Verifying the Absence of Buffer Overflows: Software Model Checking with Proof Templates

Tom Hart[a], Kelvin Ku[a], Marsha Chechik[a], David Lie[a], Arie Gurfinkel[b]

[a]University of Toronto [b]Carnegie Mellon University

Software Vulnerabilities

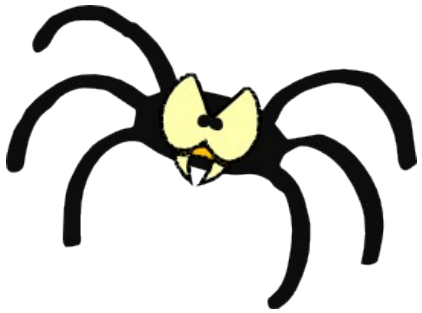
Web sites threatened by Samy worm

Munir Kotadia, ZDNet Australia

17 October 2005 04:37 PM

Tags: attack, cross site, xss, mx logic, scripting, vulnerability, worm, vulnerable

The newly discovered Samy worm is the first to exploit a cross site scripting vulnerability, prompting security experts to fear the technique could be used to open a new front in the war against malware.



Slammed!

An inside view of the worm that crashed the Internet in 15 minutes.

By Paul Boutin

"Gah!" Owen Maresh almost choked whe
popped up on his panel of screens just
Saturday, January 25. Sitting inside Aka
Operations Control Center, the comman
high-speed servers stationed around th

- ✗ Bugs in software
- ✗ Compromise system integrity:
  - ✗ rootkits
  - ✗ worms
- ✗ Patching is expensive
- ✗ Economic cost estimated to be billions per year

Many Sources of Vulnerabilities

Command Injection

Buffer Overflows

Cross-site Scripting

SQL Injection

Race Conditions

Format String Errors

API Errors

Many Sources of Vulnerabilities

Command Injection

Buffer Overflows

SQL Injection

Cross-site Scripting

Many Sources of Vulnerabilities

Command Injection

Buffer Overflows

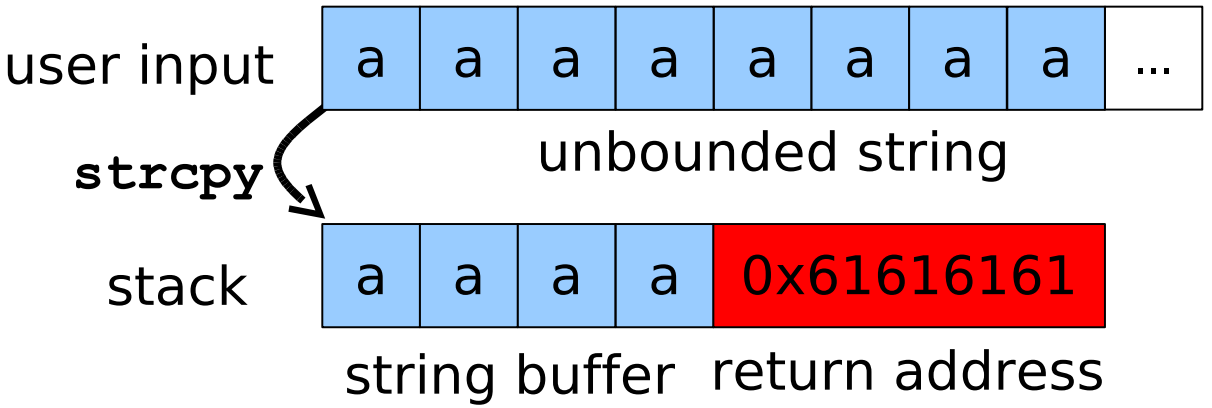
→Largest vulnerability class in C programs

SQL Injection

Cross-site Scripting

Buffer Overflows

A user-controlled operation which may access a buffer (array) beyond its bounds.



Approaches to Dealing with Buffer Overflows

- Runtime protection
  - incomplete and/or expensive
- Testing
  - finds buffer overflows well (MOBB, MOKB....)
  - does not prove absence
- Lightweight static analysis (Lint, Flawfinder,...)
  - lots of false alarms
  - may also miss errors
- Verification
  - proves absence of errors

Approaches to Dealing with Buffer Overflows

- Runtime protection
  - incomplete and/or expensive
- Testing
  - finds buffer overflows well (MOBB, MOKB....)
  - does not prove absence
- Lightweight static analysis (Lint, Flawfinder,...)
  - lots of false alarms
  - may also miss errors
- Verification
  - proves absence of errors

The Case for Verification

- Buffer overflow found in Madwifi (CVE-2006-6332)

```
for (i = 0; i < ielen && bufsize > 2; i++)
    p += sprintf(p, "%02x", ie[i]);
```

- No bounds checking on **p**

The Case for Verification

- Buffer overflow found in Madwifi (CVE-2006-6332)

```
if (bufsize < ielen)
    return 0;
for (i = 0; i < ielen && bufsize > 2; i++)
    p += sprintf(p, "%02x", ie[i]);
```

- No bounds checking on **p**
- A fix was committed

The Case for Verification

- Buffer overflow found in Madwifi (CVE-2006-6332)

```
if (bufsize < ielen)
    return 0;
for (i = 0; i < ielen && bufsize > 2; i++) {
    p += sprintf(p, "%02x", ie[i]);
    bufsize -= 2;
}
```

- No bounds checking on **p**
- A fix was committed
- The fix was incorrect

Limits of Verification

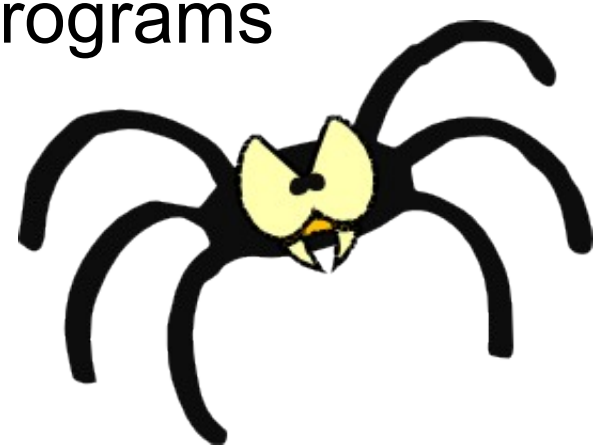
- **Goal:** prove the absence of buffer overflows

Limits of Verification

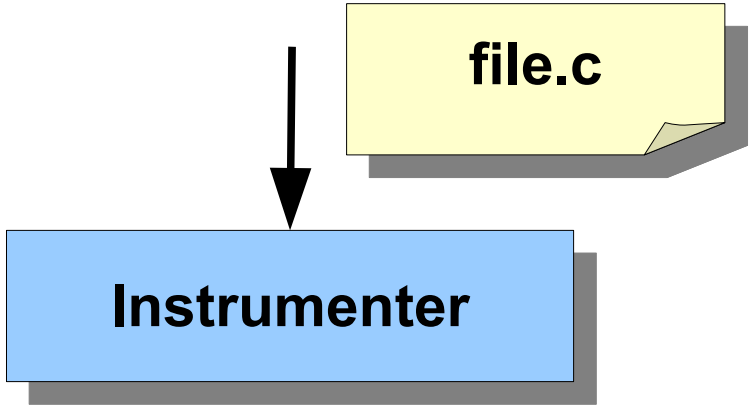
- **Goal:** prove the absence of buffer overflows
- Impossible *in general* (halting problem)

Limits of Verification

- **Goal:** prove the absence of buffer overflows
- Impossible *in general* (halting problem)
- Try to make verification succeed *often*
  - Design analyses which work well for common types of programs

