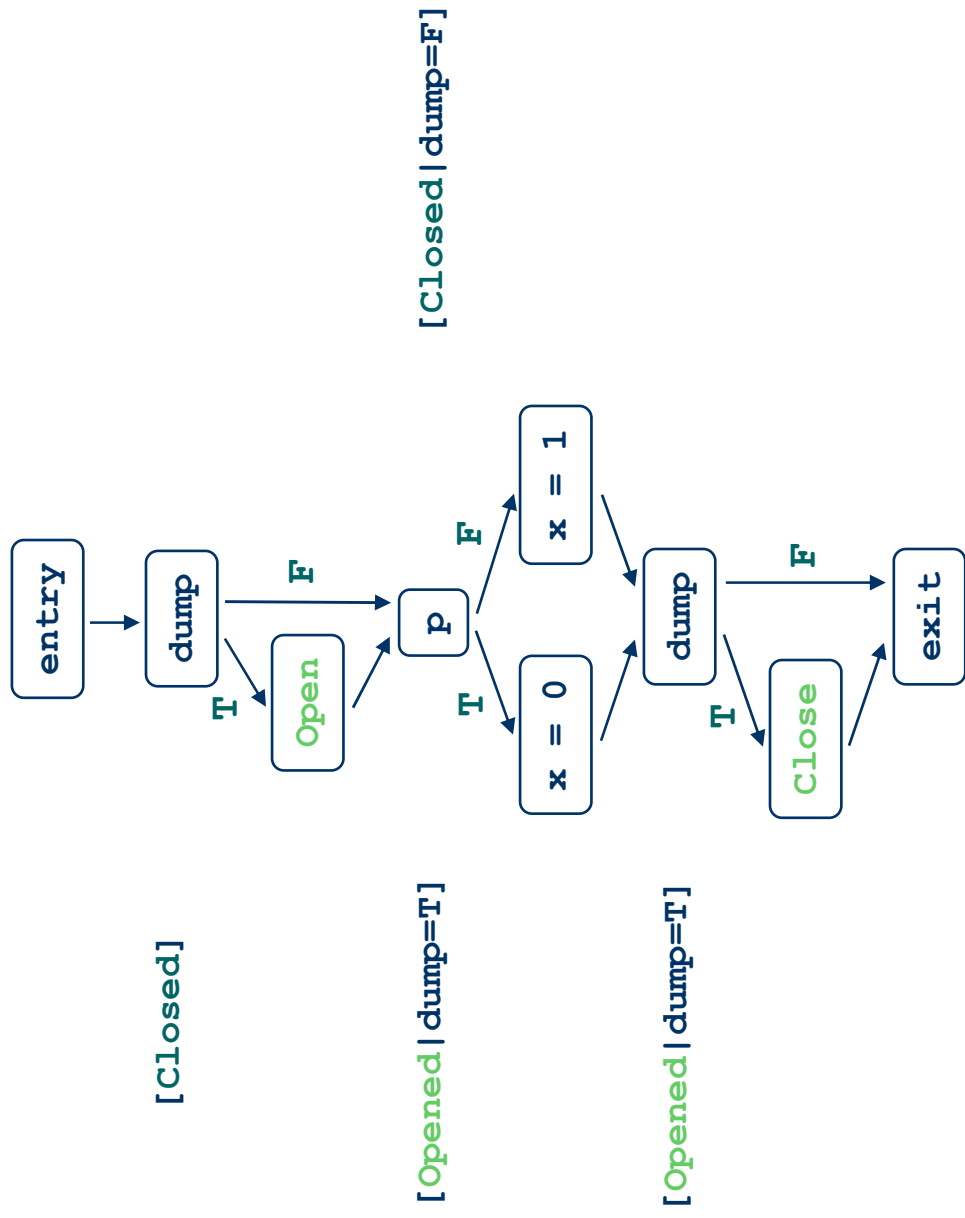
Example



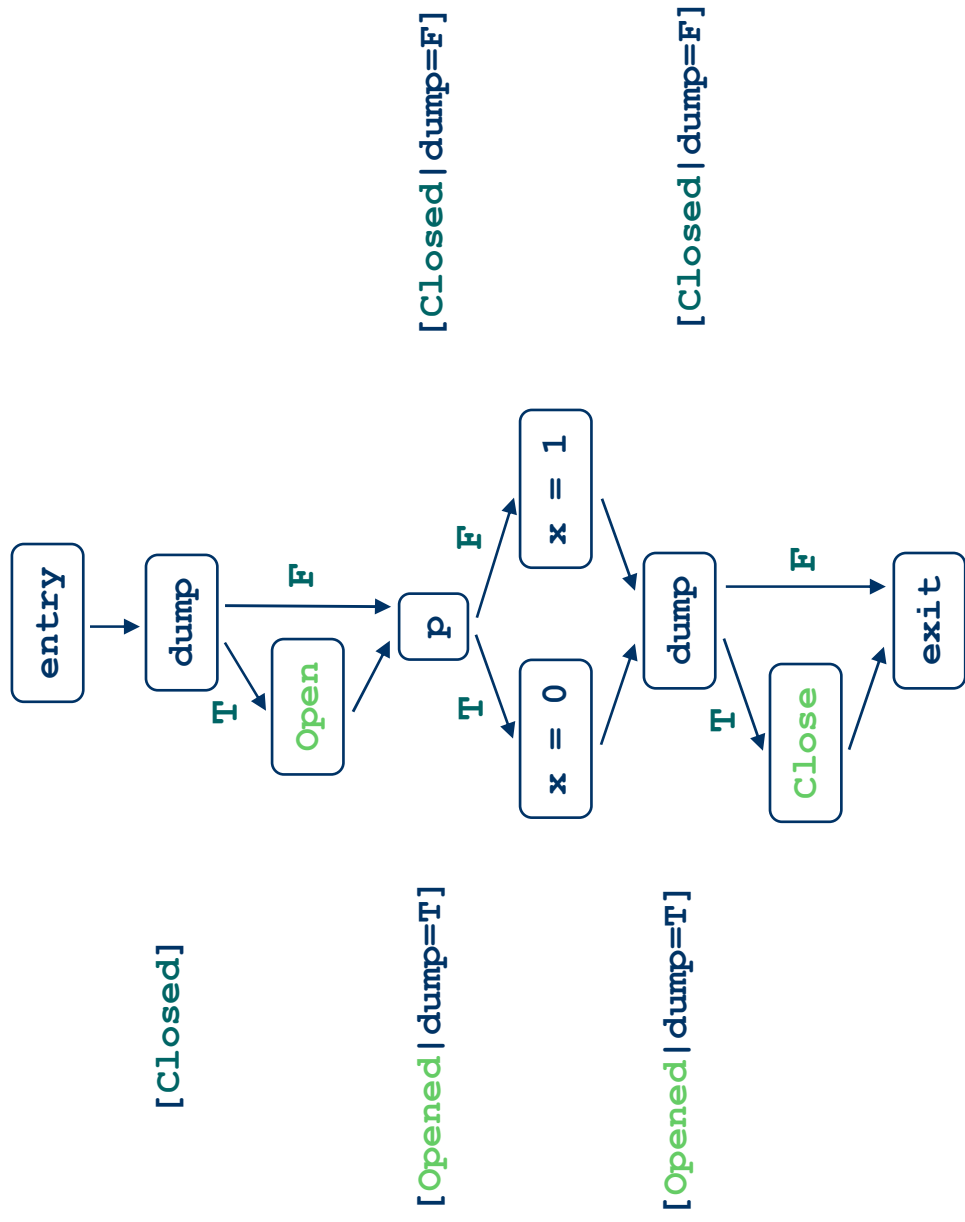
Example



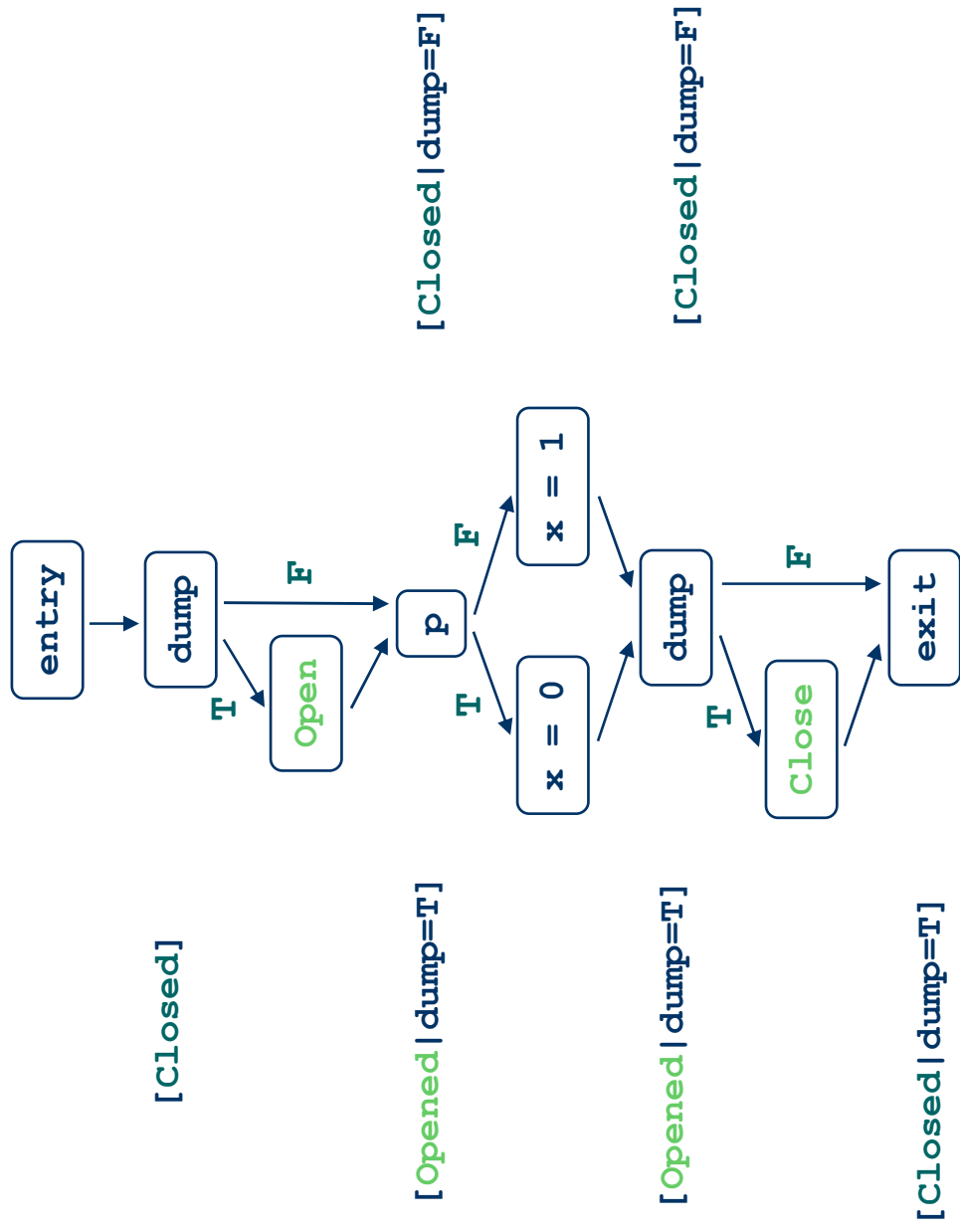
Example



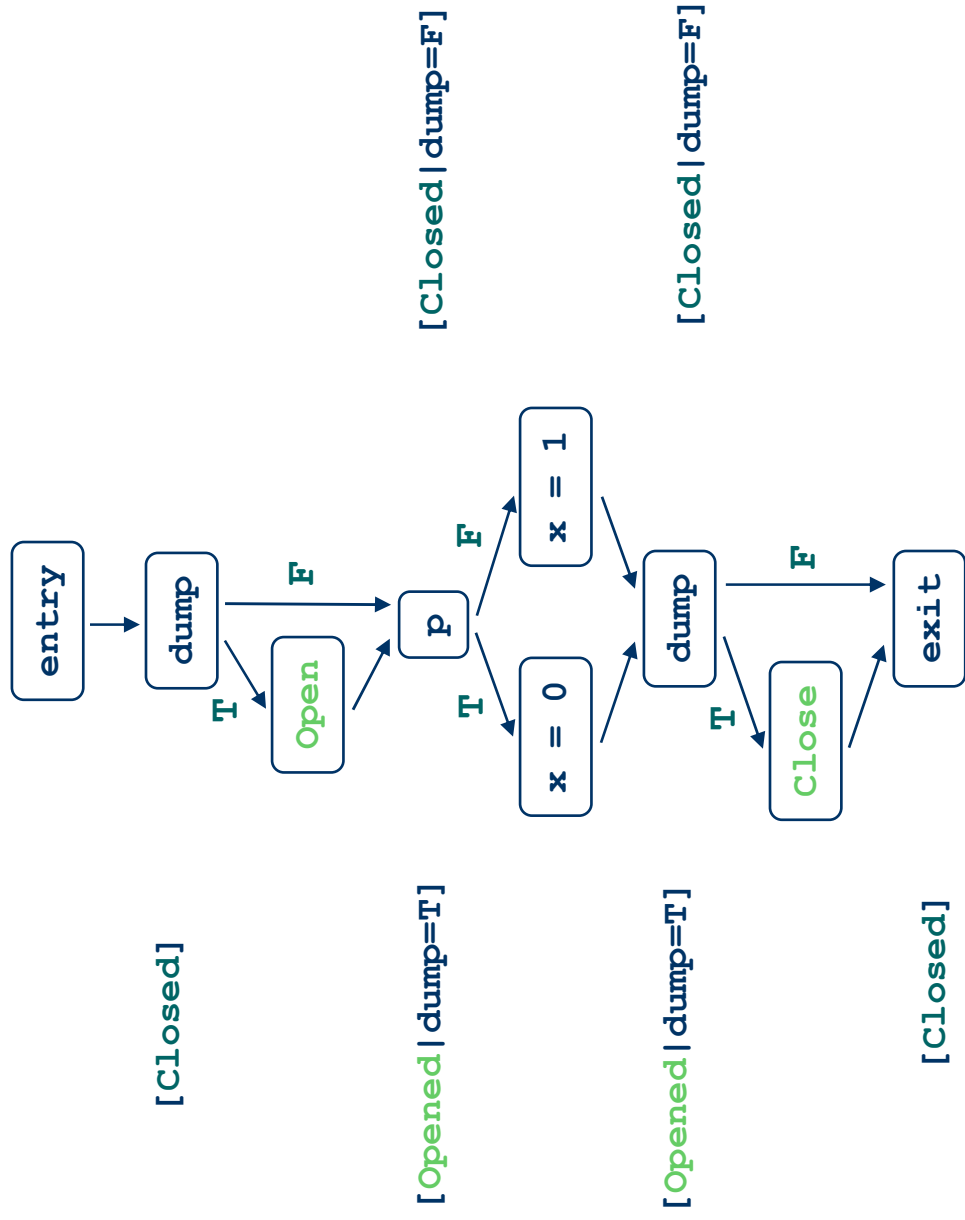
Example



Example



Example



Lessons

Forcing functions for change

- Gen 1: Manual Review
 - Too many paths
- Gen 2: Massive Testing
 - Inefficient detection of common patterns
- Gen 3: Global Program Analysis
 - Stale results
- Gen 4: Local Program Analysis
 - Lack of context
- Gen 5: Specifications

Don't bother doing this without -

- No-brainer must-haves
 - Defect viewer, docs, champions, partners
- A mechanism for developers to teach the tool
