Symbolic Execution

Advanced Topics in Software Systems

Stefan Bucur // 23.09.2011

School of Computer and Communication Sciences

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```

15

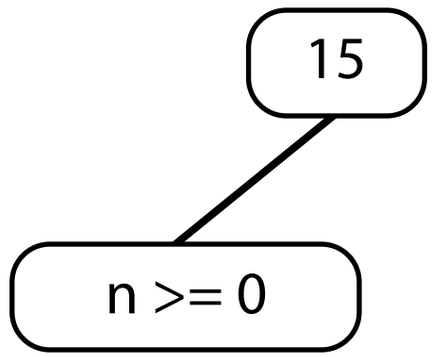
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```

15

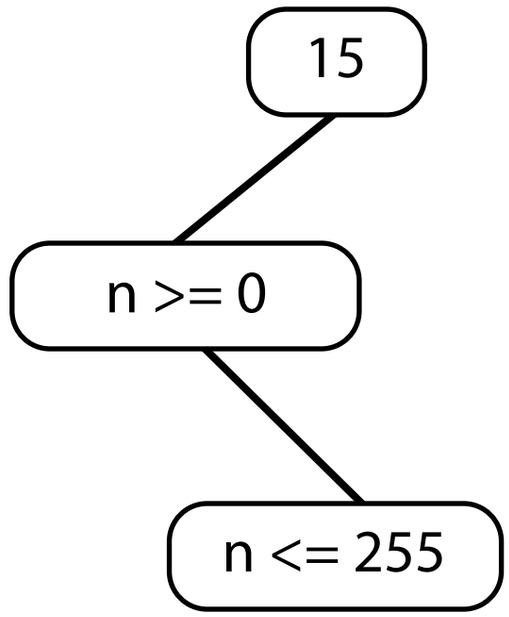
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



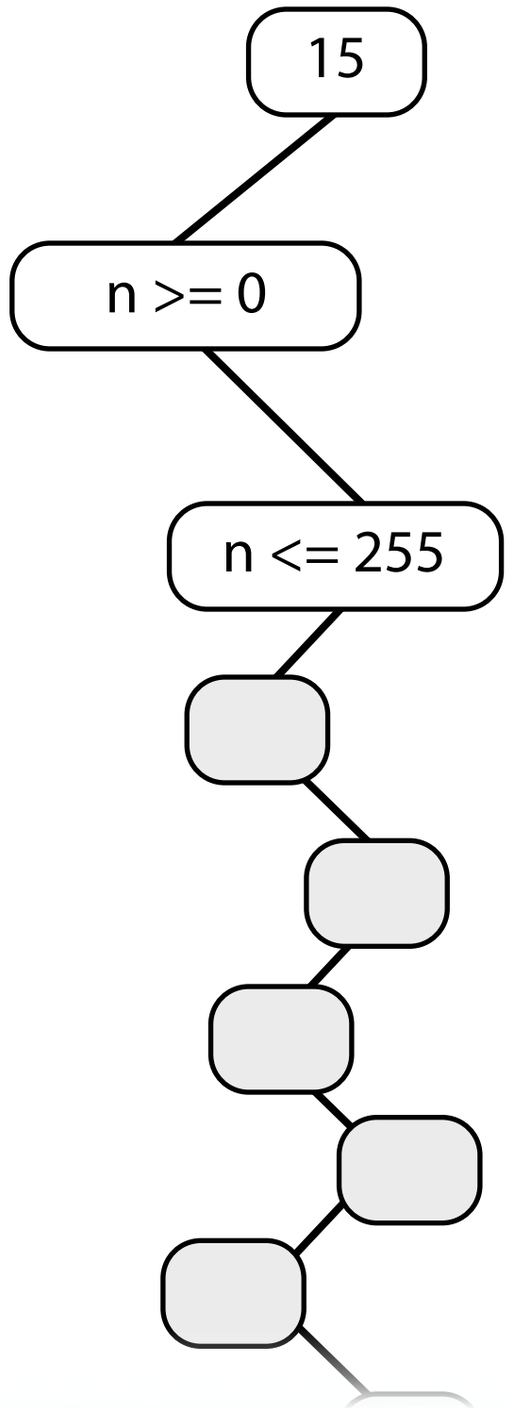
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



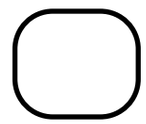
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



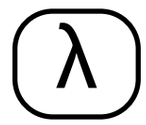
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



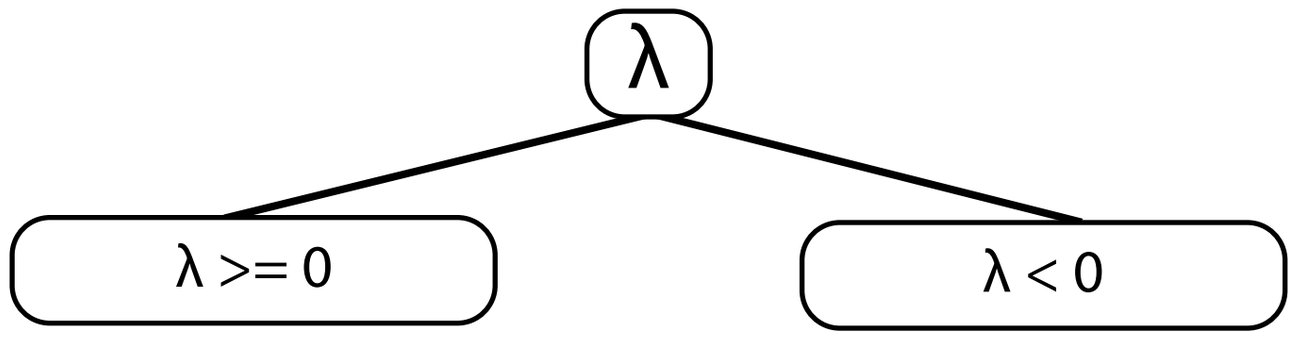
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



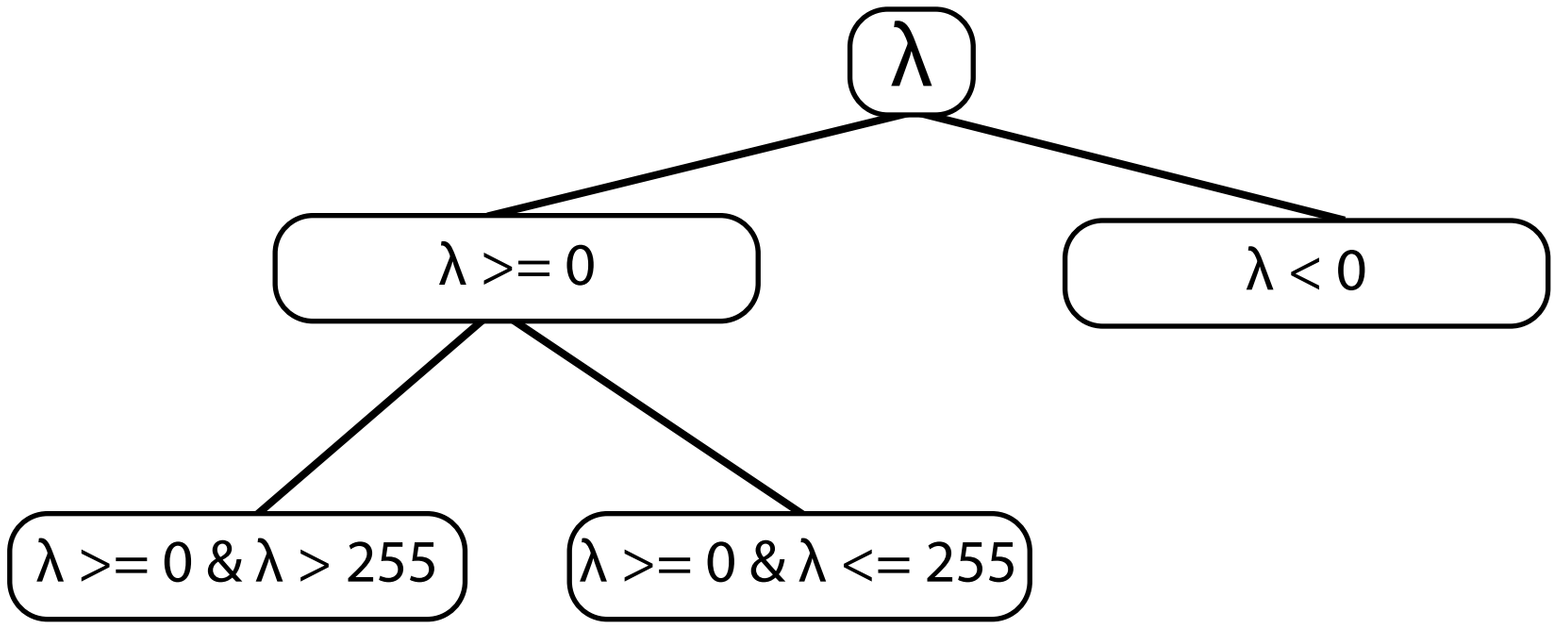
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



