Static Analysis: Control- and Dataflow analysis

Claire Le Goues

October 24, 2019

FIXME Learning goals

- Define software analysis.
- Reason about QA activities with respect to coverage and coverage/adequacy criteria, both traditional (structural) and non-traditional.
- Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
- Explain at a high level why static analyses cannot be sound, complete, and terminating; assess tradeoffs in analysis design.
- Give tradeoffs and identify when various techniques might be useful.

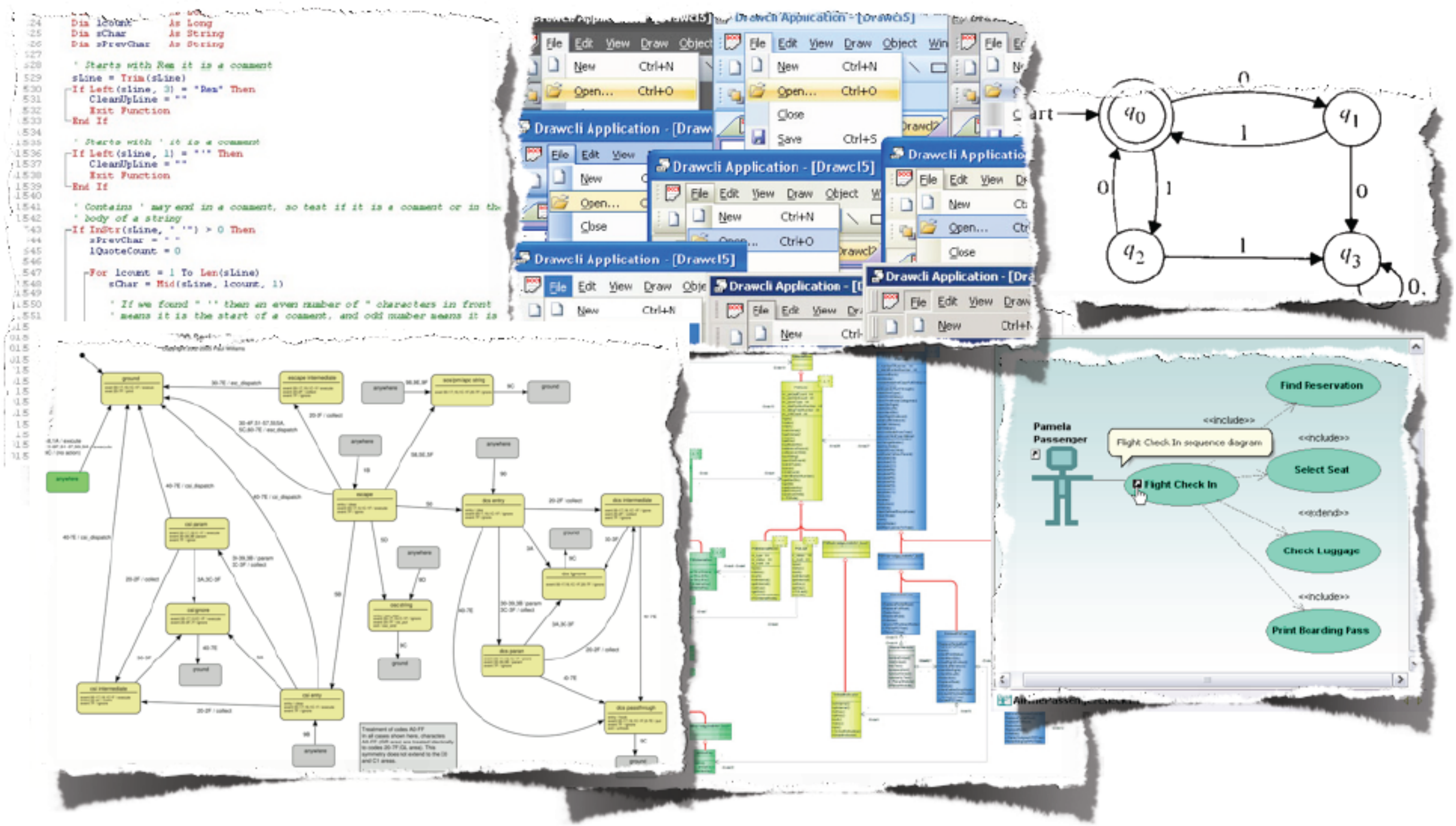
Definition: software analysis

The **systematic** examination of a software artifact to determine its properties.

Attempting to be comprehensive, as measured by, as examples:

Test coverage, inspection checklists, exhaustive model checking.

We can measure coverage on almost anything



What is Static Analysis?

- **Systematic** examination of an **abstraction** of program **state space**.
  - Does not execute code!
- **Abstraction:** produce a representation of a program that is simpler to analyze.
  - Results in fewer states to explore; makes difficult problems tractable.
- Check if a **particular property** holds over the entire state space:
  - Liveness: “something good eventually happens.”
  - Safety: “this bad thing can’t ever happen.”