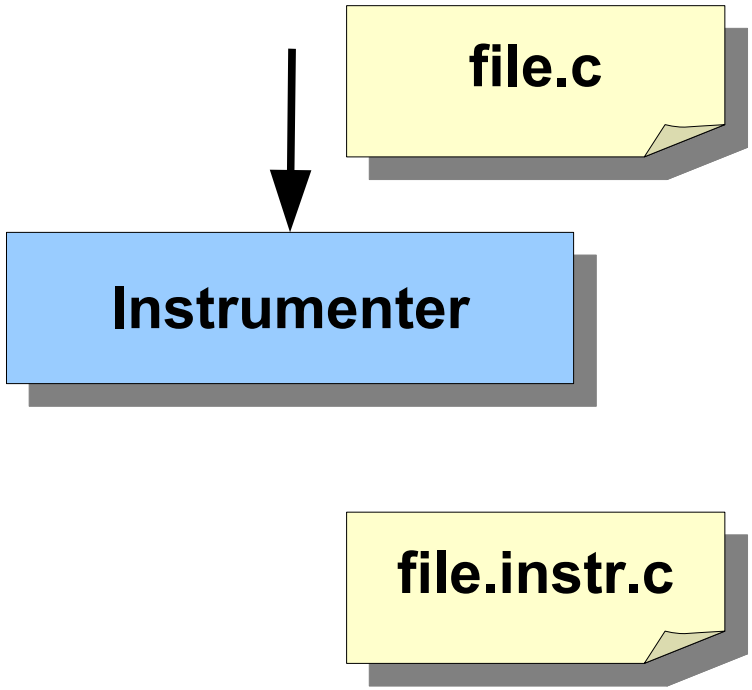


Our Approach to Verification



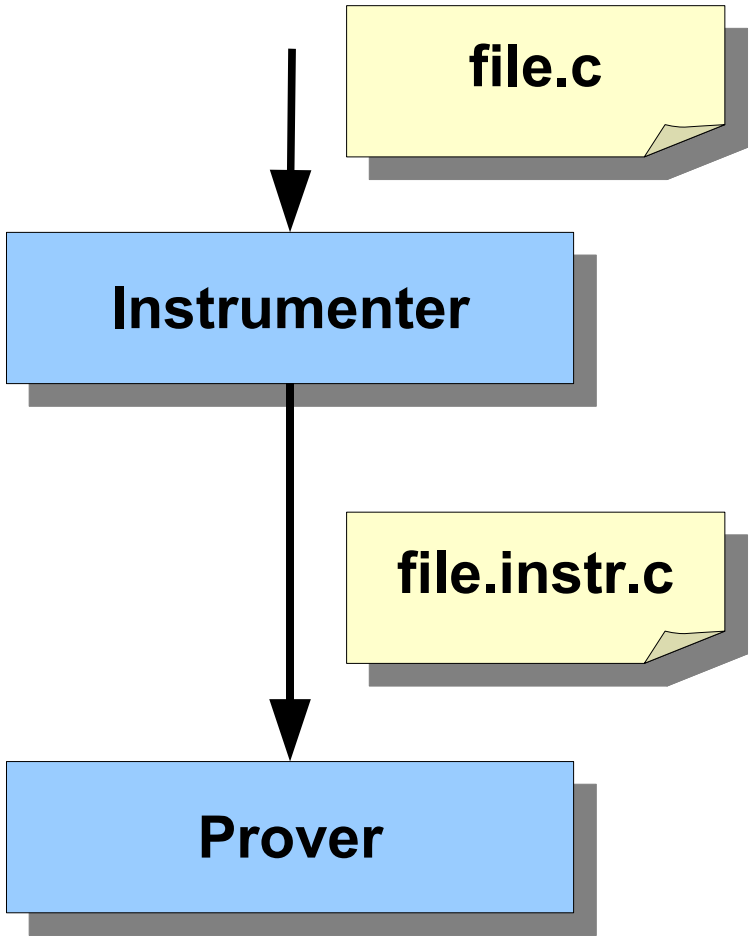

```
char A [100];
...

A[i] = x;
```

Our Approach to Verification



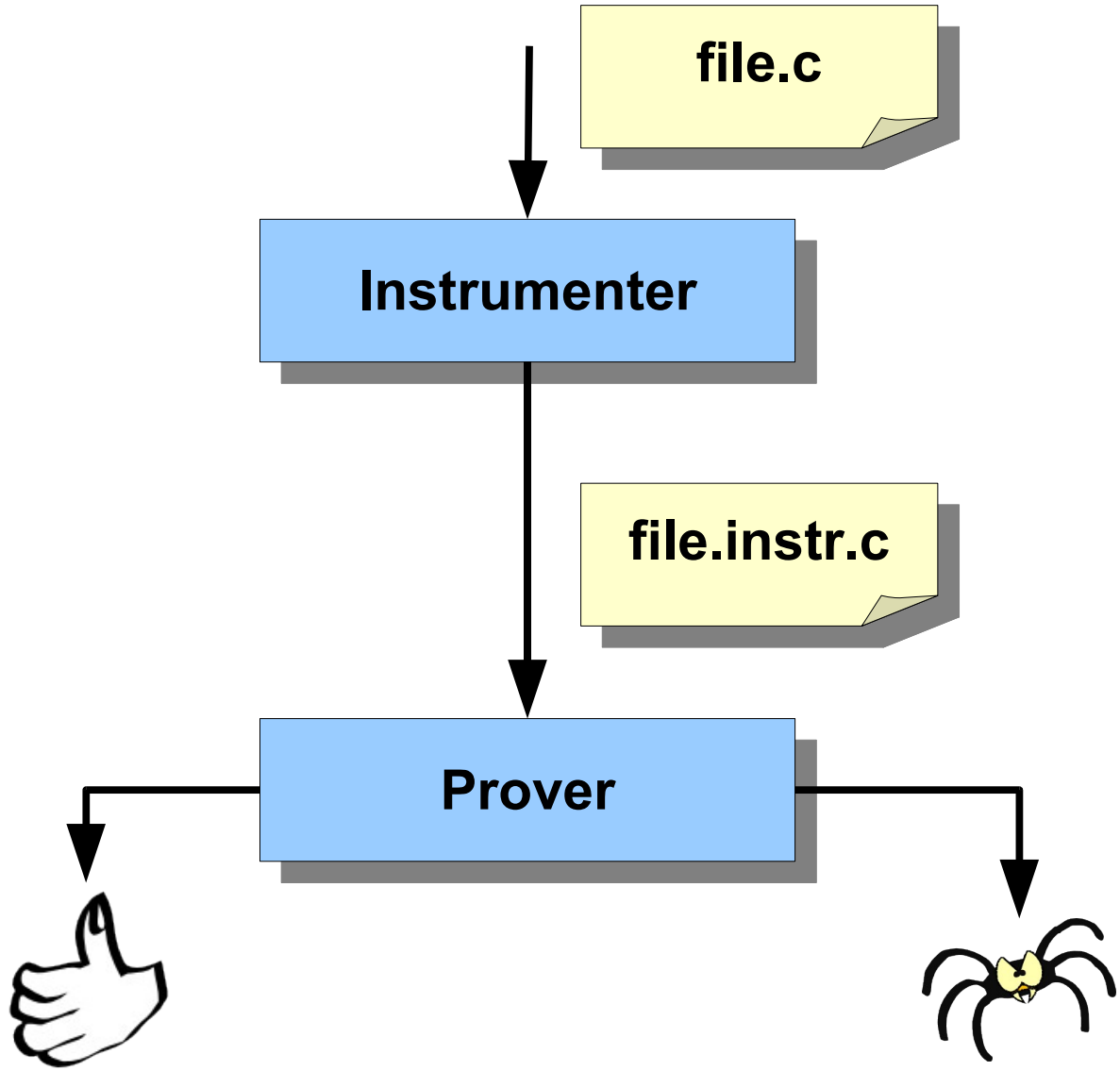

```
char A [100];
...
assert (0 <= i < 100);
A[i] = x;
```

Our Approach to Verification




```
char A [100];
...
assert (0 <= i < 100);
A[i] = x;
```

Our Approach to Verification

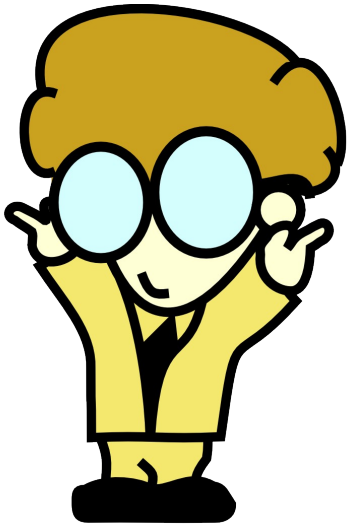



```
char A [100];
...
assert (0 <= i < 100);
A[i] = x;
```

Invoke prover once for *each* assertion

Software Model Checking for Buffer Overflows?