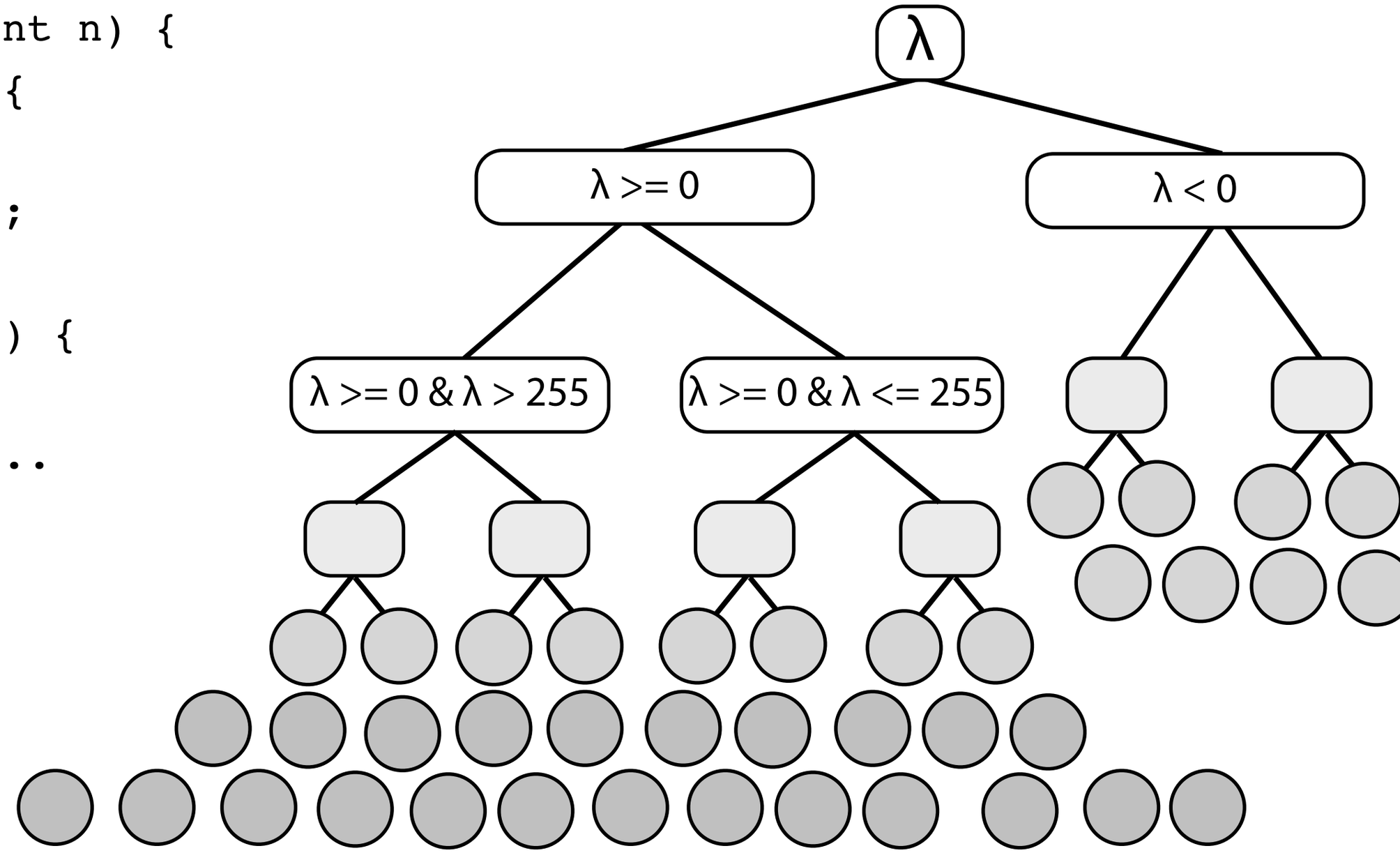
Symbolic Execution

```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```

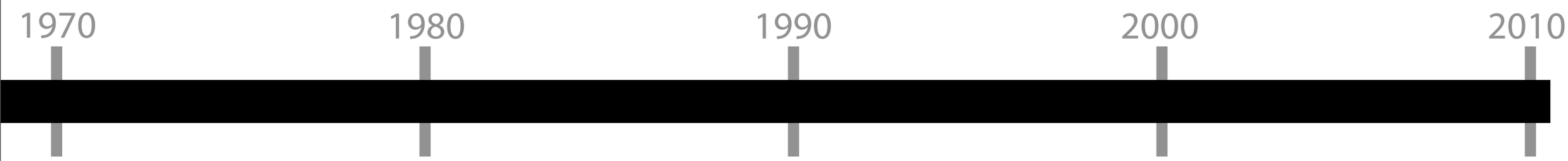


Symbolic Execution

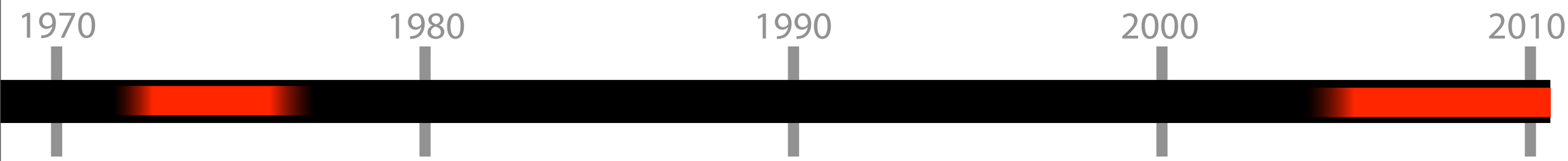
```
int compute(int n) {
  if (n < 0) {
    err(n);
    return -1;
  }
  if (n > 255) {
    ...
  } else if ...
  ...
}
```



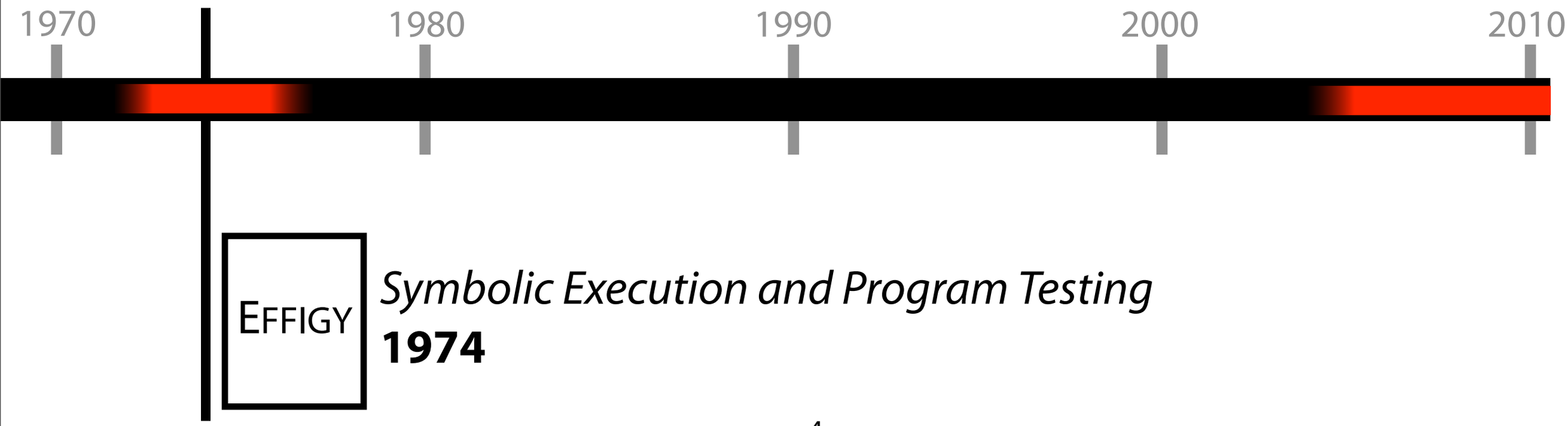
Timeline



Timeline



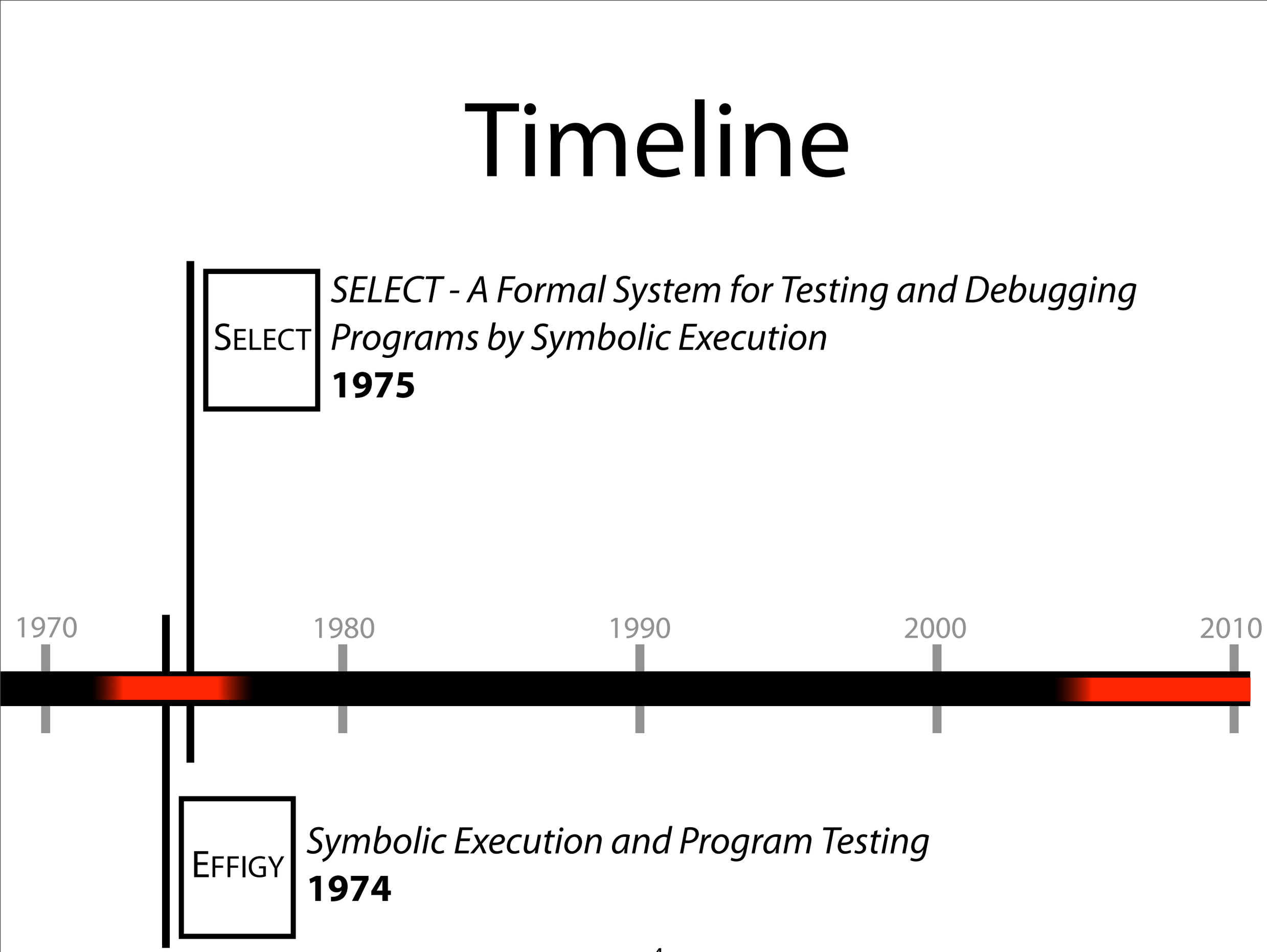
Timeline


1970
1980
1990
2000
2010
EFFIGY
Symbolic Execution and Program Testing
1974

Timeline

SELECT

SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution
1975



EFFIGY

Symbolic Execution and Program Testing
1974

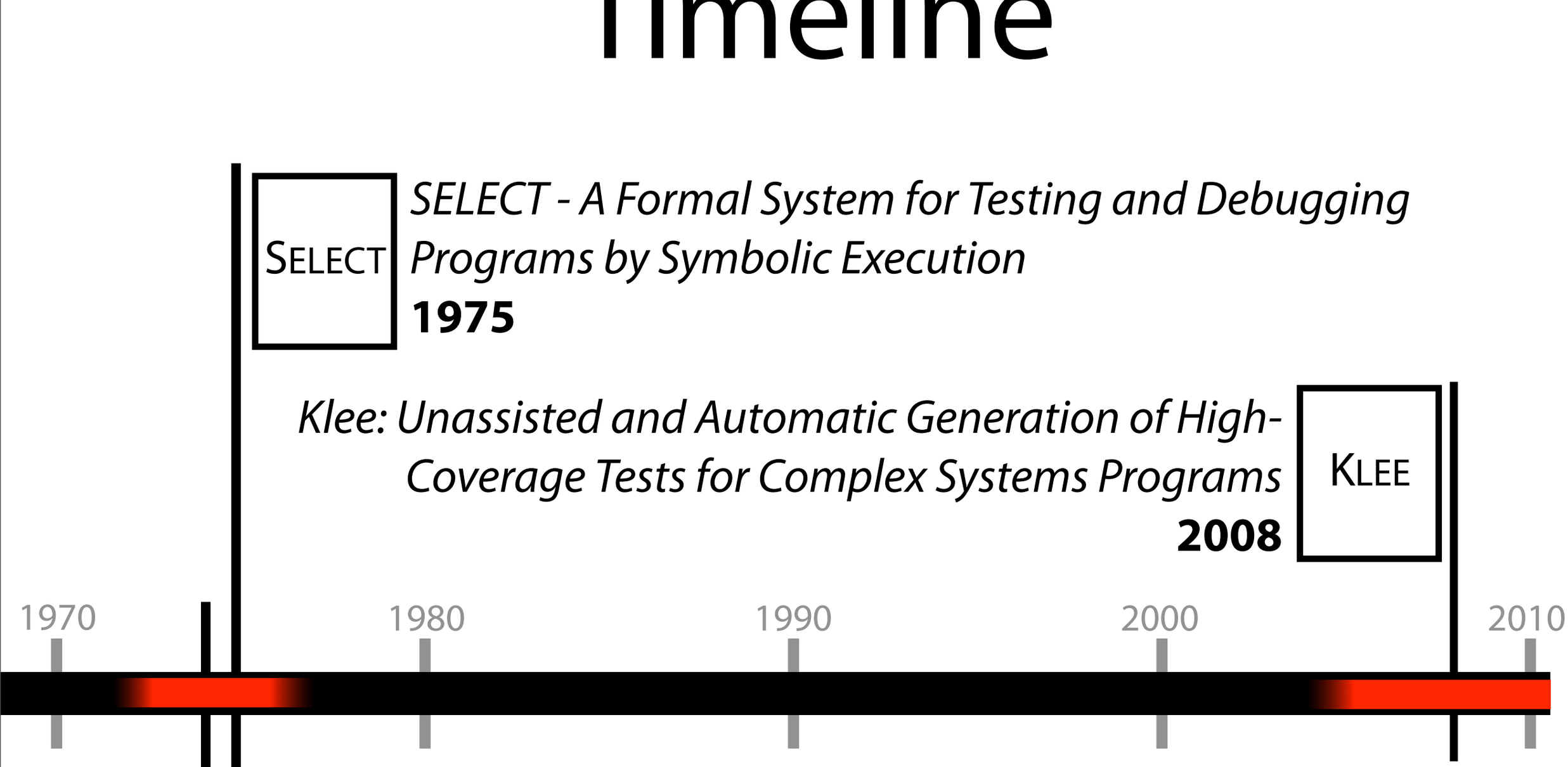
Timeline

SELECT

SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution
1975

Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
2008

KLEE



EFFIGY

Symbolic Execution and Program Testing
1974

Outline

- Symbolic Execution
- EFFIGY and SELECT
- KLEE
- Final Remarks

Outline

- **Symbolic Execution**
- EFFIGY and SELECT
- KLEE
- Final Remarks

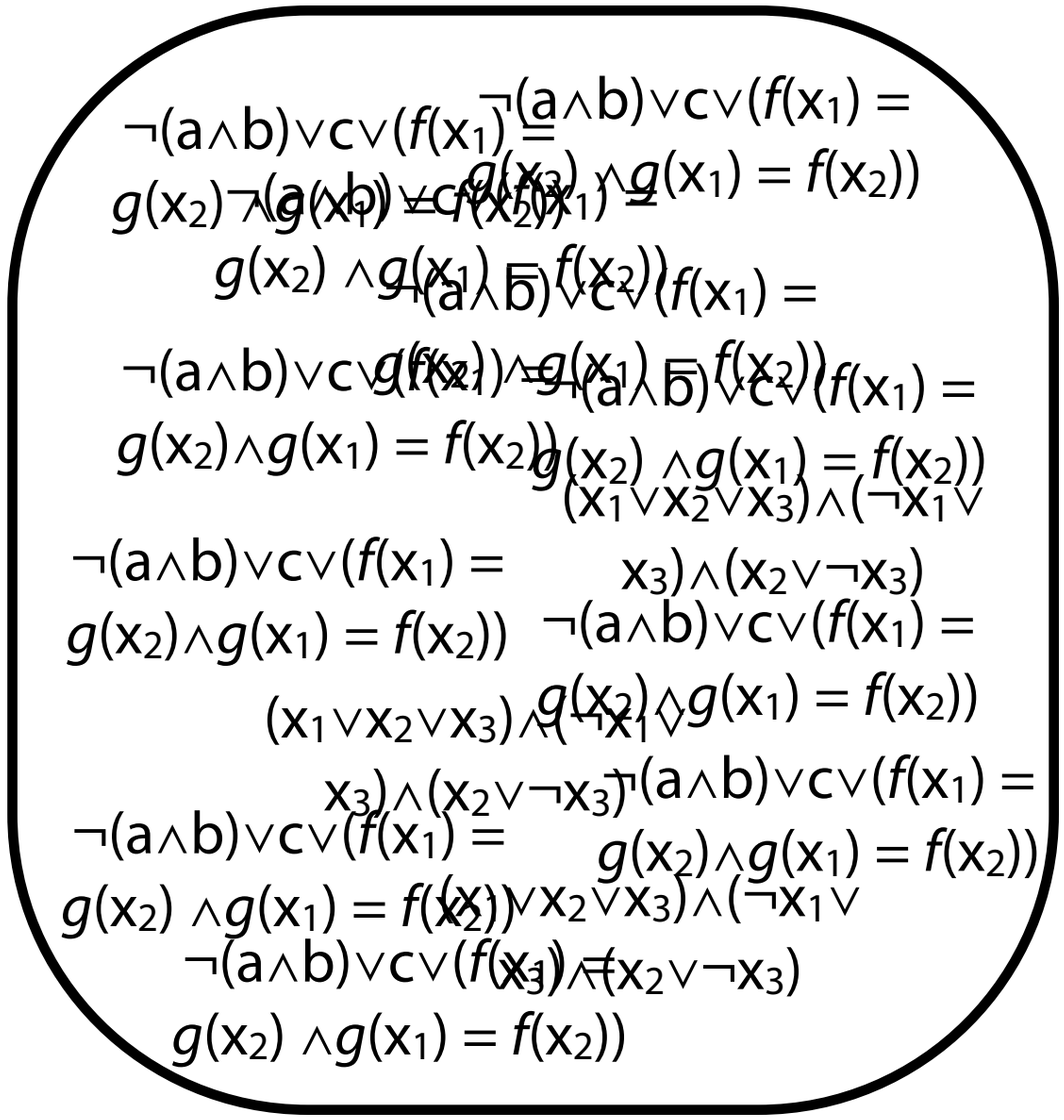
Formal Verification