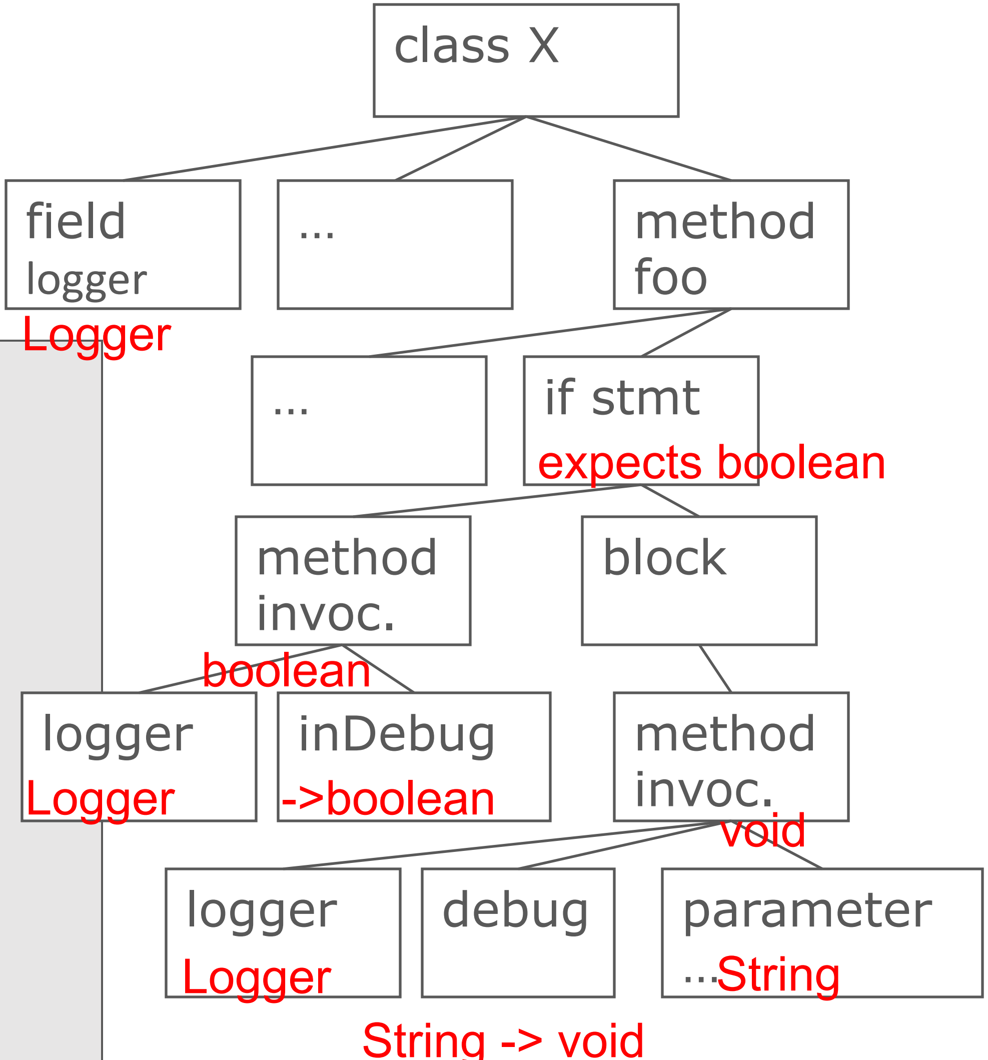
```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

ERROR: function returns with interrupts disabled!

With thanks to Jonathan Aldrich; example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Type checking

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " +
conn + "connections.");
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
    - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

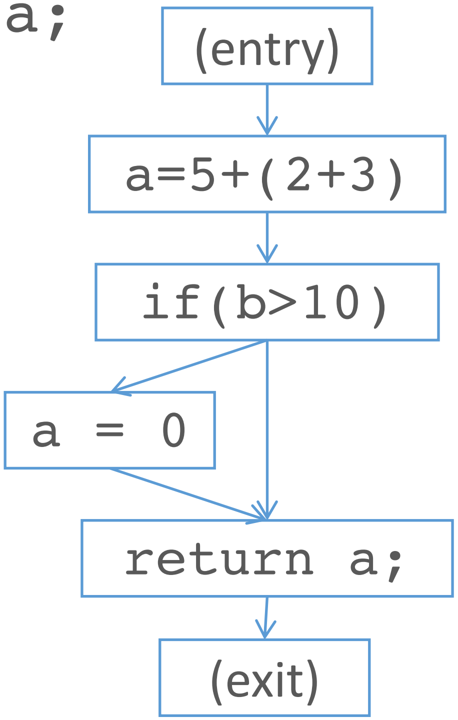
Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

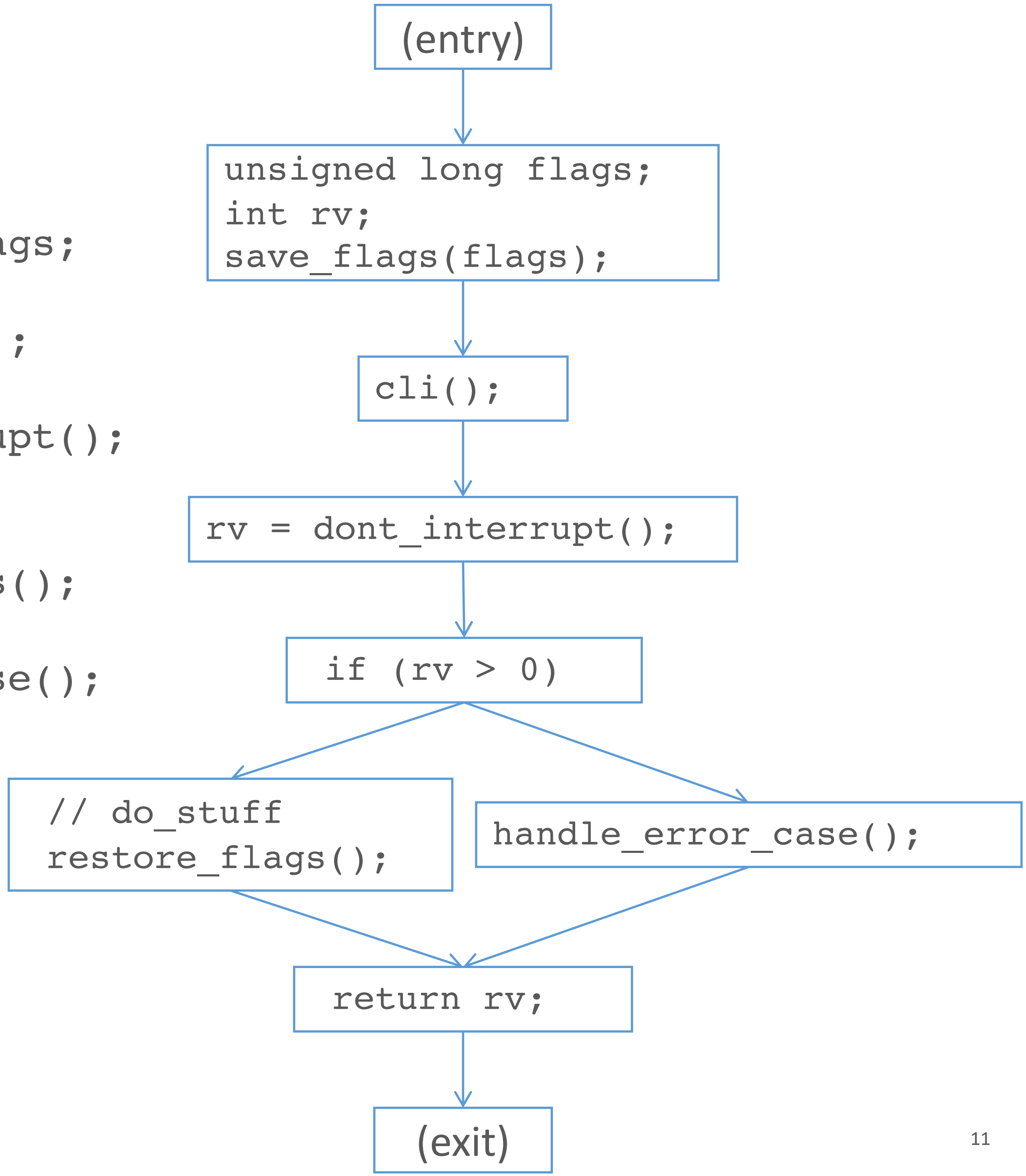
Abstraction: Control flow graphs

- A tree/graph-based representation of the flow of control through the program.
  - Captures all possible execution paths.
- Each node is a basic block: no jumps in or out.
- Edges represent control flow options between nodes.
- Intra-procedural: within one function.
  - cf. inter-procedural

```
1. a = 5 + (2 + 3)
2. if (b > 10) {
3.    a = 0;
4. }
5. return a;
```



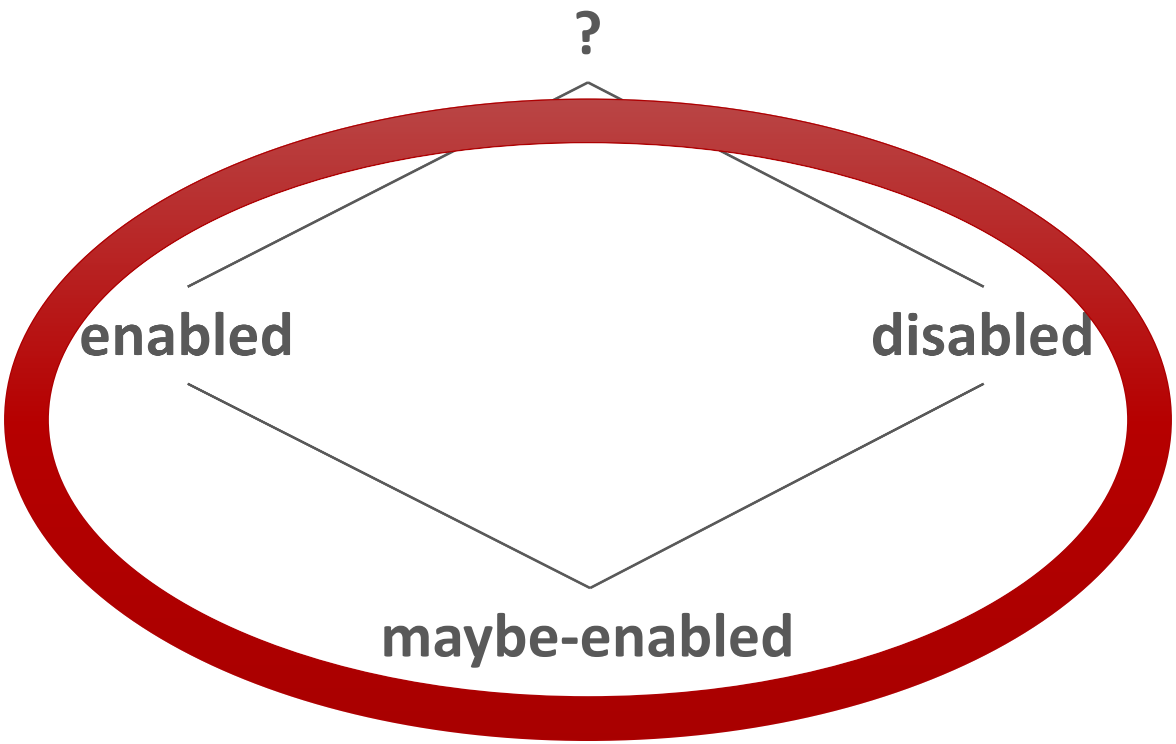
```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    if (rv > 0) {
       // do_stuff
        restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every program point.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