- Technical advantages:
  - ✔ path-sensitive
  - ✔ iterative --- removes *false alarms*
  - ✔ produces both *proofs* and *counterexamples* (bugs)
- Successful track record:
  - ✔ powers Microsoft's Static Driver Verifier



Preliminary Evaluation

- Verisec suite for benchmarking (ASE'07)
  - Simplified testcases based on vulnerabilities in 12 open source applications
  - Patched and unpatched versions
- Evaluate software model checking for:
  - Finding buffer overflows
  - Proving their absence

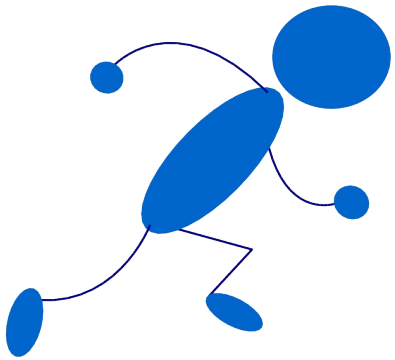
Preliminary Evaluation

- Verisec suite for benchmarking (ASE'07)
  - Simplified testcases based on vulnerabilities in 12 open source applications
  - Patched and unpatched versions
- Evaluate software model checking for:
  - ✗ Finding buffer overflows
  - ✗ Proving their absence

- **Ineffective** for both tasks due to ***loop unrolling***

Our Goal

- Enable software model checking to prove the absence of buffer overflows *without* loop unrolling
- Methodology:
  - Take advantage of knowledge of *common loop structures*
  - Guide tool towards succinct proofs

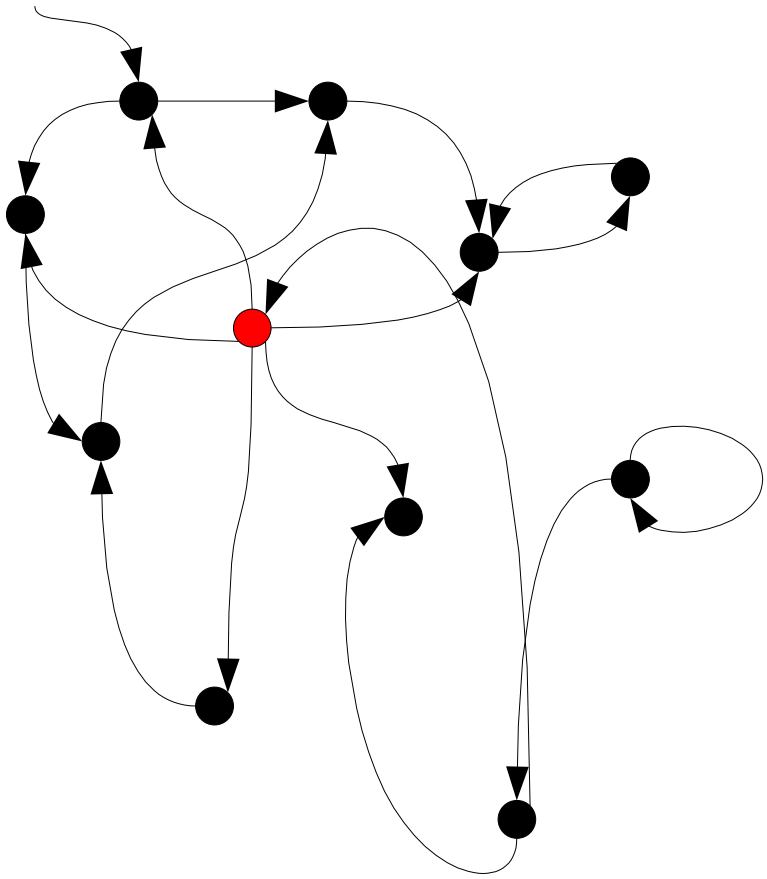


Outline

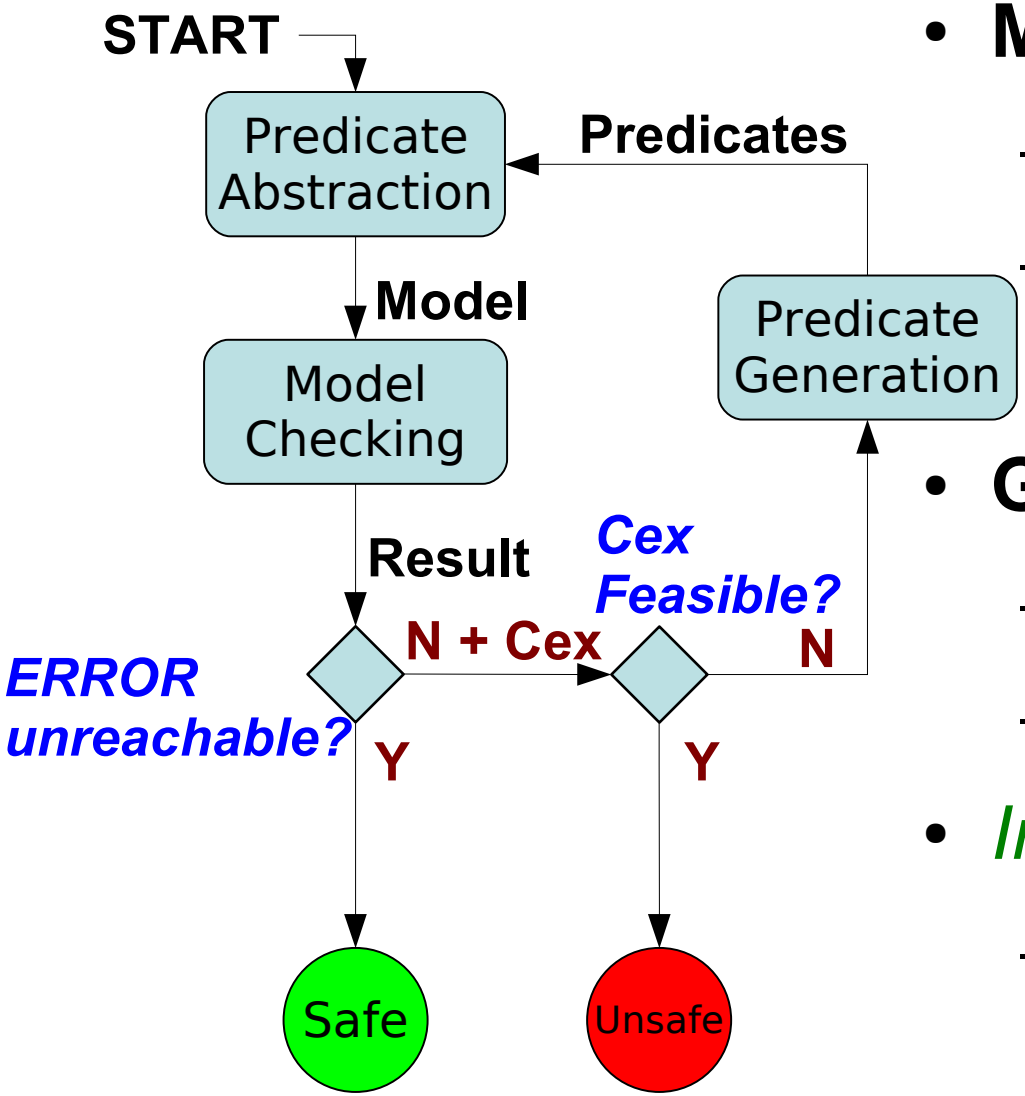
1. Introduction
   i. Buffer overflows
   ii. Need for verification
   iii. Preliminary Evaluation
2. Background
   i. Sofware Model Checking with CEGAR
   ii. Why loop unrolling happens
3. Solution: proof templates
4. Tool support and results
5. Conclusions and Future Work

Model Checking for Programs

- Model program as a *finite state machine*
  - one node per program state
  - *assertion failure* represented by a set of nodes (**red**)
- Process:
  - sophisticated graph search
  - checks if any assertion failure node is *reachable* from any *initial state*



CEGAR Software Model Checking



- **Model**:
  - control state: CFG
  - data state: *predicates*
    - ie. "*x* >= *y*"
- **Goal:** find the *right abstraction*
  - to prove/disprove safety
  - be tractable
- *Incremental* search for predicates
  - guided by *infeasible counterexamples* (cexes)

Example

```
1. i = 0;
2. while (i != 10) {
3.    assert (i < 10);
4.
5.    i = i + 1;
6. }
7.
```

Can assertion fail?

Example

```
1. i = 0;
2. while (i != 10) {
3.    assert (i < 10);
4.
5.    i = i + 1;
6. }
7.
```

Can assertion fail?

```
1. i = 0;
2. if (! (i == 10)) {
3.    if (! (i < 10))
4.       /* ERROR */ ;
5.    i = i + 1;
6.    goto 2; }
7. /* END */ ;
```

Is line 4 reachable?