```
static in_port_t
__get_unused_port(void) {
  unsigned int i;
  char found = 0;
  while (!found) {
    found = 1;
    for (i = 0; i < MAX_PORTS; i++) {
      if (!
STATIC_LIST_CHECK(__net.end_points, i))
        continue;
      struct sockaddr_in *addr =
(struct sockaddr_in*)
__net.end_points[i].addr;
      if (__net.next_port == addr-
>sin_port) {
        __net.next_port =
htons(ntohs(__net.next_port) + 1);
        found = 0;
        break;
      }
    }
  }
  in_port_t res = __net.next_port;
  __net.next_port =
htons(ntohs(__net.next_port) + 1);
  return res;
}
```

Formal Verification

```
static in_port_t
__get_unused_port(void) {
  unsigned int i;
  char found = 0;
  while (!found) {
    found = 1;
    for (i = 0; i < MAX_PORTS; i++) {
      if (!
STATIC_LIST_CHECK(__net.end_points, i))
        continue;
      struct sockaddr_in *addr =
(struct sockaddr_in*)
__net.end_points[i].addr;
      if (__net.next_port == addr-
>sin_port) {
        __net.next_port =
htons(ntohs(__net.next_port) + 1);
        found = 0;
        break;
      }
    }
  }
  in_port_t res = __net.next_port;
  __net.next_port =
htons(ntohs(__net.next_port) + 1);
  return res;
}
```

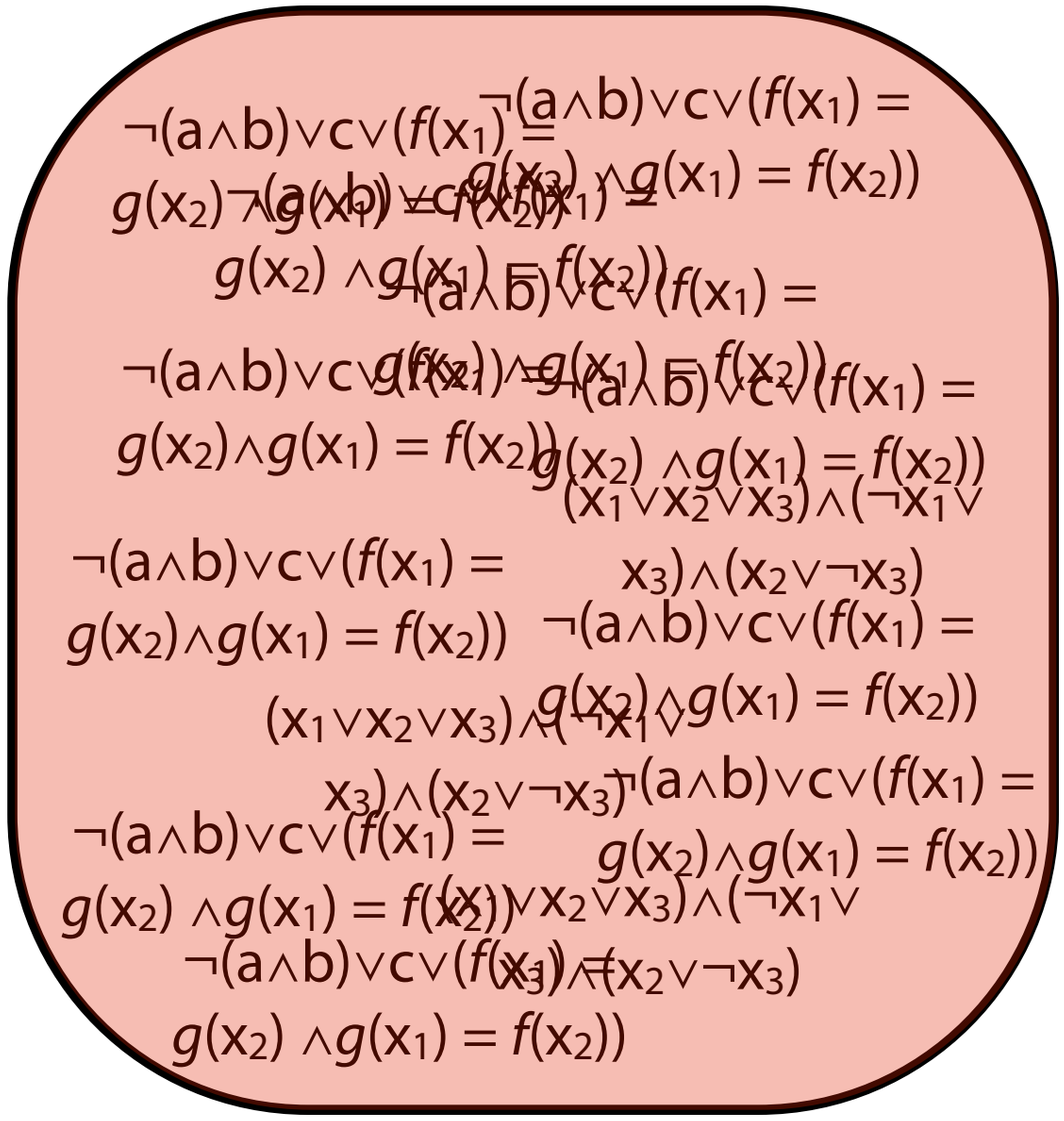
Model



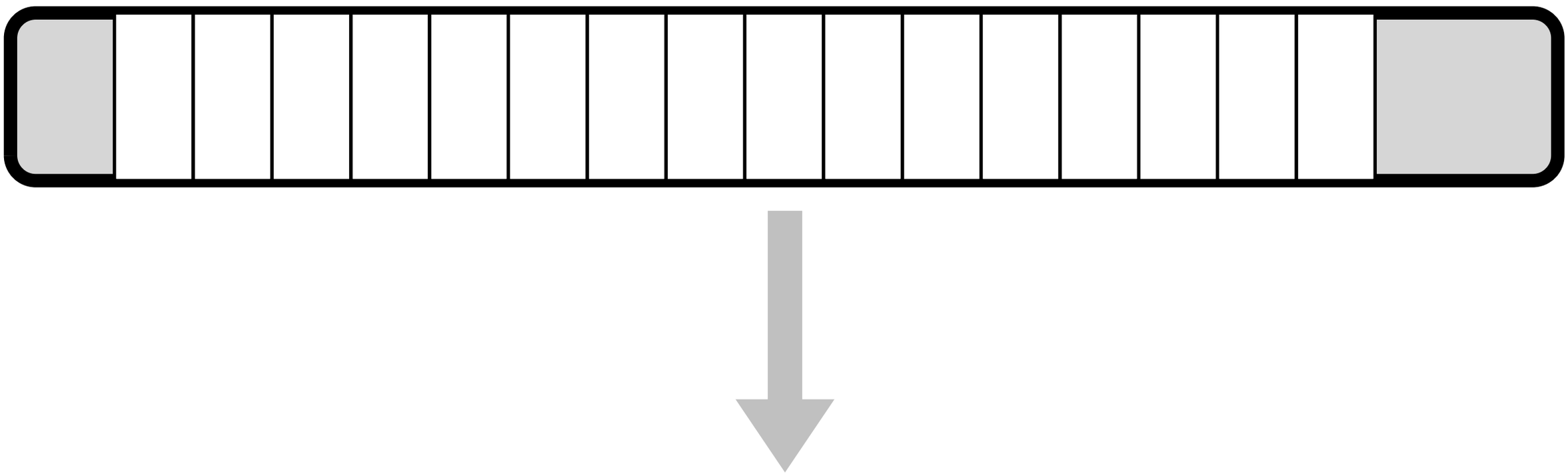
Formal Verification

```
static in_port_t
__get_unused_port(void) {
  unsigned int i;
  char found = 0;
  while (!found) {
    found = 1;
    for (i = 0; i < MAX_PORTS; i++) {
      if (!
STATIC_LIST_CHECK(__net.end_points, i))
        continue;
      struct sockaddr_in *addr =
(struct sockaddr_in*)
__net.end_points[i].addr;
      if (__net.next_port == addr-
>sin_port) {
        __net.next_port =
htons(ntohs(__net.next_port) + 1);
        found = 0;
        break;
      }
    }
  }
  in_port_t res = __net.next_port;
  __net.next_port =
htons(ntohs(__net.next_port) + 1);
  return res;
}
```

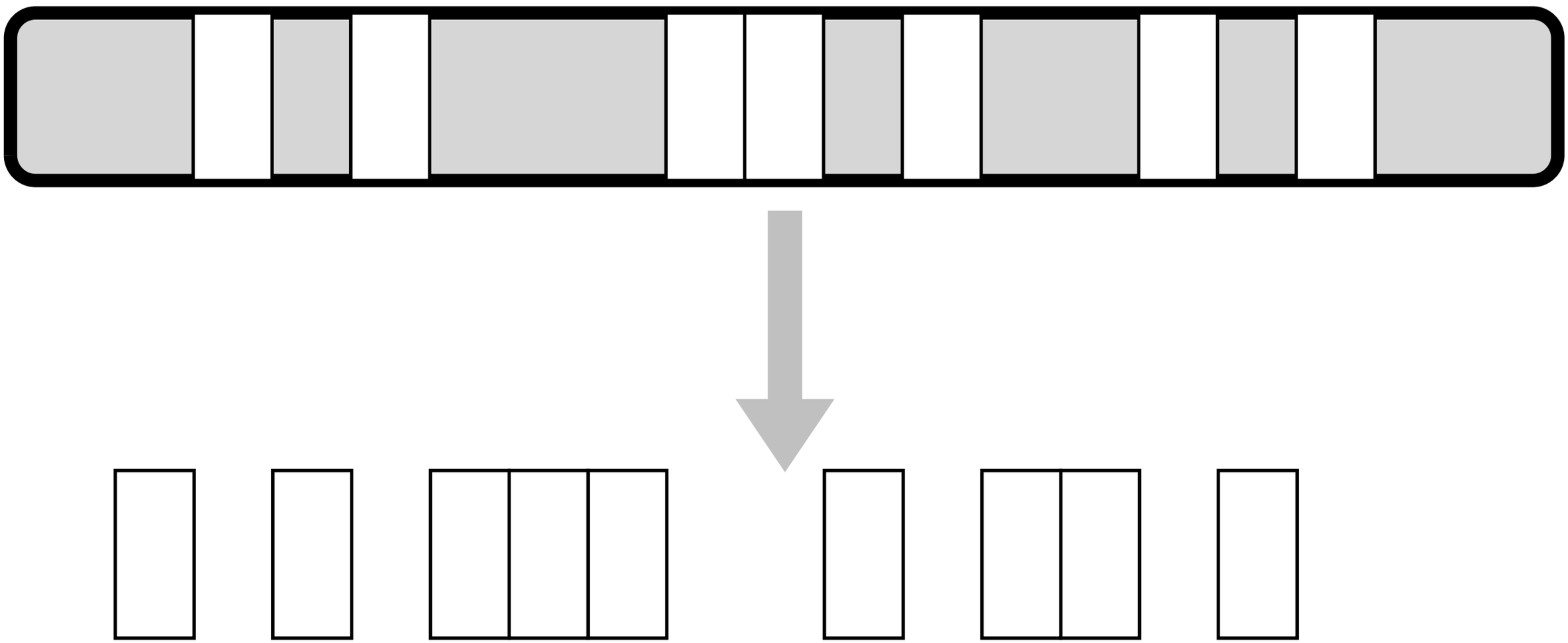
Model



Per-Path Execution



Per-Path Execution



States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

$n = \lambda$

States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

$n = \lambda$

$m = \lambda + 42$

States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

State:
$n = \lambda$
$m = \lambda + 42$

States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:      return 0;
5:    else
6:      BUG;
7: }
```