Example: interrupt checker

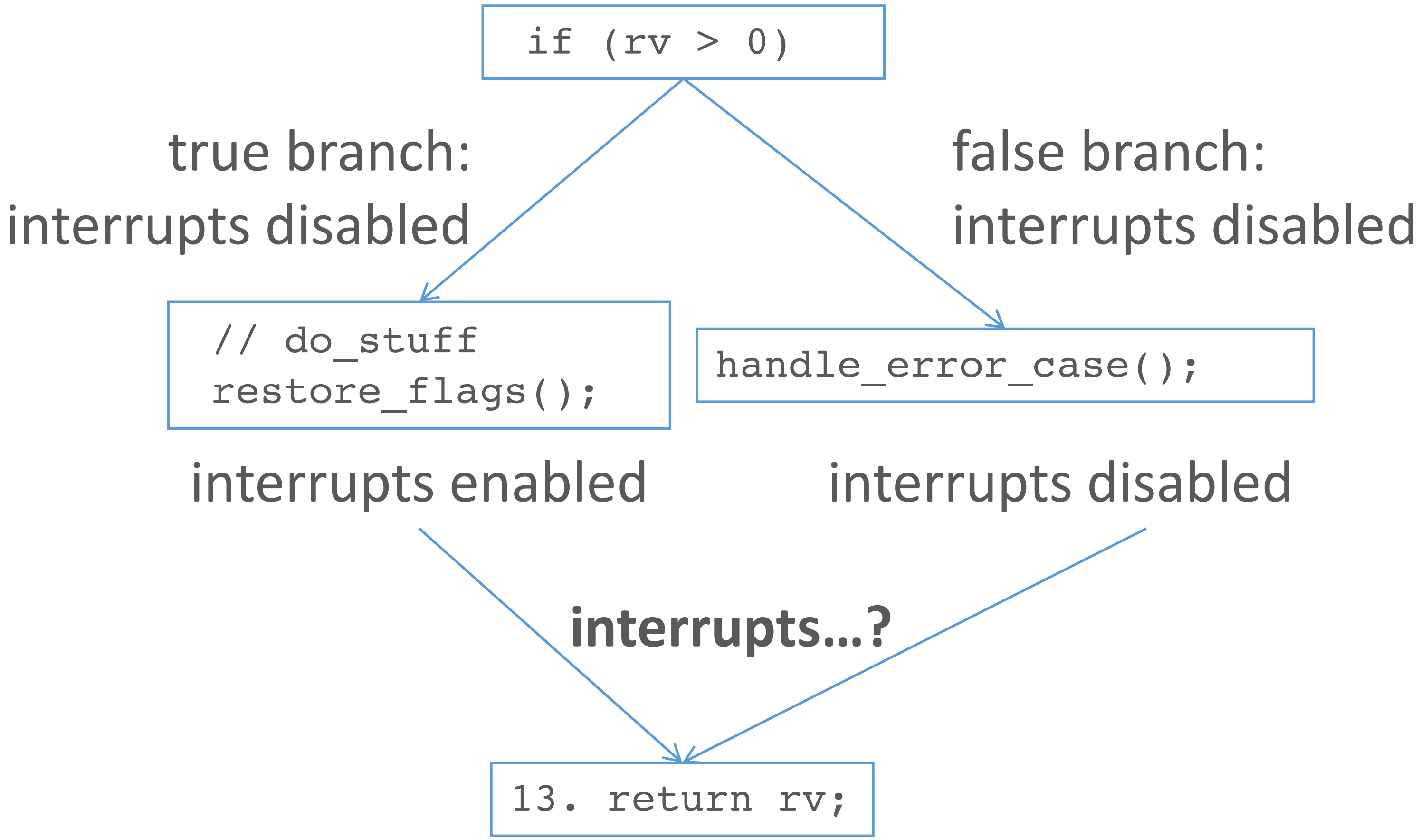


An interrupt checker

- **Abstraction**
  - Three abstract states: enabled, disabled, maybe-enabled
  - Warning if we can reach the end of the function with interrupts disabled.
- Transfer function:
  - If a basic block includes a call to `cli()`, then it moves the state of the analysis from **disabled** to **enabled**.
  - If a basic block includes a call to `restore_flags()`, then it moves the state of the analysis from **enabled** to **disabled**.

Join

assume: pre-block program point: interrupts disabled



Join: abstract!

assume: pre-block program point: interrupts disabled

```
if (rv > 0)
```

true branch:
interrupts disabled

false branch:
interrupts disabled

```
// do_stuff
restore_flags();
```

```
handle_error_case();
```

interrupts enabled

interrupts disabled

Join(enabled, disabled) → maybe enabled

```
13. return rv;
```

(Note: this is where information gets "lost.")