Simplify

Example

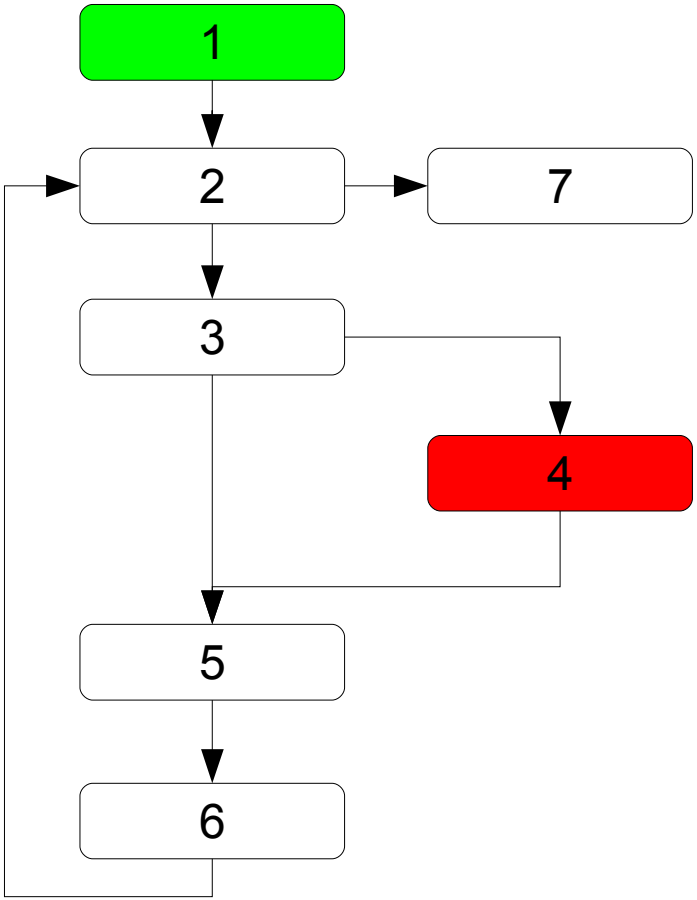
```
1. i = 0;
2. if (! (i == 10)) {
3.    if (! (i < 10))
4.       /* ERROR */ ;
5.    i = i + 1;
6.    goto 2; }
7. /* END */ ;
```

Simplify

Example

Predicate Set: {}

```
1. i = 0;
2. if (! (i == 10)) {
3.    if (! (i < 10))
4.       /* ERROR */ ;
5.    i = i + 1;
6.    goto 2; }
7. /* END */ ;
```

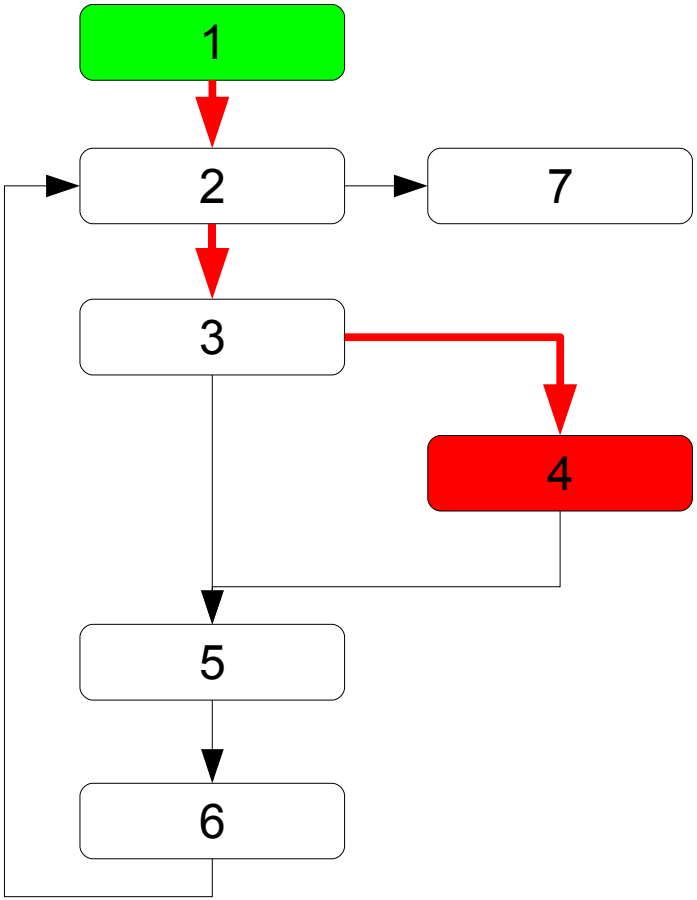


Simplify
Abstract

Example

Predicate Set: {}

```
1. i = 0;
2. if (! (i == 10)) {
3.    if (! (i < 10))
4.       /* ERROR */ ;
5.    i = i + 1;
6.    goto 2; }
7. /* END */ ;
```

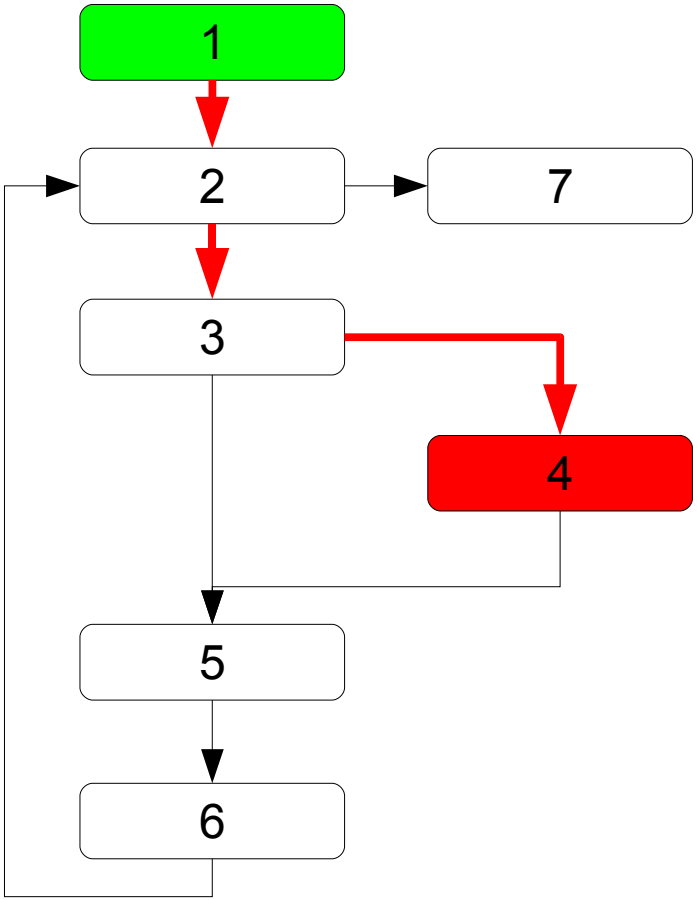


Simplify
Abstract → Check Model

Example

Predicate Set: {}

```
1. i = 0;
2. if (! (i == 10)) {
3.   if (! (i < 10))
4.     /* ERROR */ ;
5.   i = i + 1;
6.   goto 2; }
7. /* END */ ;
```

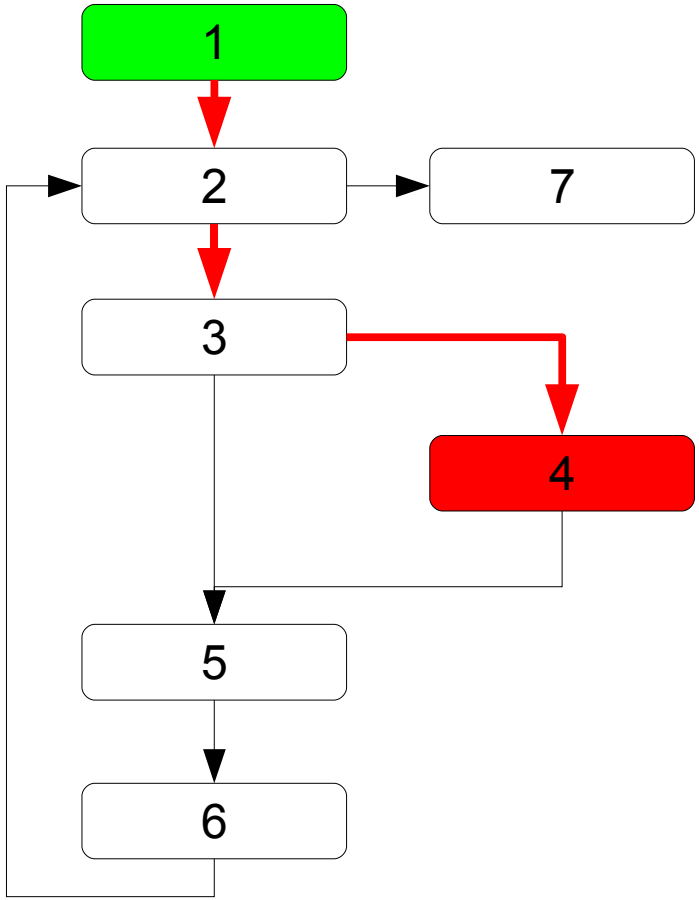


Simplify
Abstract → Check Model → Cex feasible?