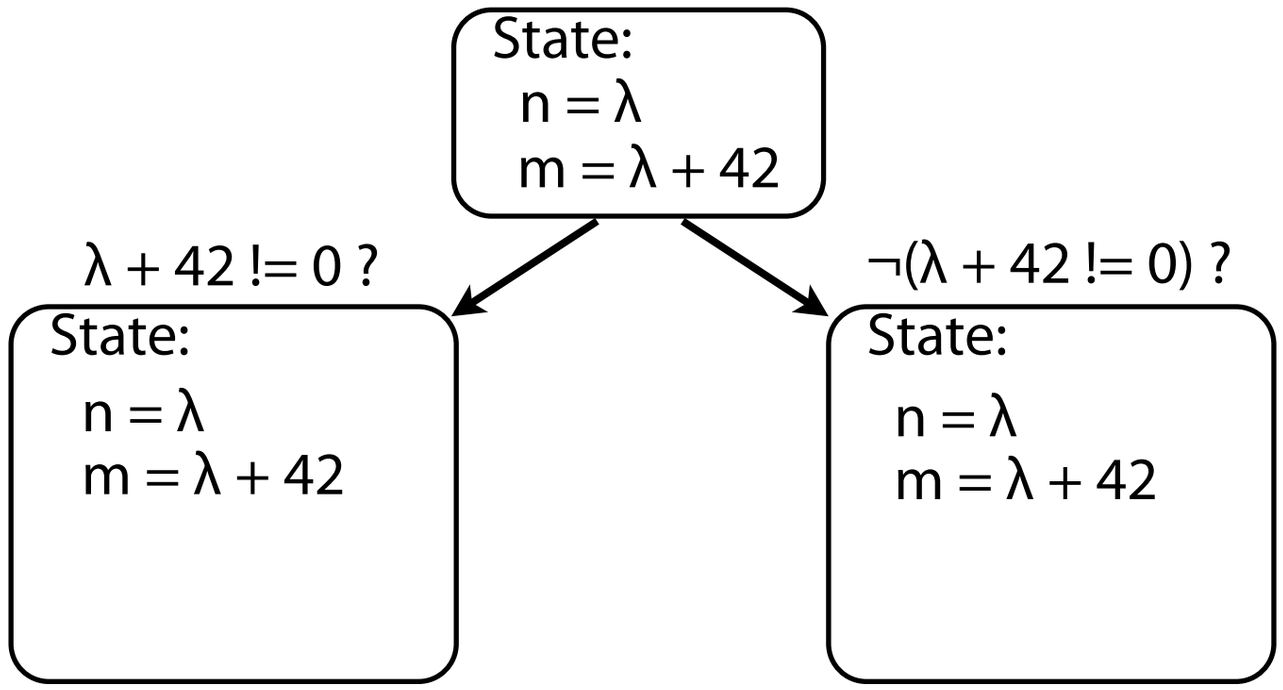
State:
n = λ
m = λ + 42

λ + 42 != 0 ?

¬(λ + 42 != 0) ?

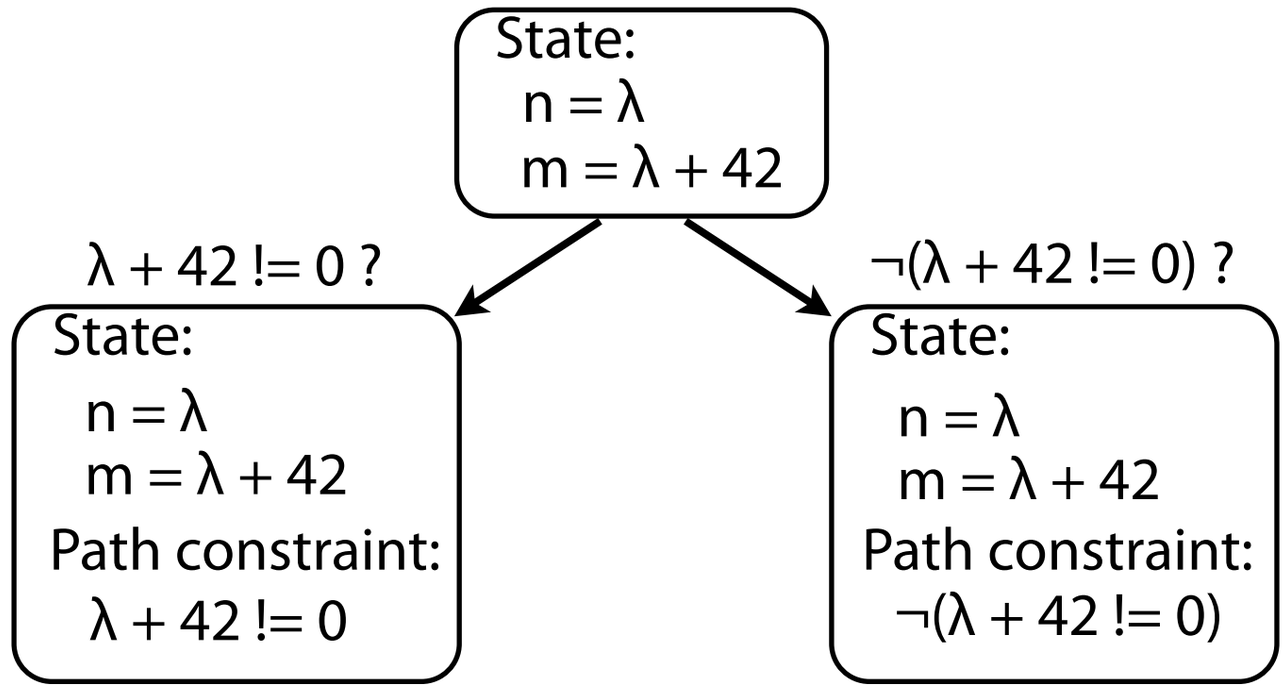
States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```



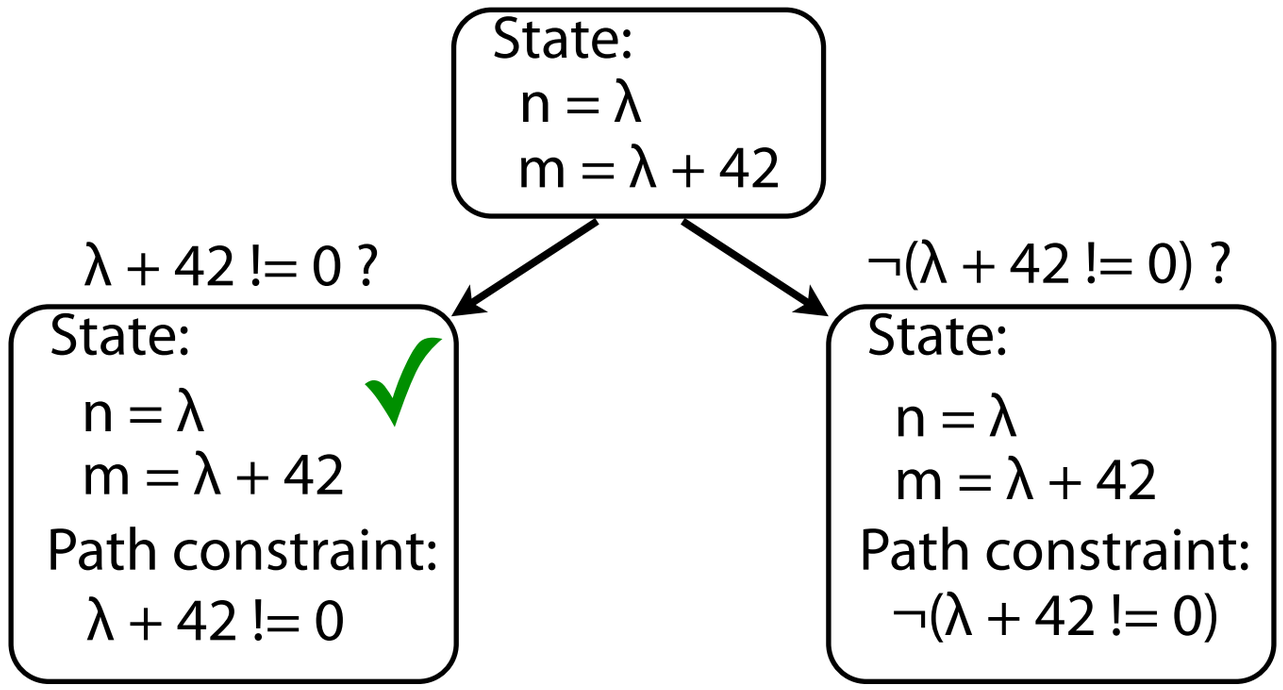
States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```



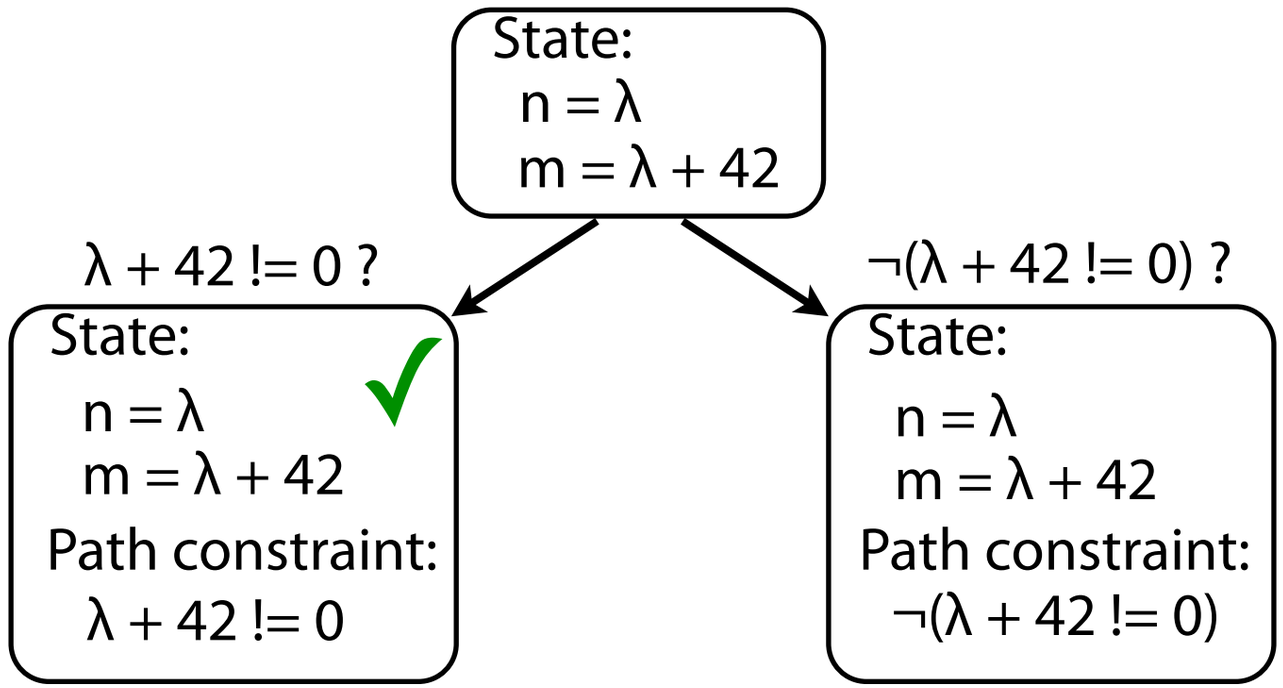
States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```