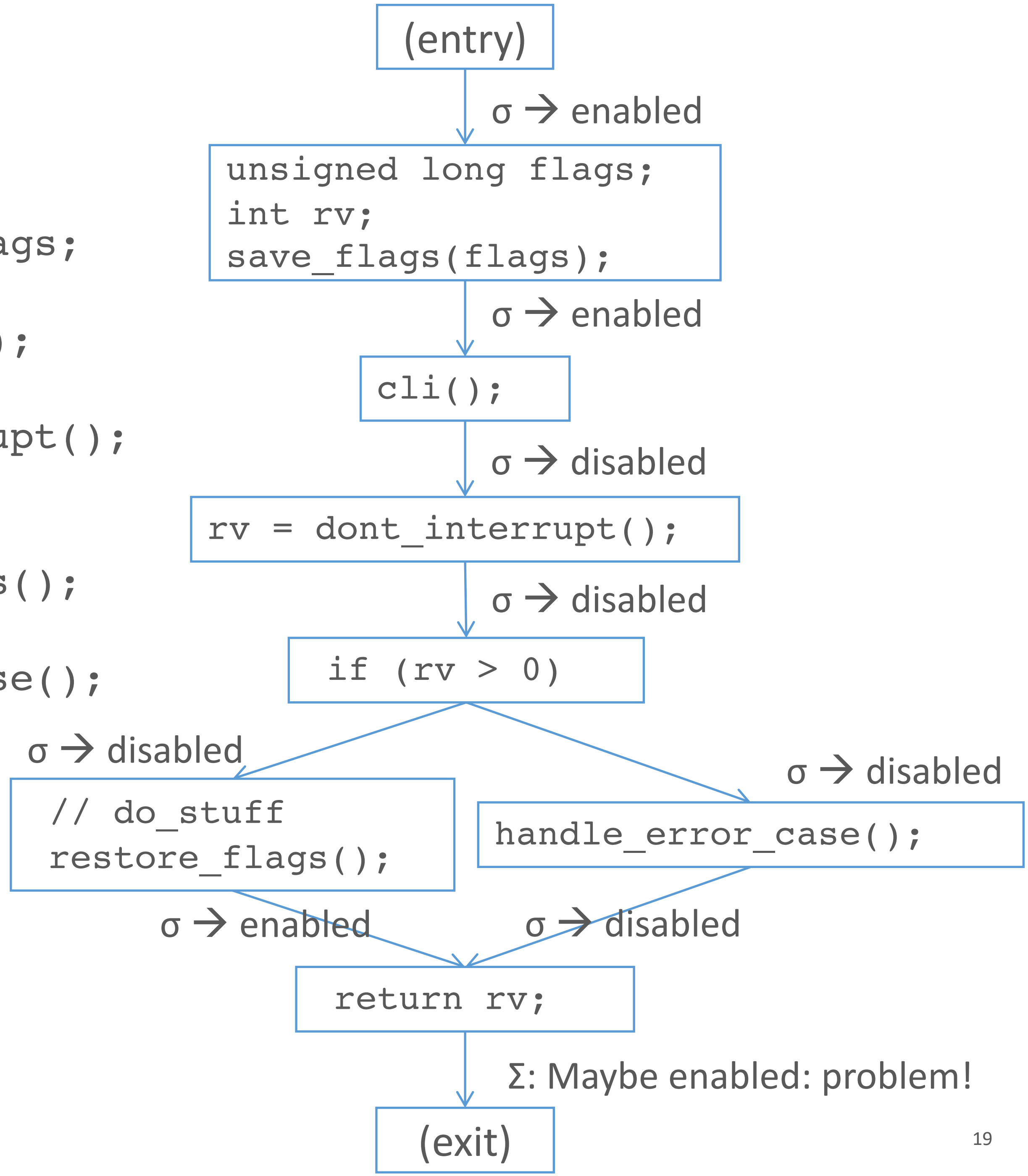
Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
    - Track information relevant to a property of interest at every program point.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

Reasoning about a CFG

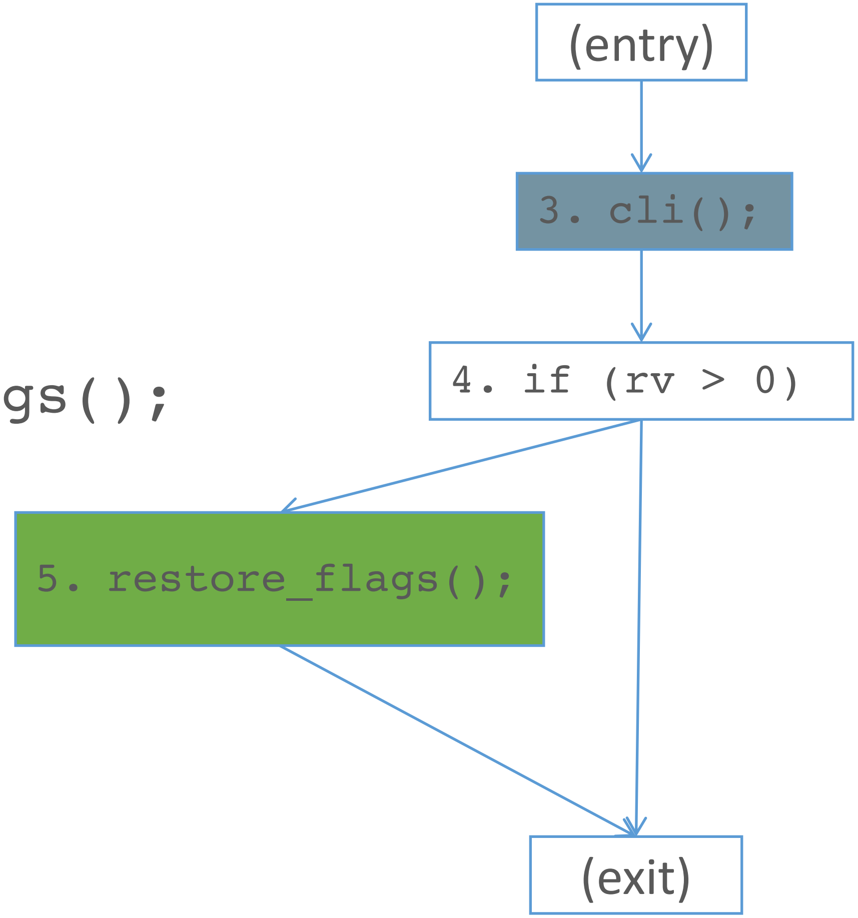
- Analysis updates state at *program points:* points between nodes.
- For each node:
  - determine state on entry by examining/combining state from predecessors.
  - evaluate state on exit of node based on effect of the operations (*transfer*).
- *Iterate through successors and over entire graph until the state at each program point stops changing.*
- **Output: state at each program point**

```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    if (rv > 0) {
       // do_stuff
        restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



Abstraction

```
1. void foo() {
2.     …
3.     cli();
4.    if (a) {
5.        restore_flags();
6.     }
7. }
```



Zero/Null-pointer Analysis: the same thing, but per-variable rather than per-function

- Could a variable x ever be 0?
    - (what kinds of errors could this check for?)
- Original domain: N maps every variable to an integer.
- Abstraction**:** every variable is non zero (NZ), zero(Z), or maybe zero (MZ)

Example

- Consider the following program:

```
x = 10;
y = x;
z = 0;
while (y > -1) {
  x = x/y;
  y = y-1;
  z = 5;
}
```

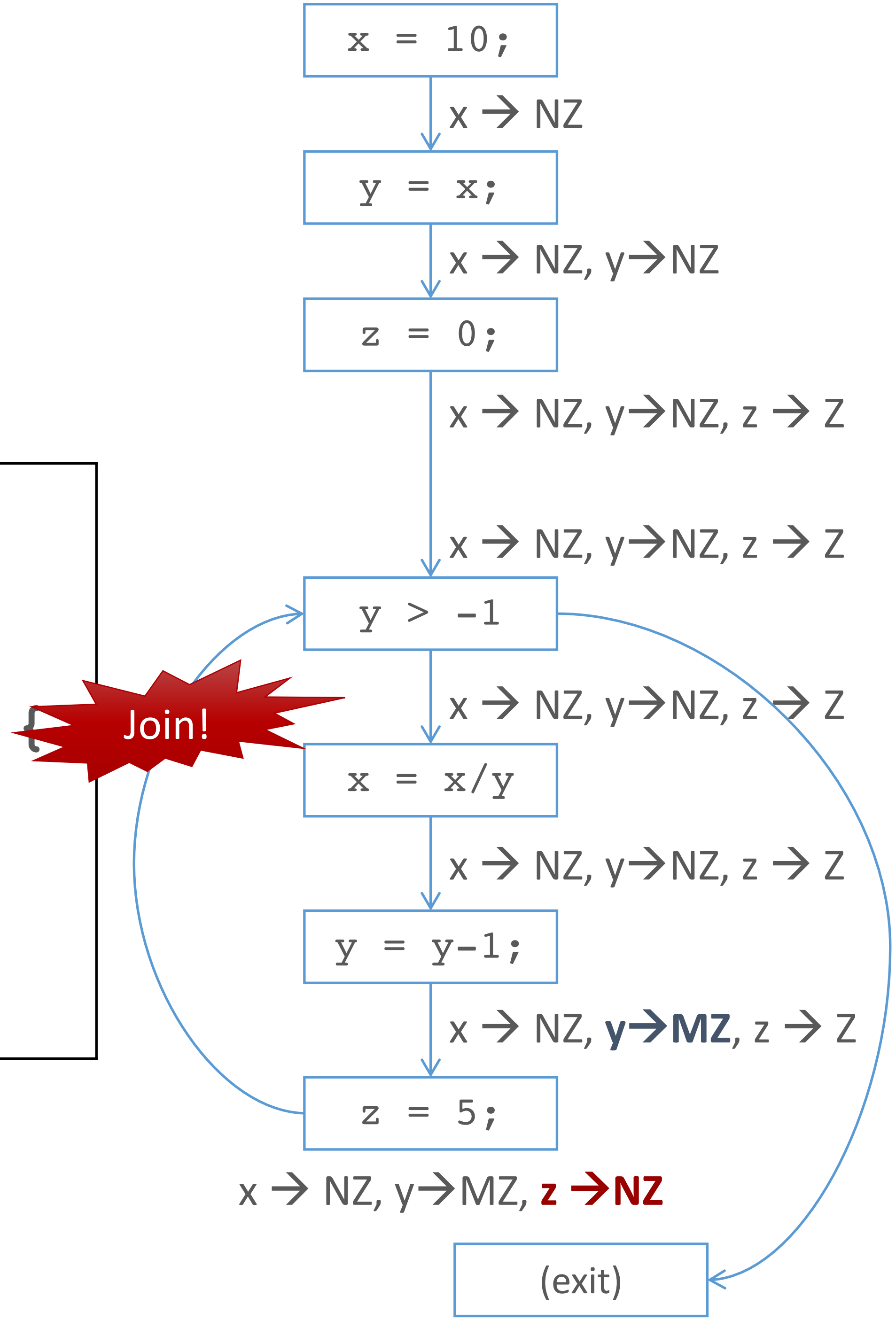
- Original domain: N maps every variable to an integer.
- Abstraction**:** every variable is non zero (NZ), zero(Z), or maybe zero (MZ)

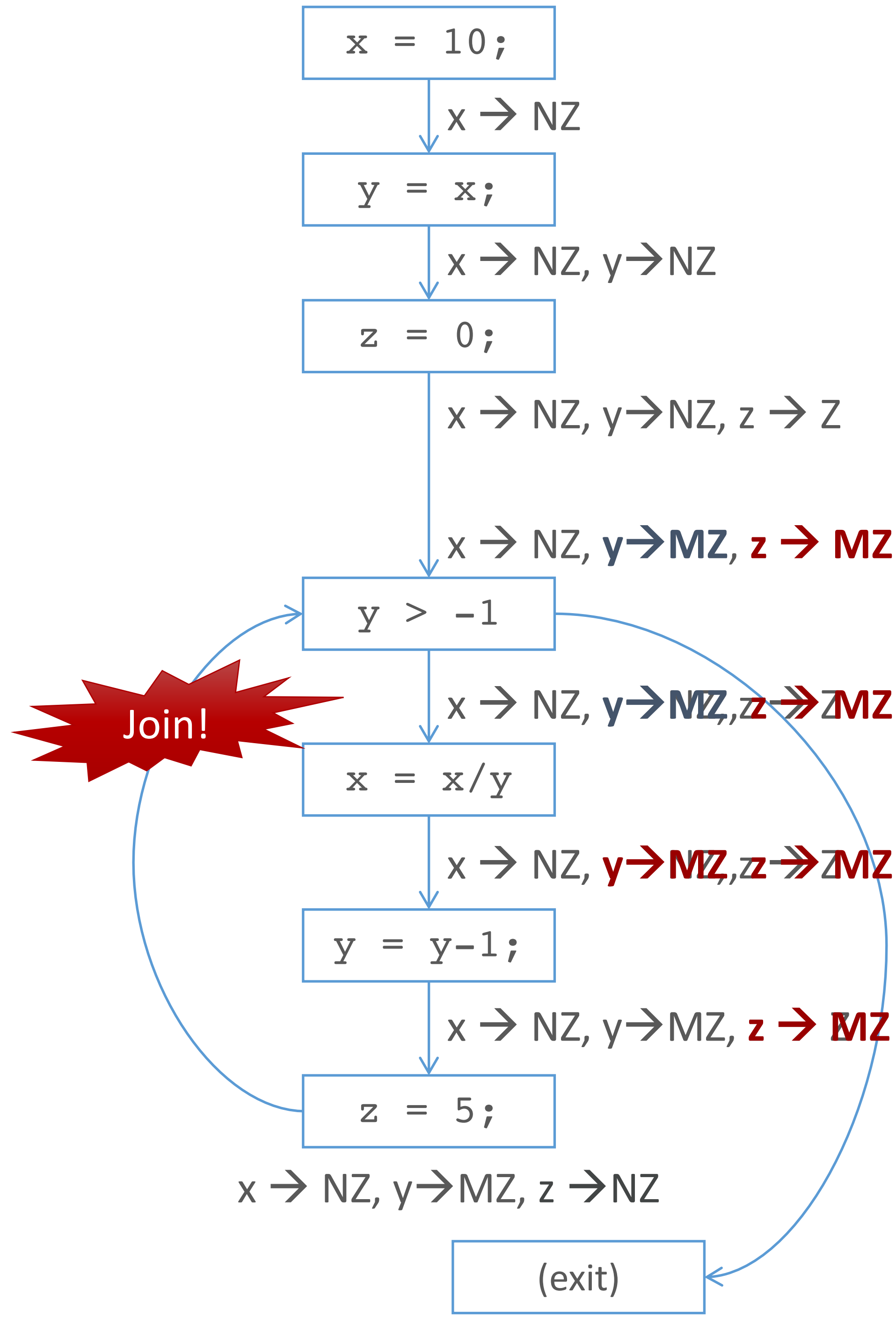
- Use **zero analysis** to determine if y could be zero at the division.

Reminder:

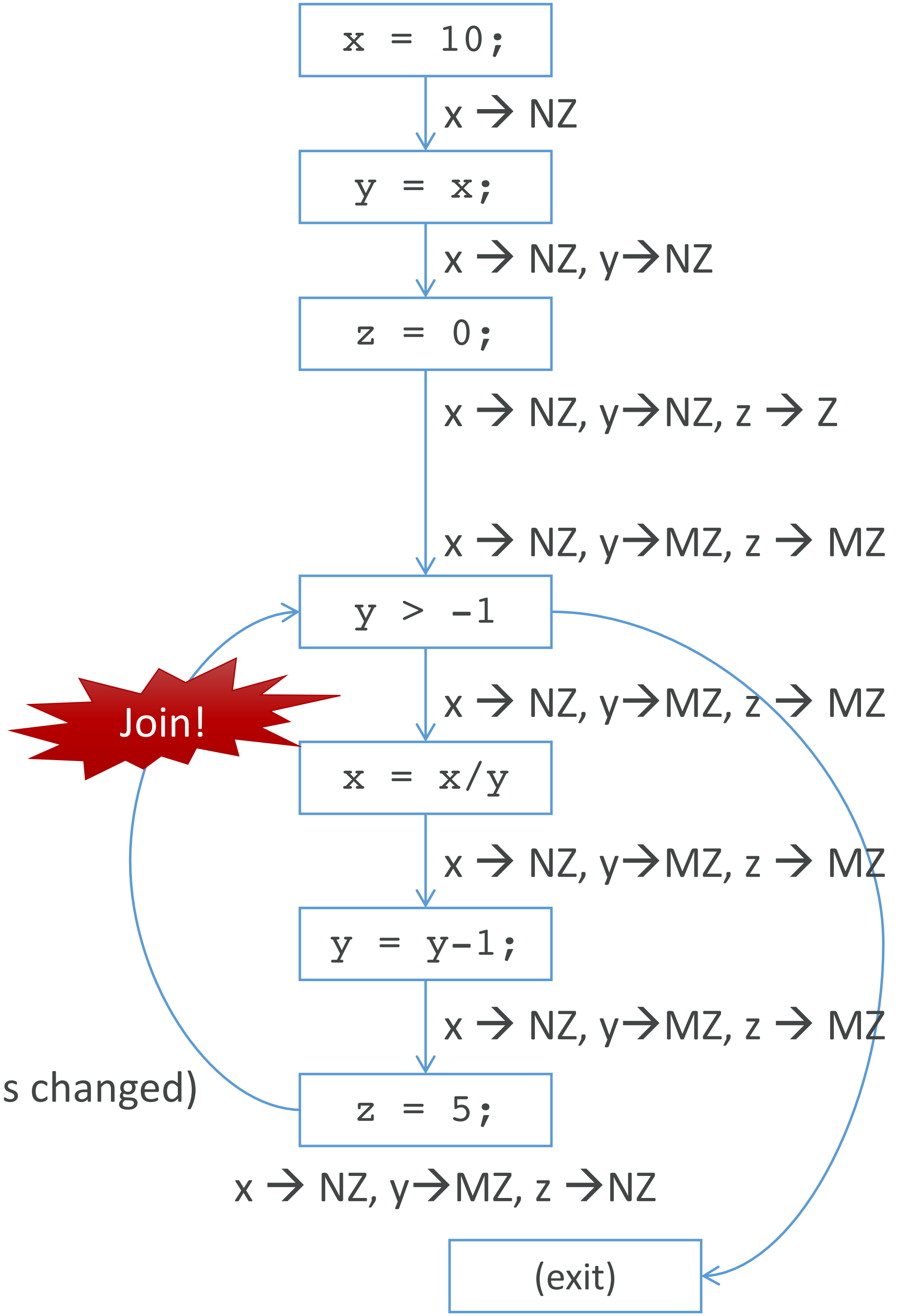
x: Join(NZ,NZ) → NZ
y: Join(MZ,NZ) → MZ
Z: Join(NZ, Z) → MZ