Example

Predicate Set: { i < 10 }

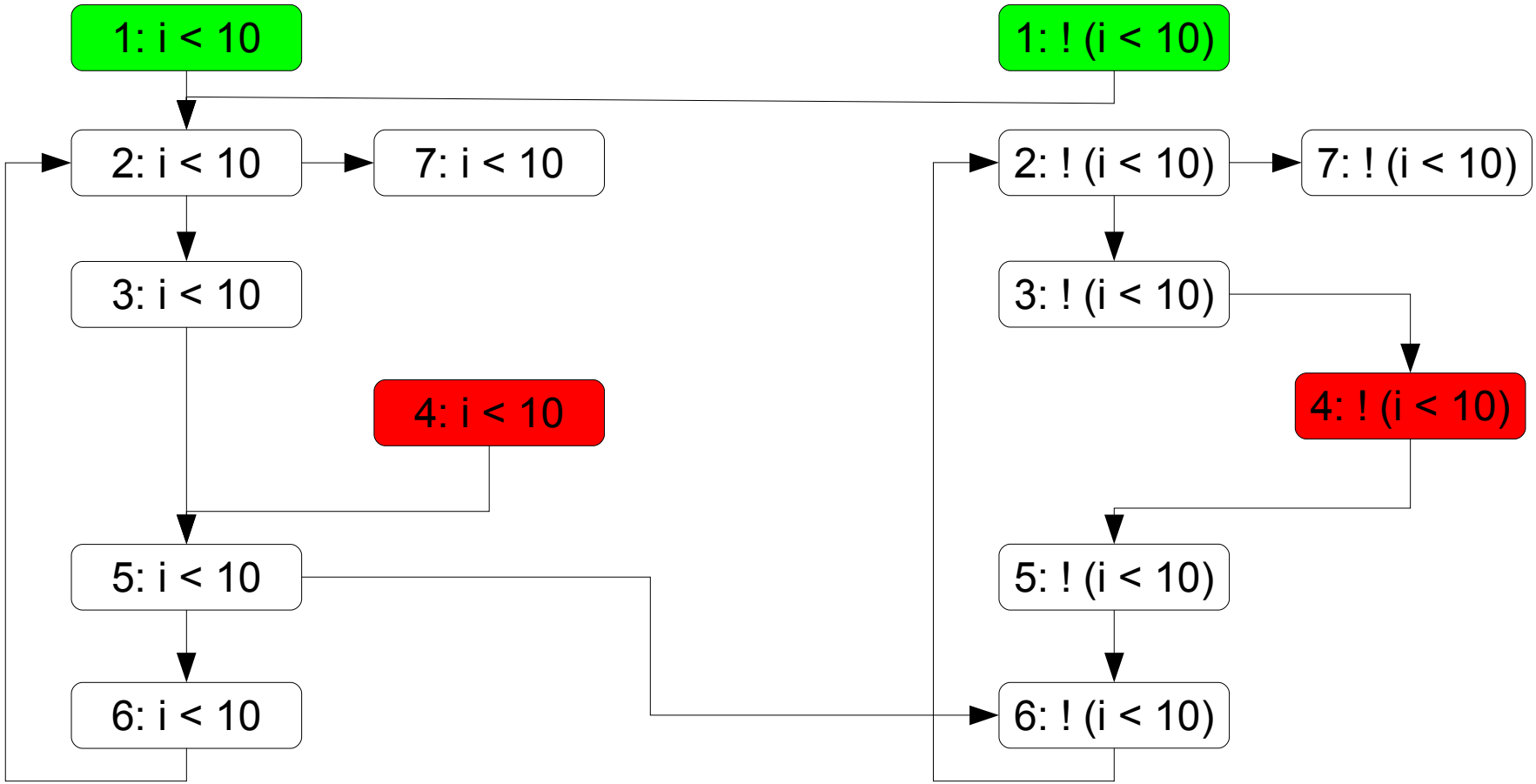
```
1. i = 0;
2. if (! (i == 10)) {
3.   if (   (i < 10))
4.     /* ERROR */ ;
5.   i = i + 1;
6.   goto 2; }
7. /* END */ ;
```