States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

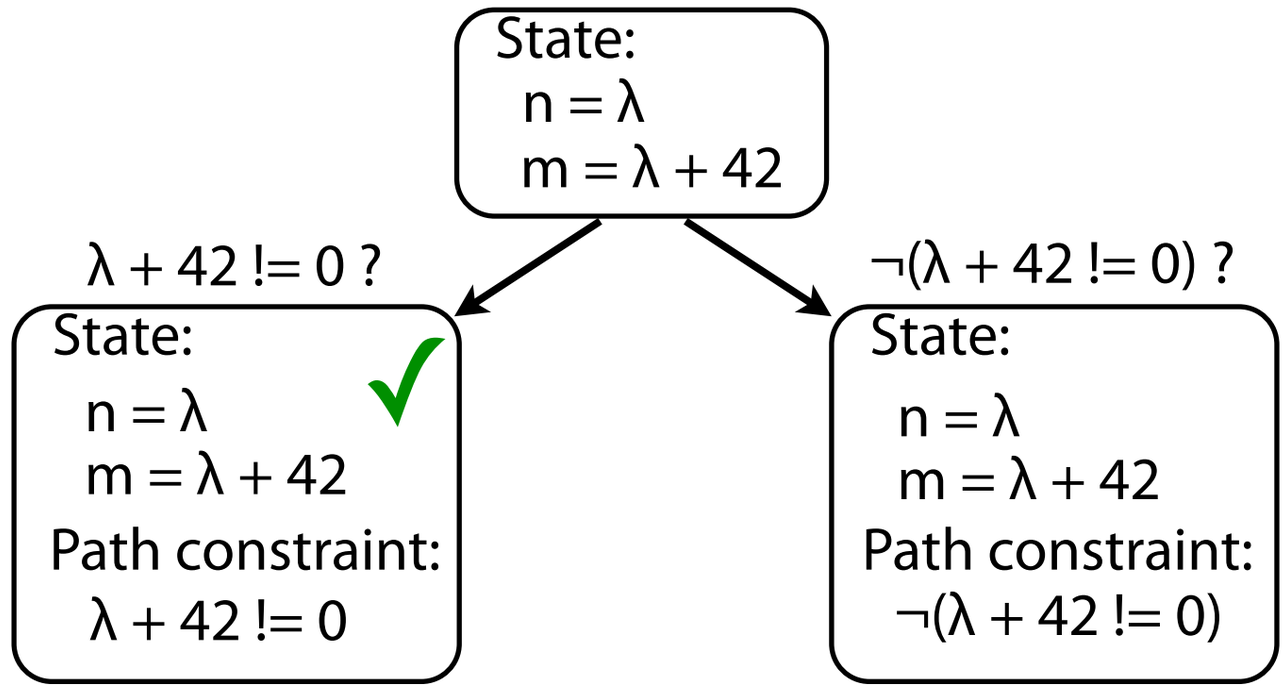


Solve for λ:

λ = 0

States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

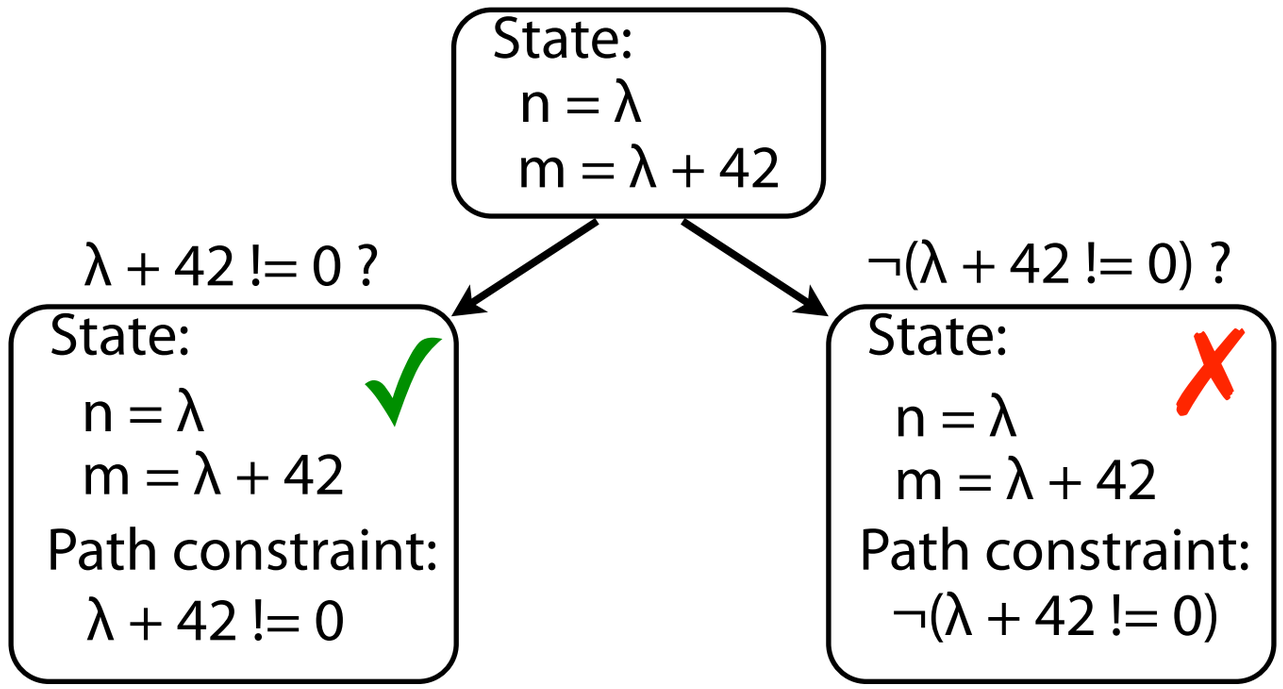


Solve for λ:

λ = 0

States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```

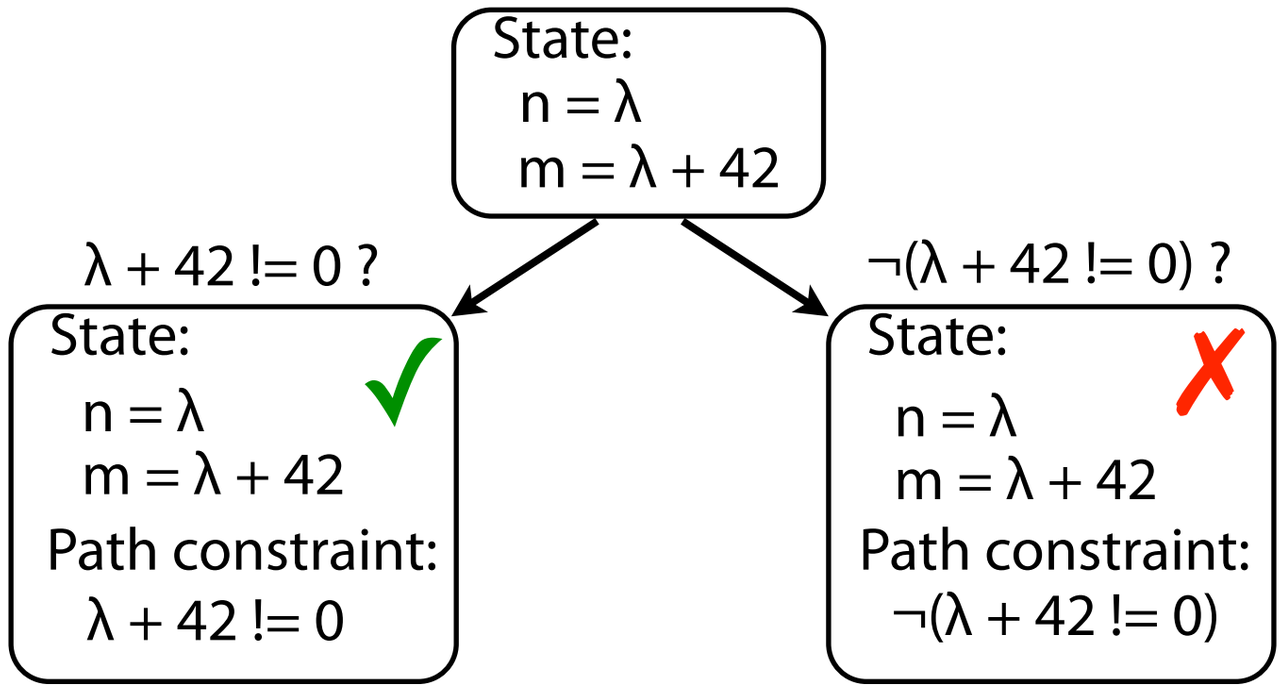


Solve for λ:

λ = 0

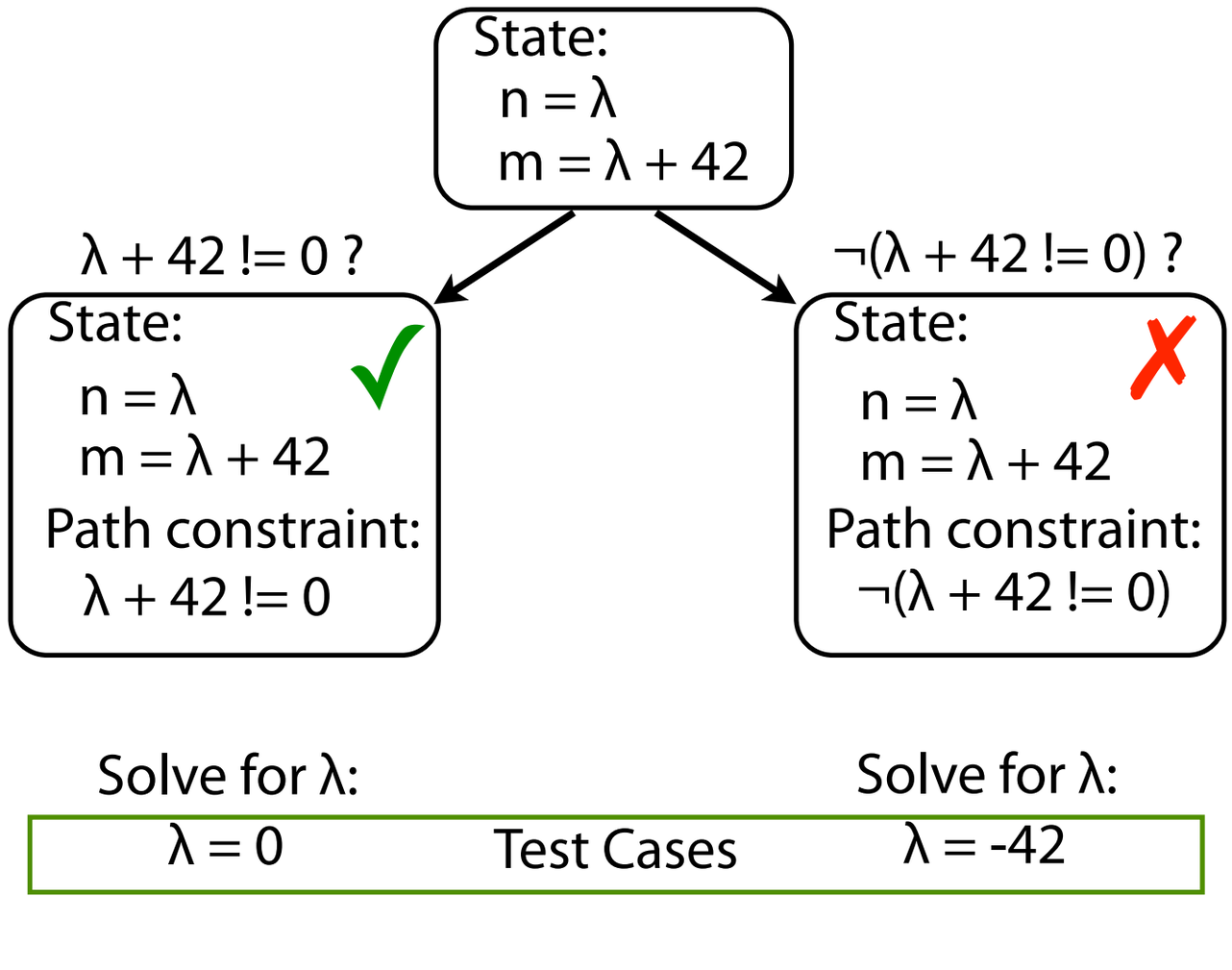
States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```