```
x = 10;
y = x;
z = 0;
while (y > -1) {
  x = x/y;
  y = y-1;
  z = 5;
}
```



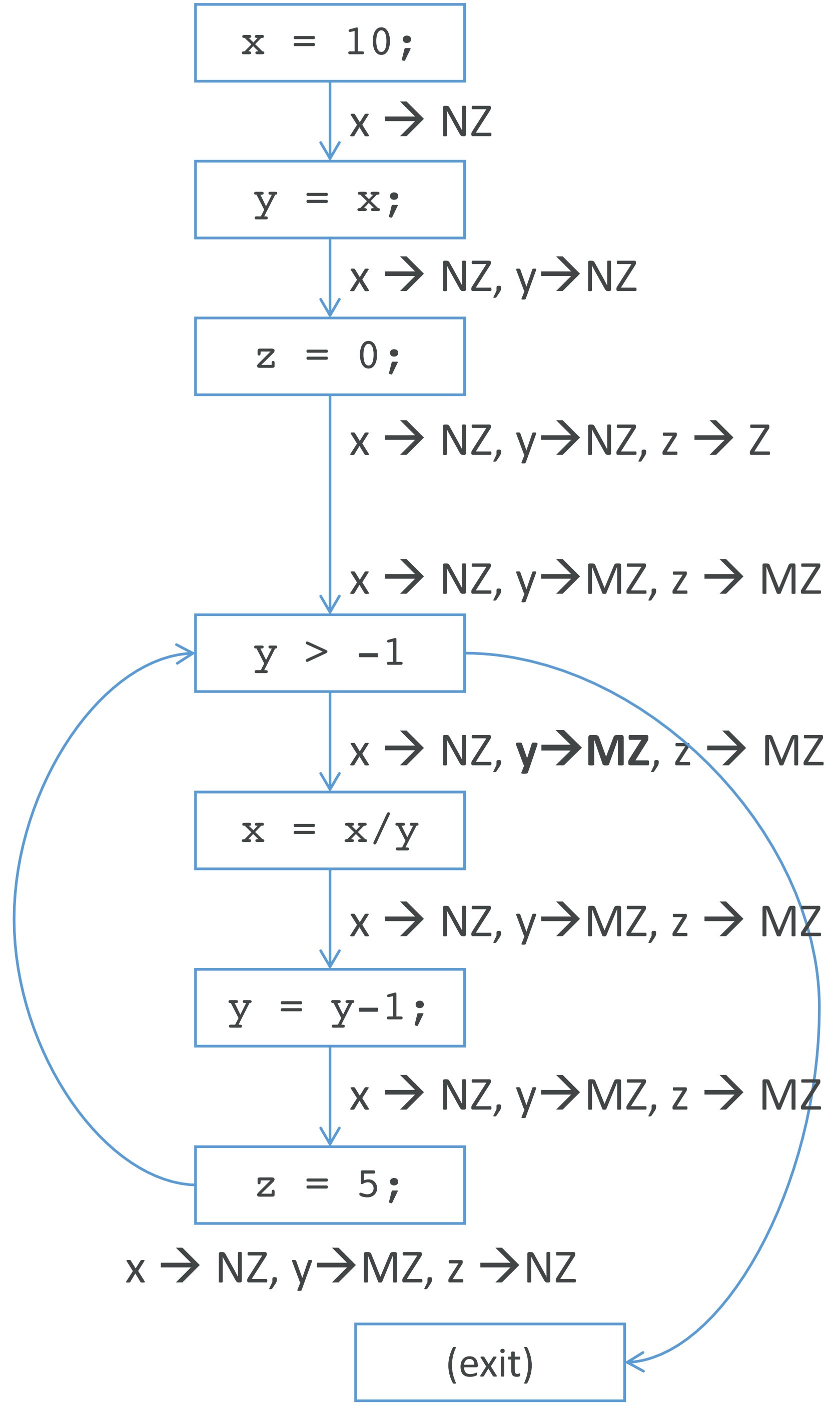


(end of iteration 2)



(end of iteration 3; nothing has changed)

Warning! Possible division by zero error!



Abstraction at work

- Number of possible states gigantic
  - n 32 bit variables results in 2^{32*n} states
    - $2^{(32*3)} = 2^{96}$
  - With loops, states can change indefinitely
- Zero Analysis narrows the state space
  - Zero or not zero
  - $2^{(2*3)} = 2^{6}$
  - When this limited space is explored, then we are done
    - Extrapolate over all loop iterations

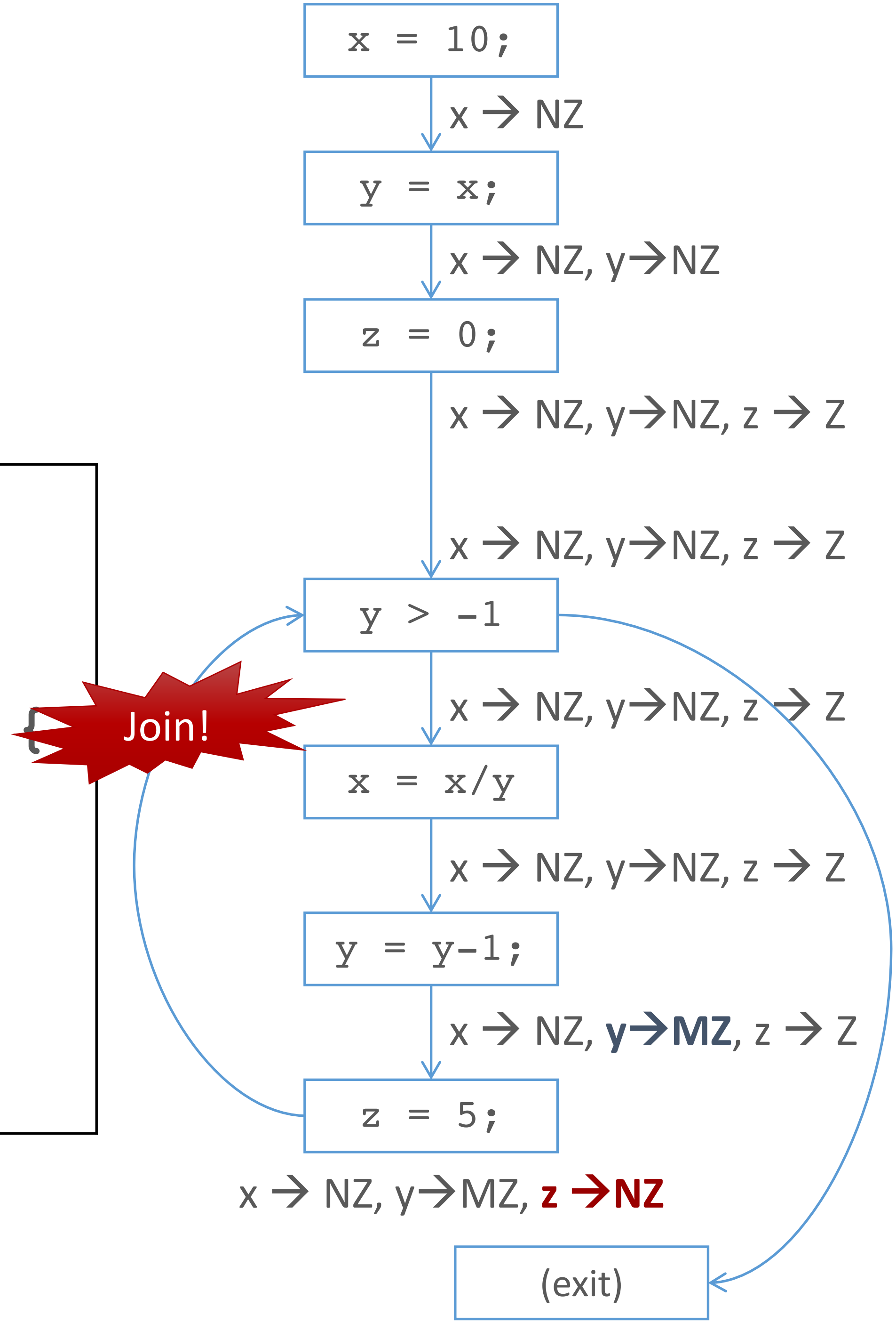
Reminder:

x: Join(NZ,NZ) → NZ

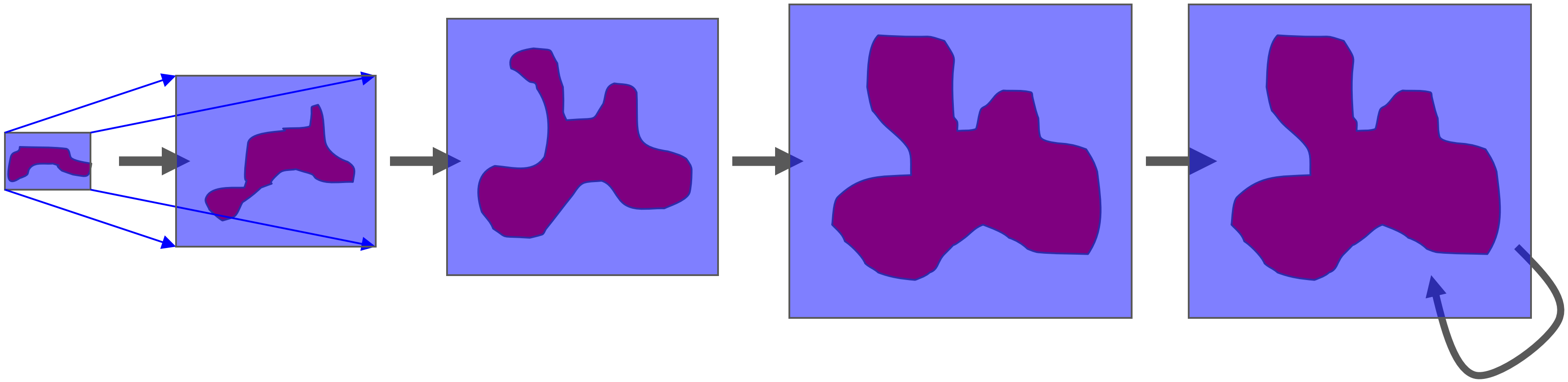
y: Join(MZ,NZ) → MZ

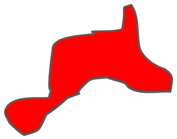
Z: Join(NZ, Z) → MZ