Simplify
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Example

Predicate Set: { i < 10 }



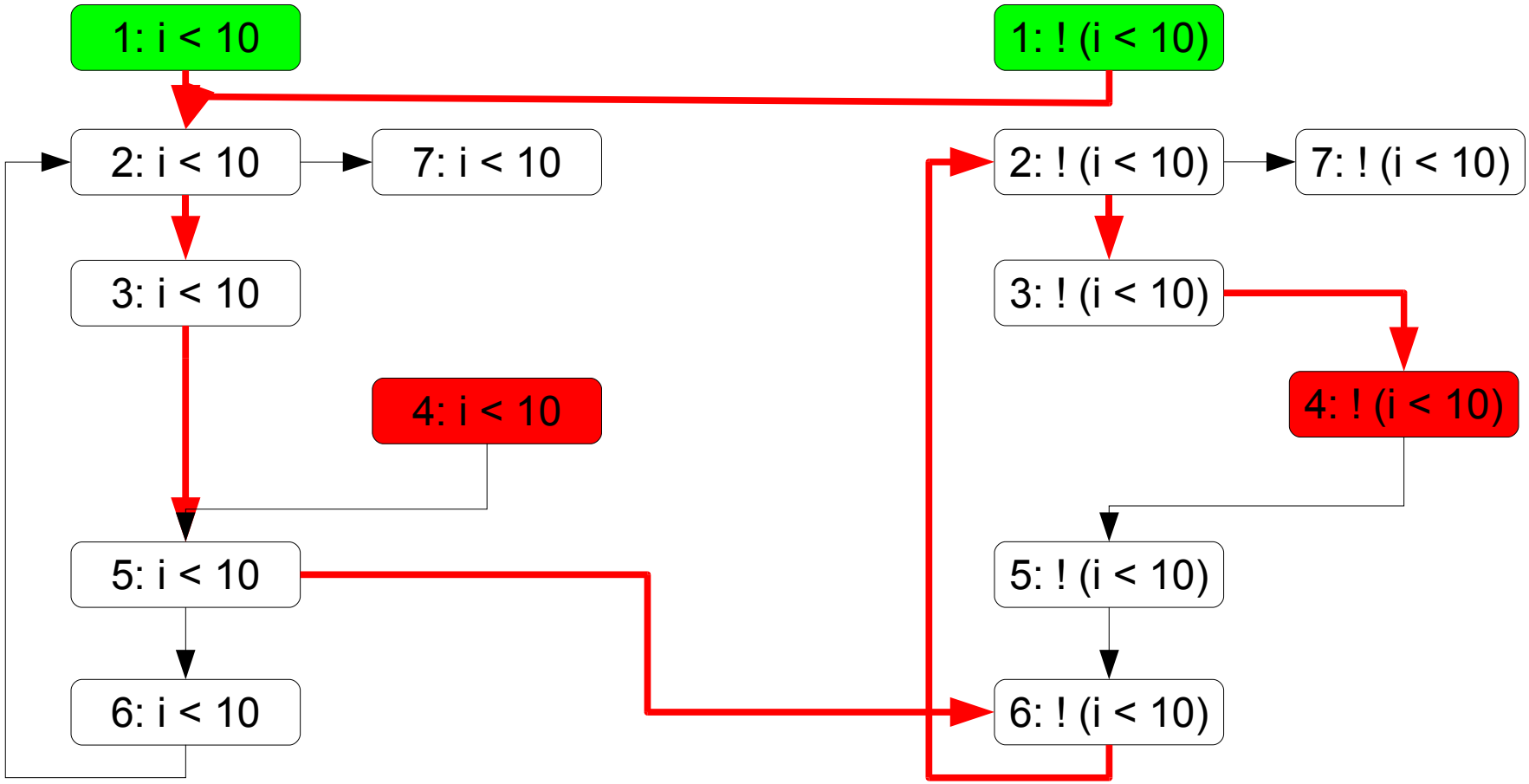
Simplify

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Abstract

Example

Predicate Set: { i < 10 }



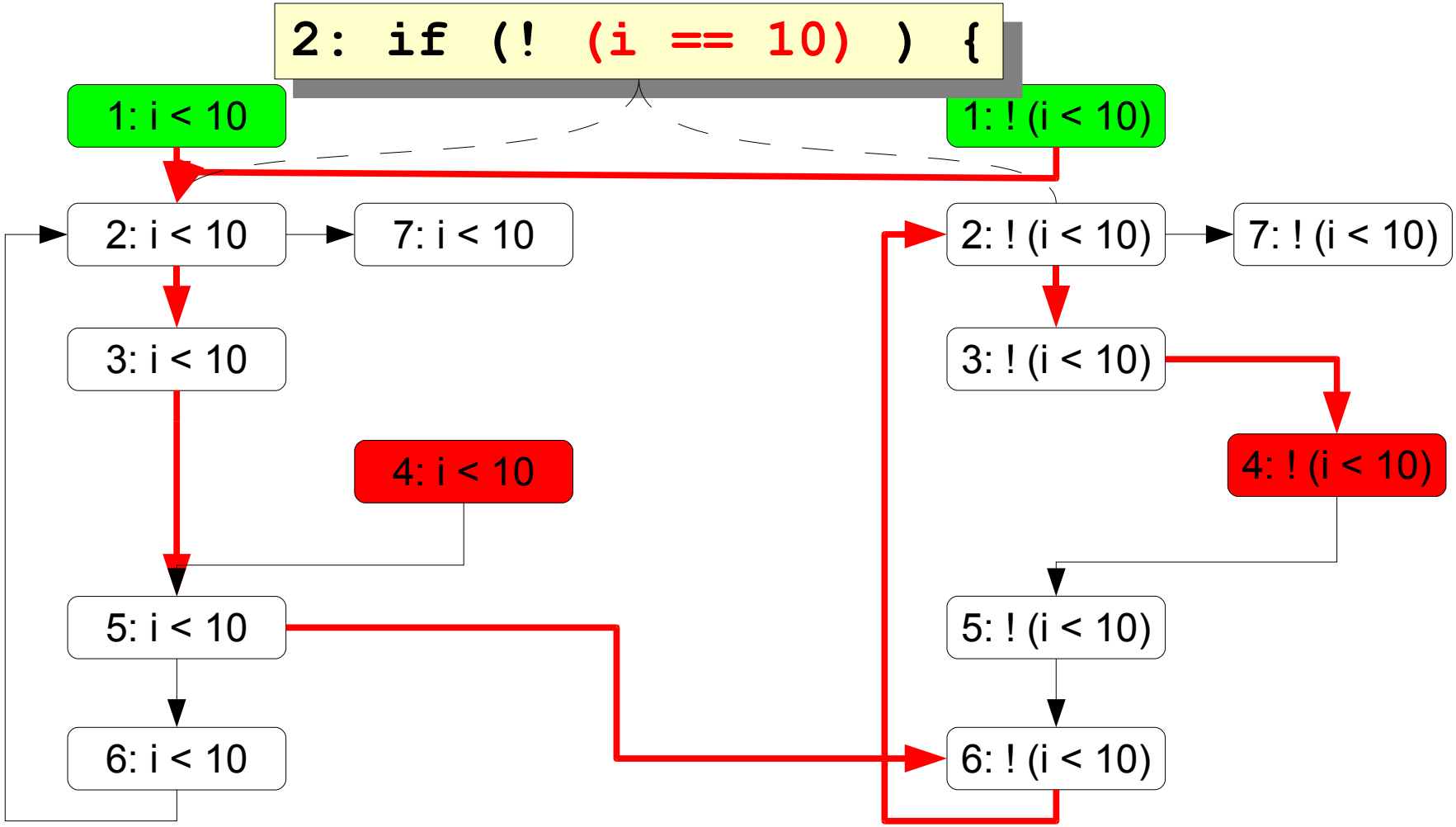
Simplify

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Abstract → Check Model

Example

Predicate Set: { i < 10, i == 10 }



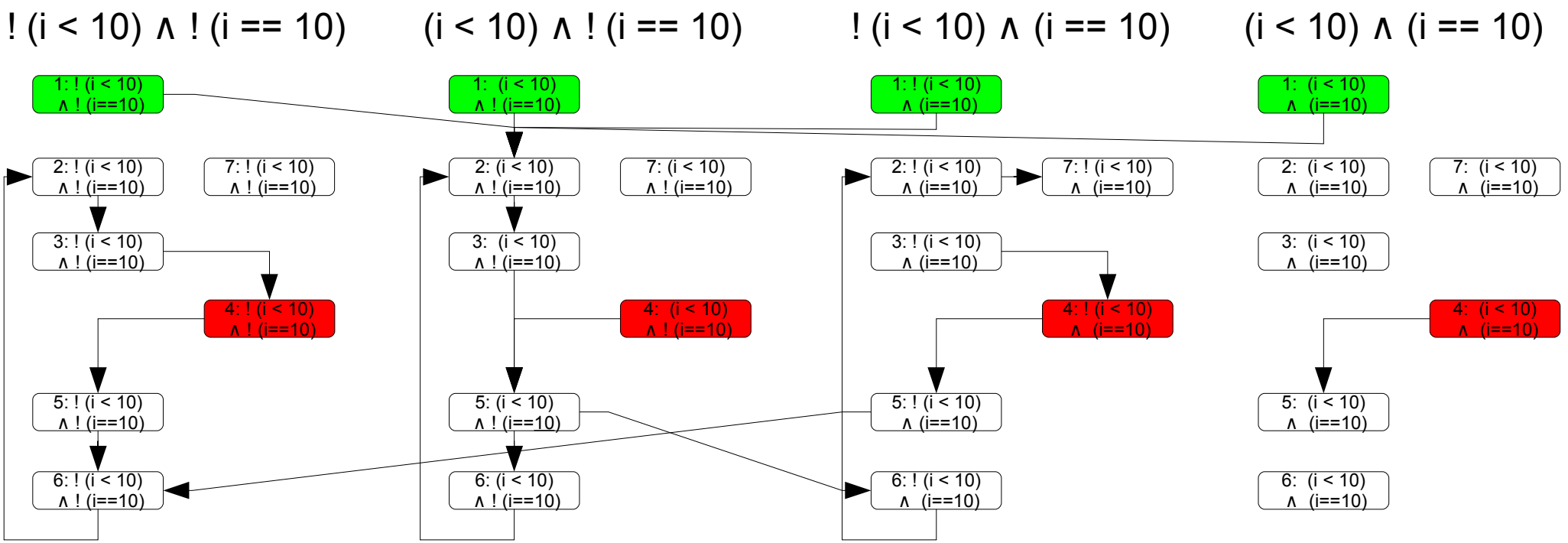
Simplify

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Example

Predicate Set: { i < 10, i == 10 }



Simplify

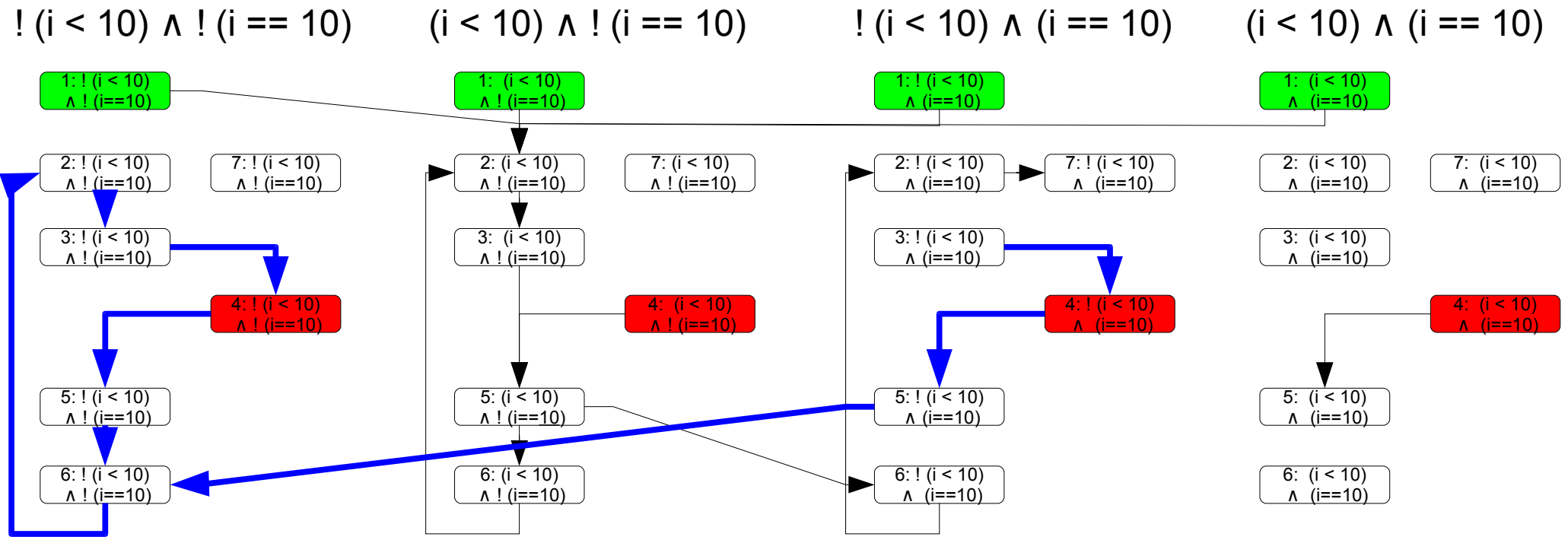
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Abstract