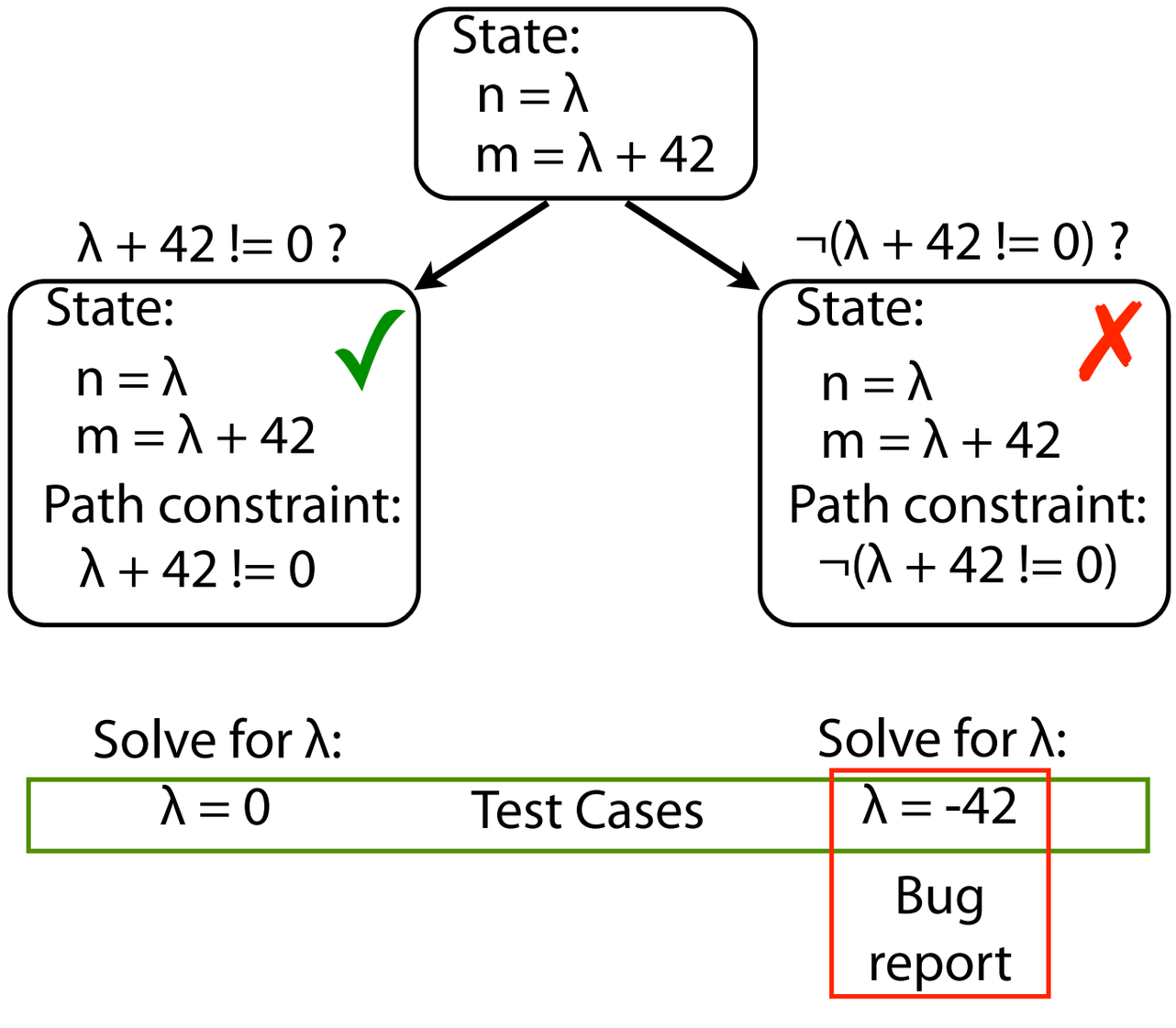
States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```



States & Path Constraints

```
1: int foo(int n) {
2:    int m = n + 42;
3:    if (m != 0)
4:       return 0;
5:    else
6:       BUG;
7: }
```



Path Explosion

- #paths exponential in program size
- CPU & memory bottleneck
- Main scalability issue

Outline

- Symbolic Execution
- **EFFIGY and SELECT**
- KLEE
- Final Remarks

Common Traits

- Common fundamental approach
- Interactive sessions
- In-house (ineffective) constraint solvers
- Small-size testing targets

EFFIGY

- Interactive tool from IBM
  - *Tracing*
  - *Breakpoints*
  - *State saving*
- PL/I programs testing
- Assume & Assert functionality

SELECT

- Research tool from Stanford
- Uses Lisp dialect for both program language & expression repr.

Outline

- Symbolic Execution
- EFFIGY and SELECT
- **KLEE**
- Final Remarks

KLEE

- Real-world software testing
- Automated test-case generation
  - *Statement coverage*
  - *Bug finding*
- Modern off-the-shelf constraint solver

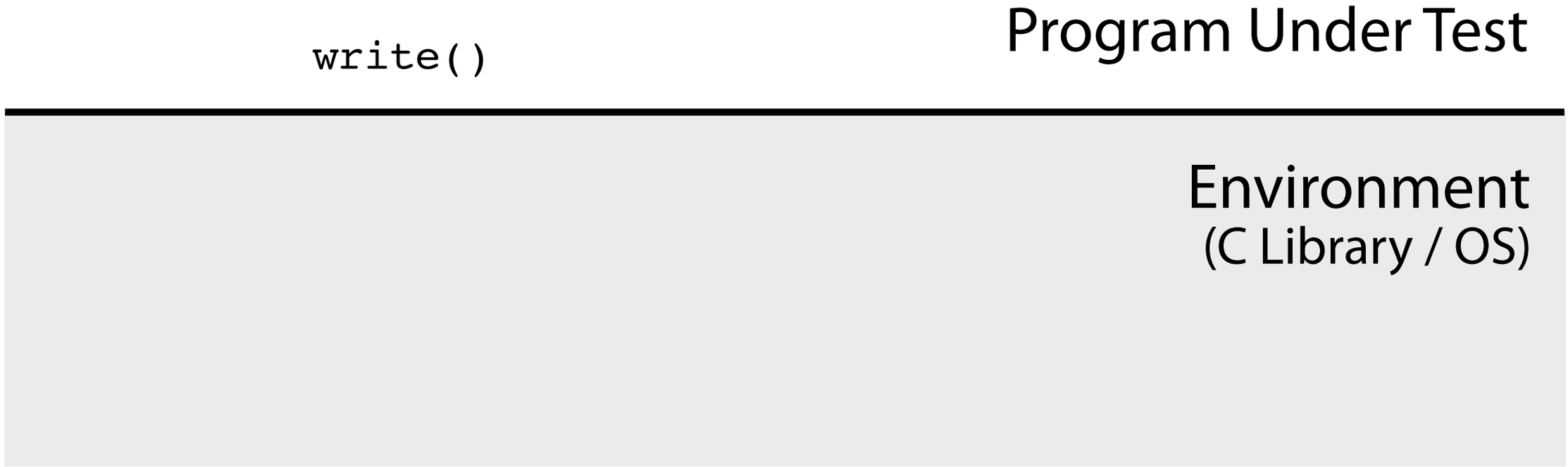
Constraint Solving

- STP: Fast solver for program verification
- Expression simplifications
- Constraint solution caching

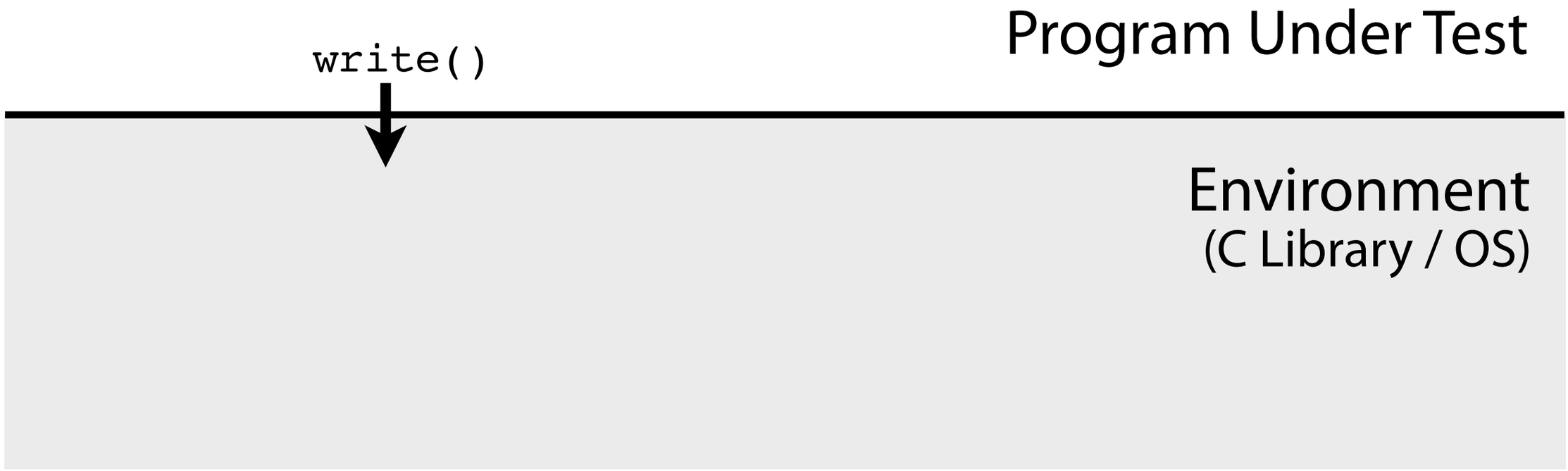
Search Heuristics

- Path explosion prevents exhaustive search
- Random tree walking strategy
- Coverage-oriented strategy