 - Suppression, assertion, assumption
- A willingness to support the tool
- A positive attitude

Myth 1 – Completeness matters

Complete == find only real bugs

- The real measure is Fix Rate
- Centralized: >50%
- Desktop: >75%
- Specification inference
 - Is it much worse than manual insertion?

Myth 2 – Soundness matters

Sound == find all the bugs

- There will never be a sound analysis
 - Partial specifications
 - Missing code
- Developers want *consistent* analysis
 - Tools should be stable w.r.t. minor code changes
 - Systematic, thorough, tunable program analysis

Myth 3 – Developers only fix real bugs

- Key attributes of a “fixable” bug
 - Easy to fix
 - Easy to verify
 - Unlikely to introduce a regression
- Simple tools can be very effective

Myth 4 – Developers hate specifications

- Control the power of the specifications
- This will work
 - Formalize invariants that are implicit in the code
- This will not work
 - Re-write code in a different language that is amenable to automated analysis
- Think like a developer

Don't break the shipping code 😊

- `__invariant()` is an annotation macro
 - generates code in the tools build, noop in the real build
- Before:
`b = a + 16; Use(b);`
- After (correct code):
`__invariant(a); b = a + 16; Use(b);`
- After (incorrect code):
`b = __invariant(a) + 16; Use(b);`
- Incorrect usage silently breaks the code!

Summary

- Program analysis technology can make a huge impact on how software is developed
- The trick is to properly balance research on new techniques with a focus on deployment
- The Center for Software Excellence (CSE) at Microsoft is doing this (well?) today

Microsoft®

<http://www.microsoft.com/cse>

<http://research.microsoft.com/manuvir>

© 2005 Microsoft Corporation. All rights reserved.

This presentation is for informational purposes only.

MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.