Example

Predicate Set: { i < 10, i == 10 }



Line 4 is unreachable

Assertion cannot fail

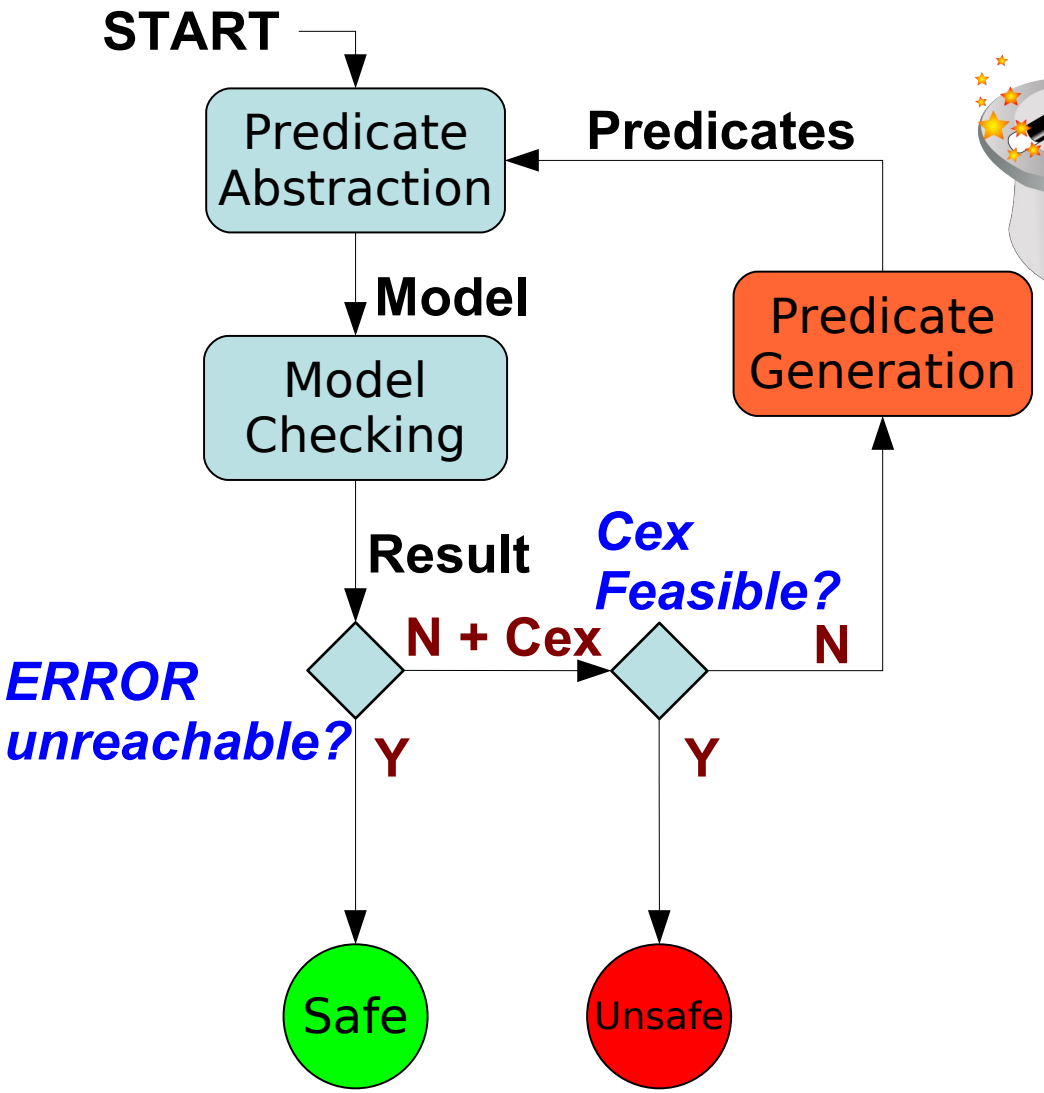
Simplify

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

Abstract → Check Model → Cex feasible? → Generate Predicates from Cex

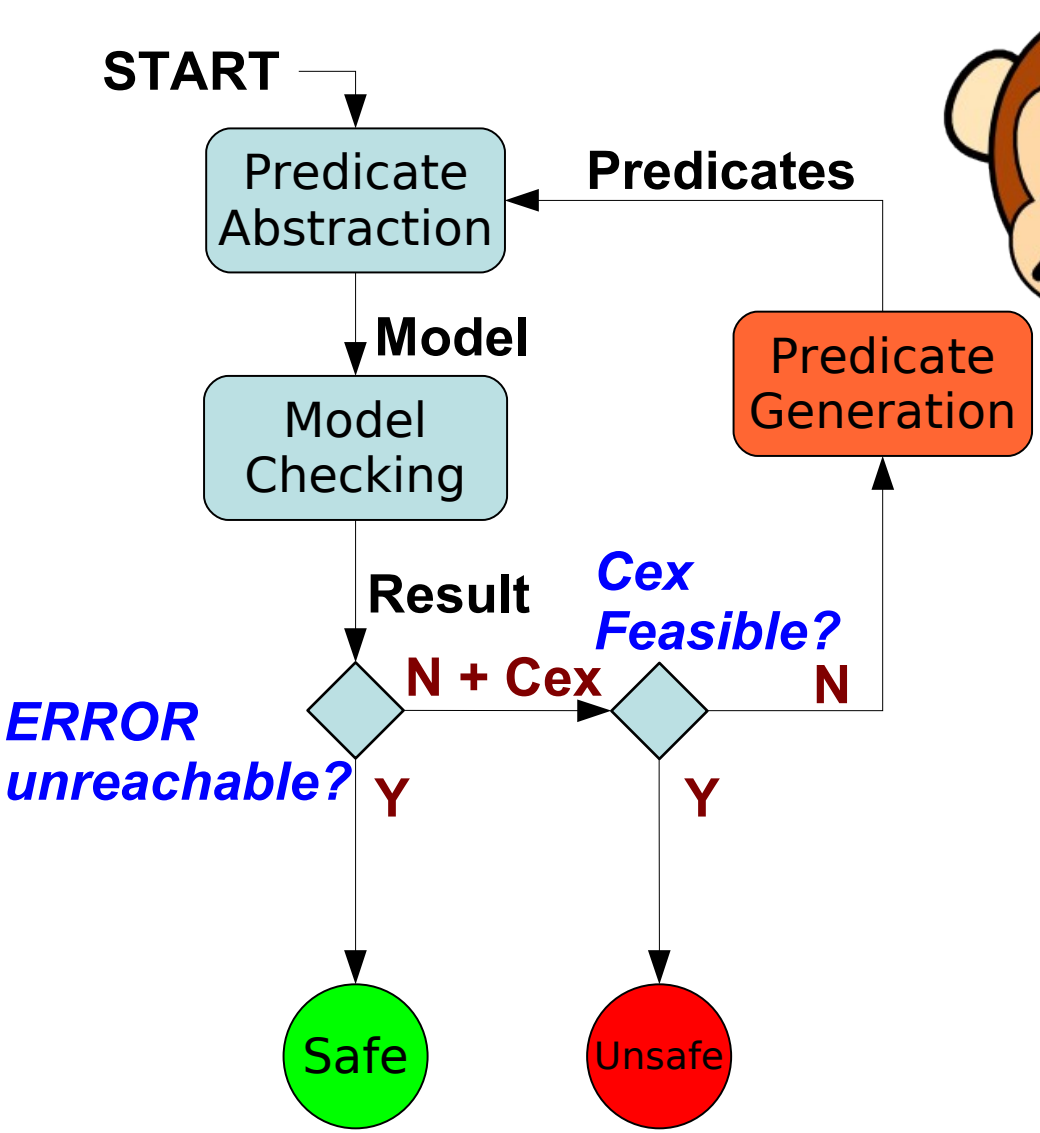
Abstract → Check Model → SAFE!