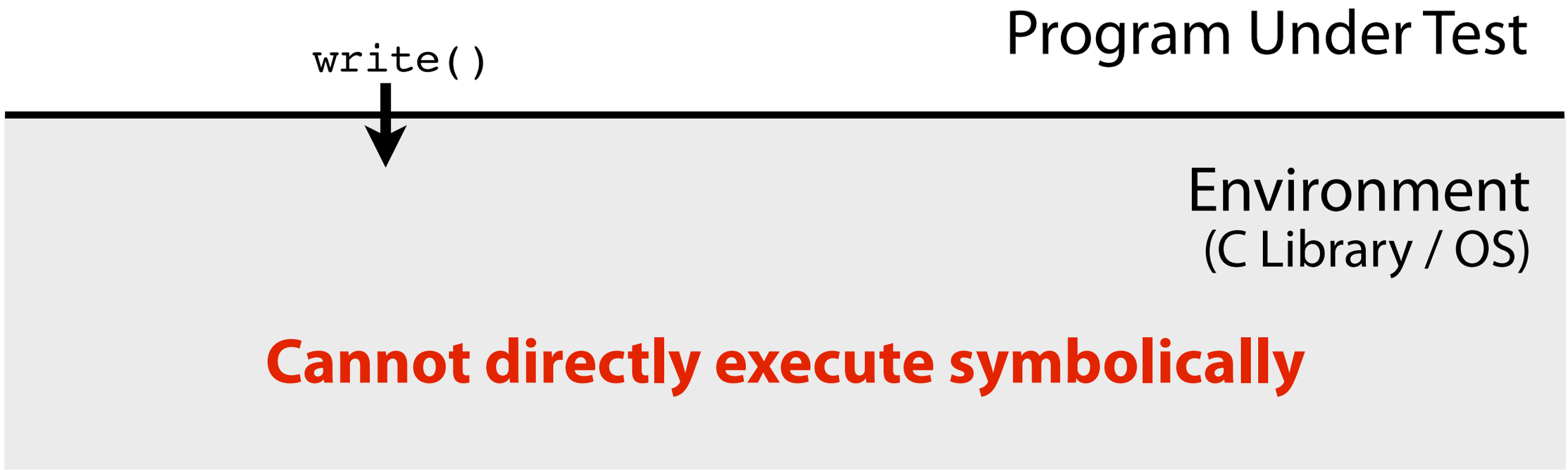
Environment Model



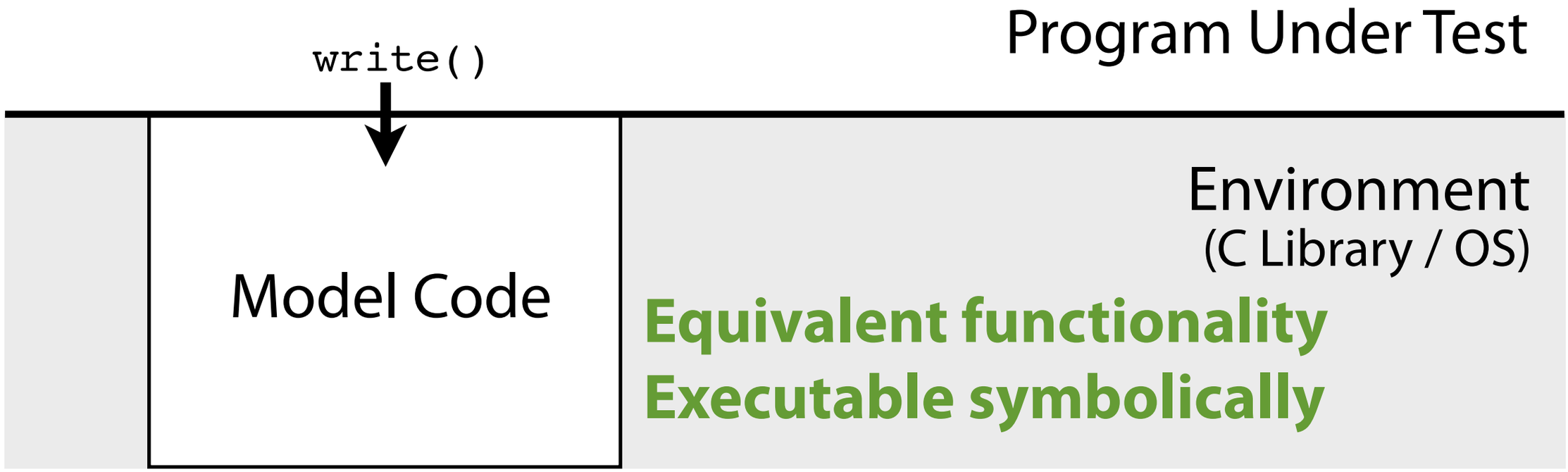
Environment Model



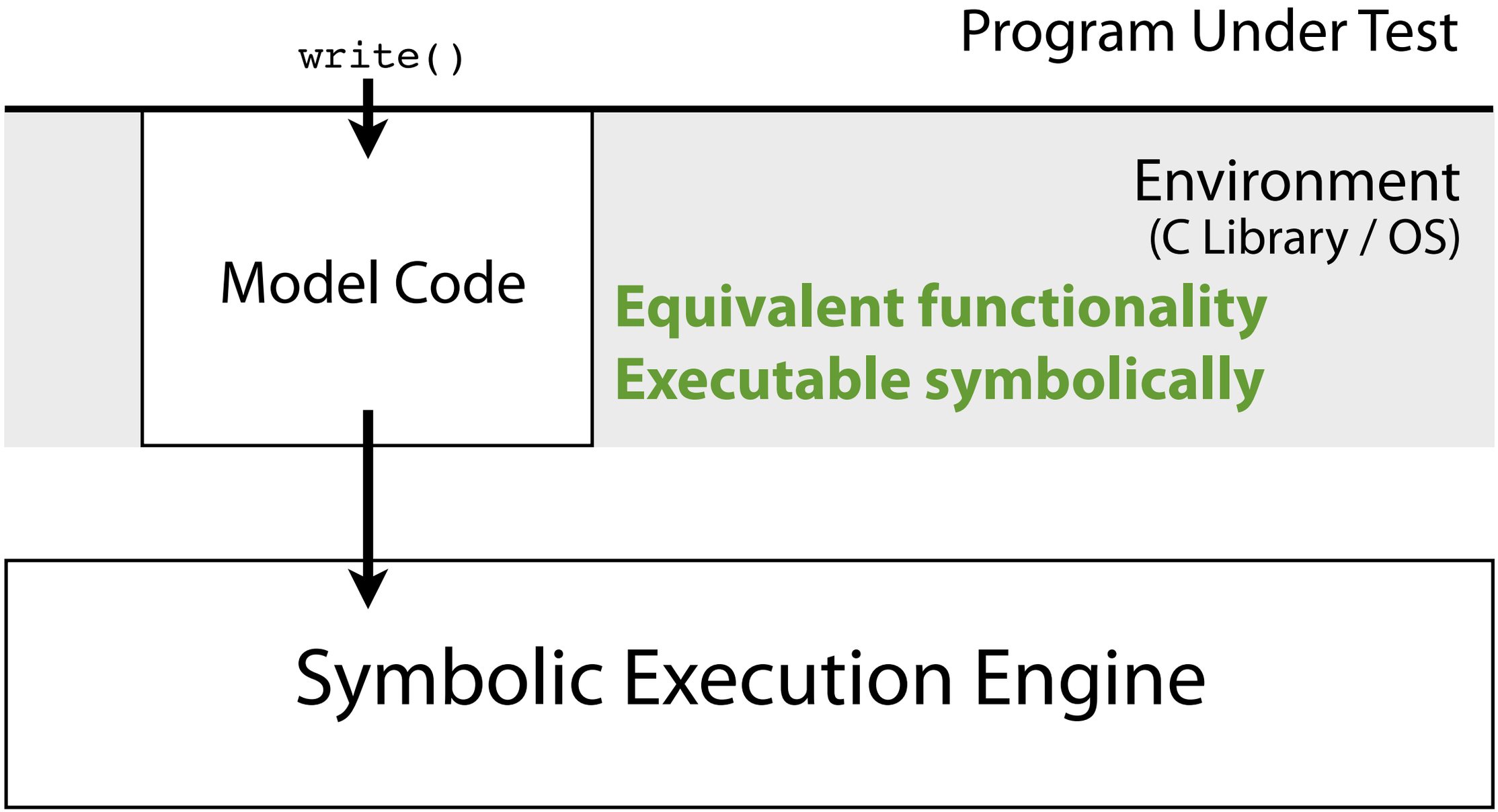
Environment Model



Environment Model



Environment Model



Real-World Software Testing

- Tested GNU Coreutils suite of system utils (89 programs, 2 to 6 KLOC)
  - Obtained 90.9% line coverage on average (min. 60%, max. 100% on 16 programs)
  - Found 10 new bugs
- Tested equivalence between GNU Coreutils and Busybox, found inconsistencies

Outline

- Symbolic Execution
- EFFIGY and SELECT
- KLEE
- **Final Remarks**

Final Remarks

- Testing = verifying against **human intent**?
- EFFIGY & SELECT: **debugging** assistants
- KLEE: automated **test case generation**

Final Remarks

- Testing = verifying against **human intent**?
- EFFIGY & SELECT: **debugging** assistants
- KLEE: automated **test case generation**

How much/What can we automate?