```
x = 10;
y = x;
z = 0;
while (y > -1) {
  x = x/y;
  y = y-1;
  z = 5;
}
return arr[y];
```



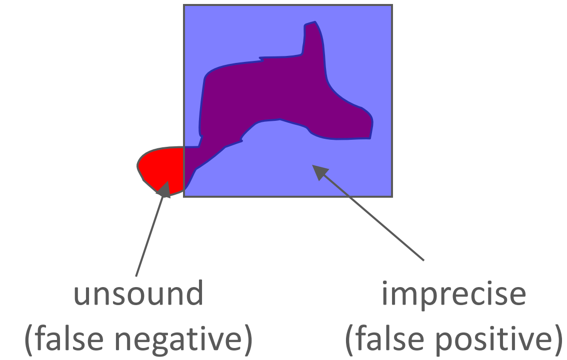
Soundness and precision





Program state covered in actual execution

Program state covered by abstract execution with analysis



Sound vs. Heuristic Analysis

- Heuristic Analysis
  - FindBugs, checkstyle, …
  - Follow rules, approximate, avoid some checks to reduce false positives
  - May report false positives and false negatives
- Sound Static Analysis
  - Type checking, Not-Null, … (specific fault classes)
  - Sound abstraction, precise analysis to reduce false positives

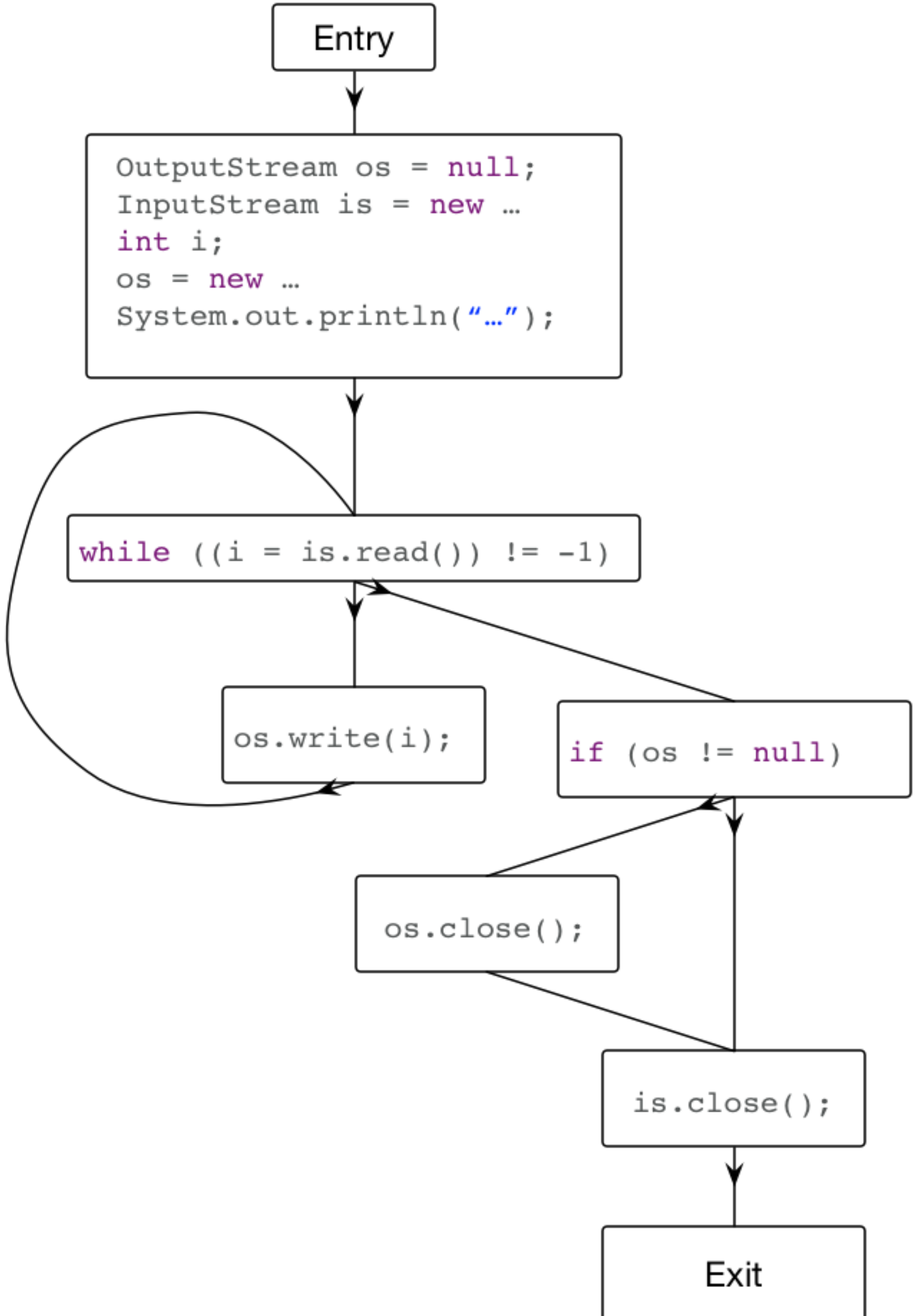
File open/close

- Abstract domain: file open, file closed, file maybe-open.
- Transfer and joins left as exercise to the reader…

```java
public class StreamDemo {
  public static void main(String[] args) throws Exception {
     OutputStream os = null;
     InputStream is = new FileInputStream("in.txt");
     int i;
     try {
      os = new FileOutputStream("out.txt");
      System.out.println("Copying in progress...");
      while ((i = is.read()) != -1) {
         os.write(i);
      }
      if (os != null) {
        os.close();
       }
     } catch (IOException e) {
        e.printStackTrace();
     }
      is.close();
    }
}
```

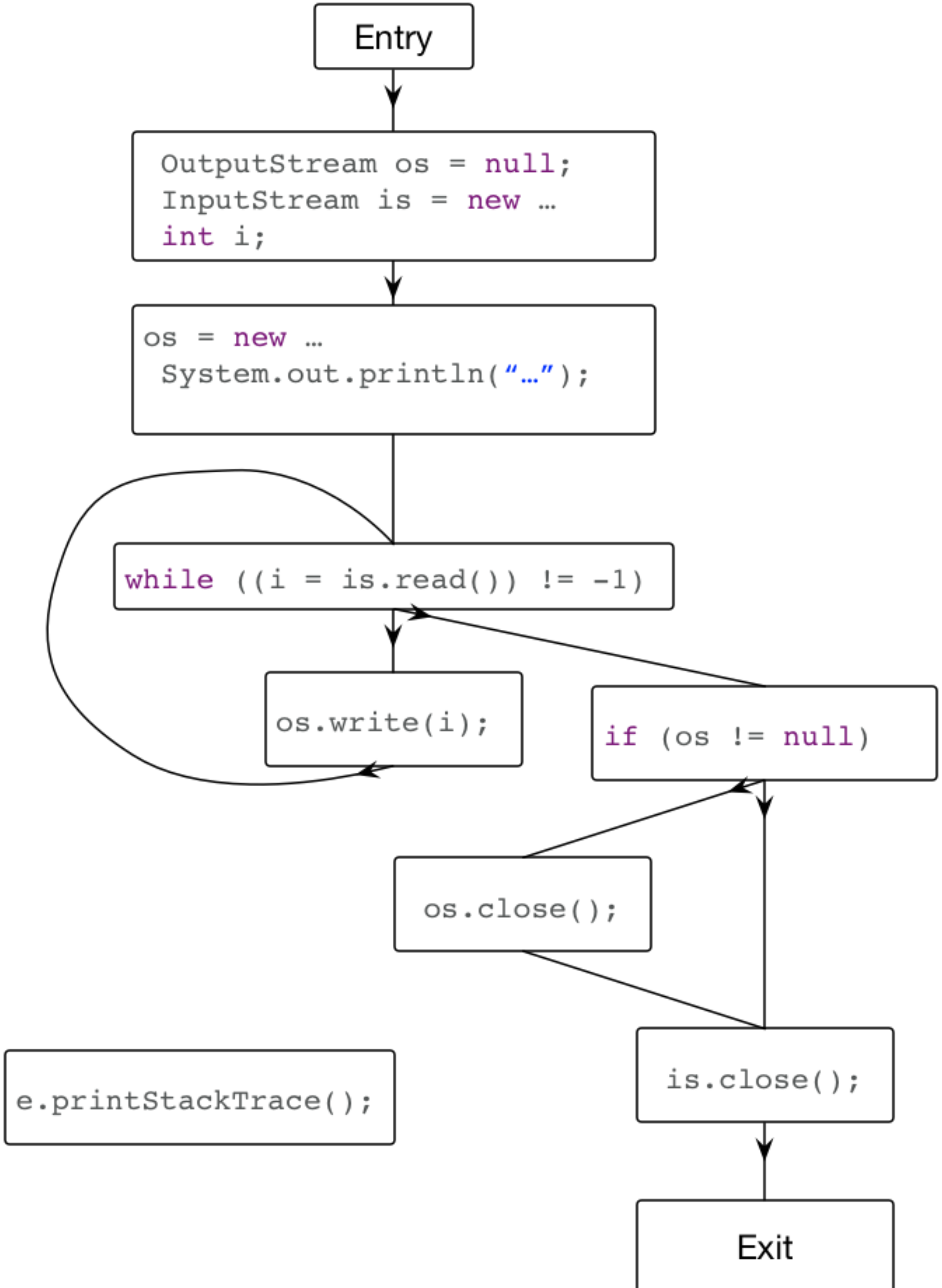
Design choices: representation and abstract domain

- What if we don't model the try/catch?


Entry
OutputStream os = null;
InputStream is = new …
int i;
os = new …
System.out.println("…");
while ((i = is.read()) != -1)
os.write(i);
if (os != null)
os.close();
is.close();
Exit

Design choices: representation and abstract domain

- What if we don't model the try/catch?
- If we do…how should we include it?


Entry
OutputStream os = null;
InputStream is = new …
int i;
os = new …
System.out.println("…");
while ((i = is.read()) != -1)
os.write(i);
if (os != null)
os.close();
is.close();
e.printStackTrace();
Exit

Design choices: representation and abstract domain

- What if we don’t model the try/catch?
- If we do…how should we include it?
- …what about non-IOExceptions?
- Broader question: How precisely should we model semantics?
    - E.g., Of instructions, of conditional checks, etc.

Upshot: analysis as approximation

- Analysis must approximate in practice
  - False positives: may report errors where there are really none
  - False negatives: may not report errors that really exist
  - All analysis tools have either false negatives or false positives
- Approximation strategy
  - Find a pattern P for correct code
    - which is feasible to check (analysis terminates quickly),
    - covers most correct code in practice (low false positives),
    - which implies no errors (no false negatives)
- Analysis can be pretty good in practice
  - Many tools have low false positive/negative rates
  - A sound tool has no false negatives
    - Never misses an error in a category that it checks