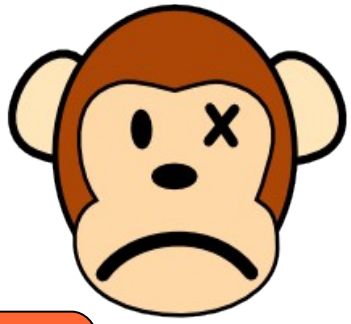
Predicate Generation



- Best case: perfect oracle

Predicate Generation





- Best case: perfect oracle
- Common case: finds *ineffective* predicates

New Example

```
1. i = 0;
2. m = 10;
3. while (i != m) {
4.    assert (i < 10);
5.
6.    i = i + 1;
7. }
8.
```

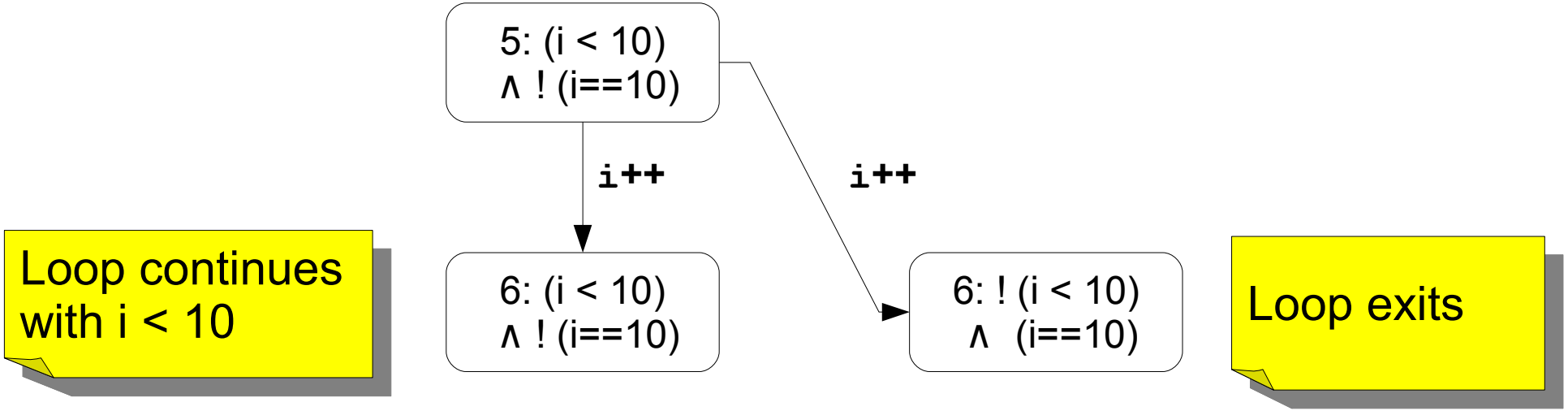
New Example

```
1. i = 0;
2. m = 10;
3. while (i != m) {
4.    assert (i < 10);
5.
6.    i = i + 1;
7. }
8.
```

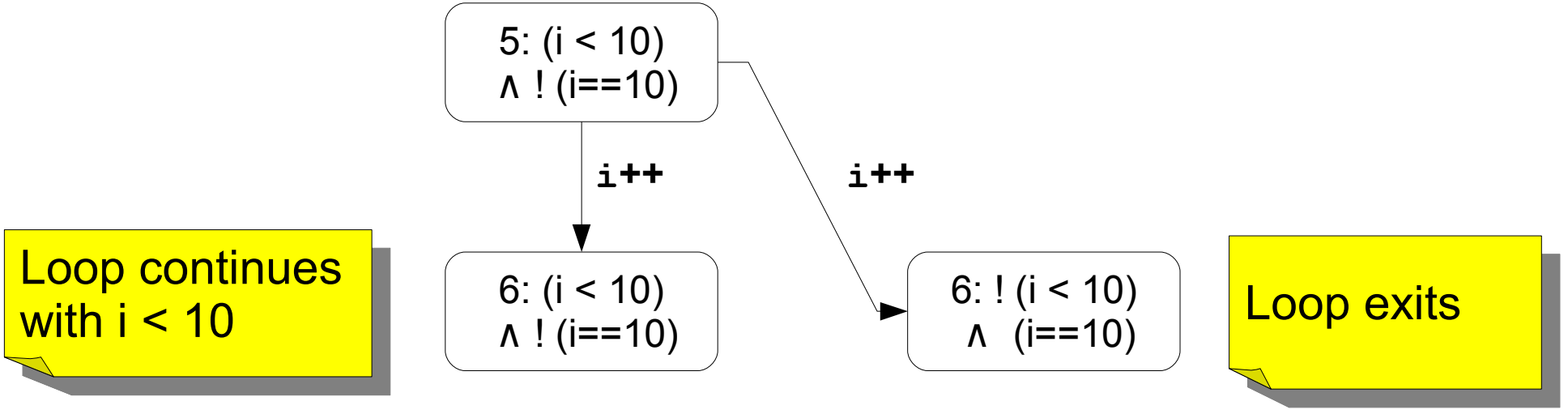
Predicate Set:
{ i < 10, i == m }

Simplify (Not shown)
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → ...

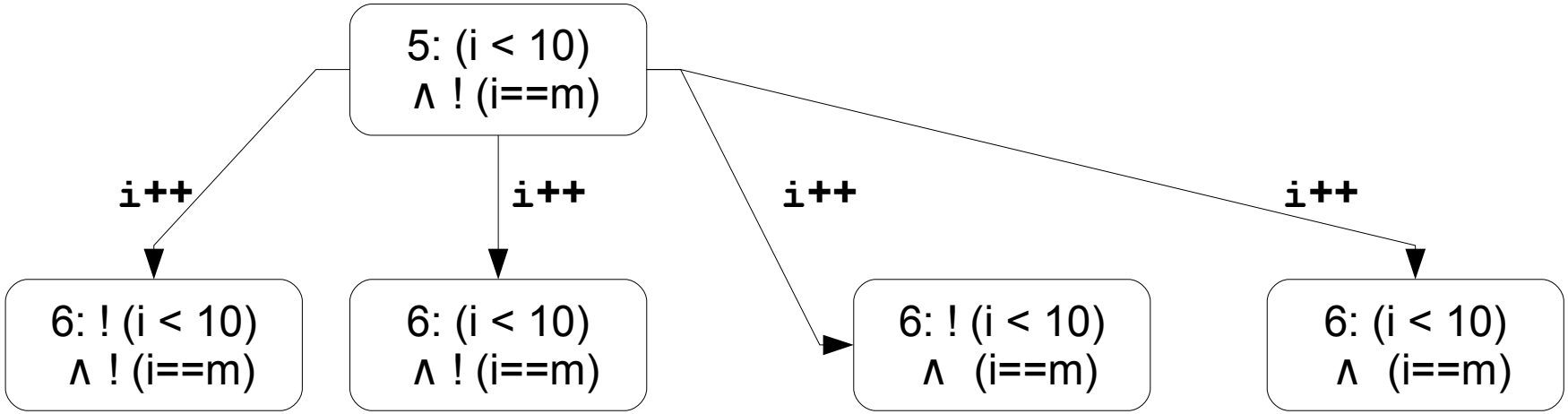
Old Example VS. New Example



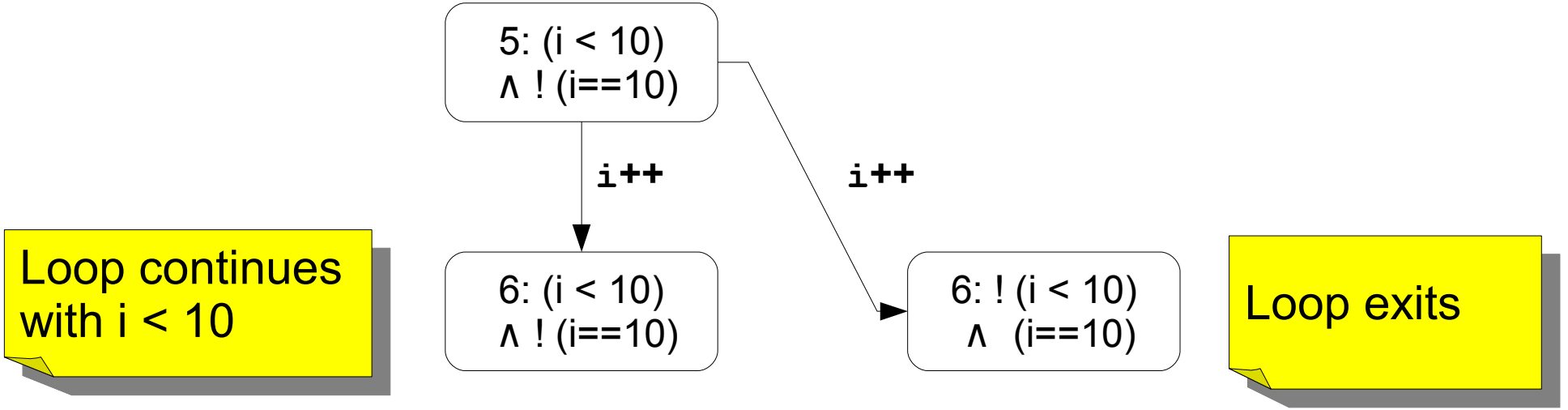
Old Example VS. New Example



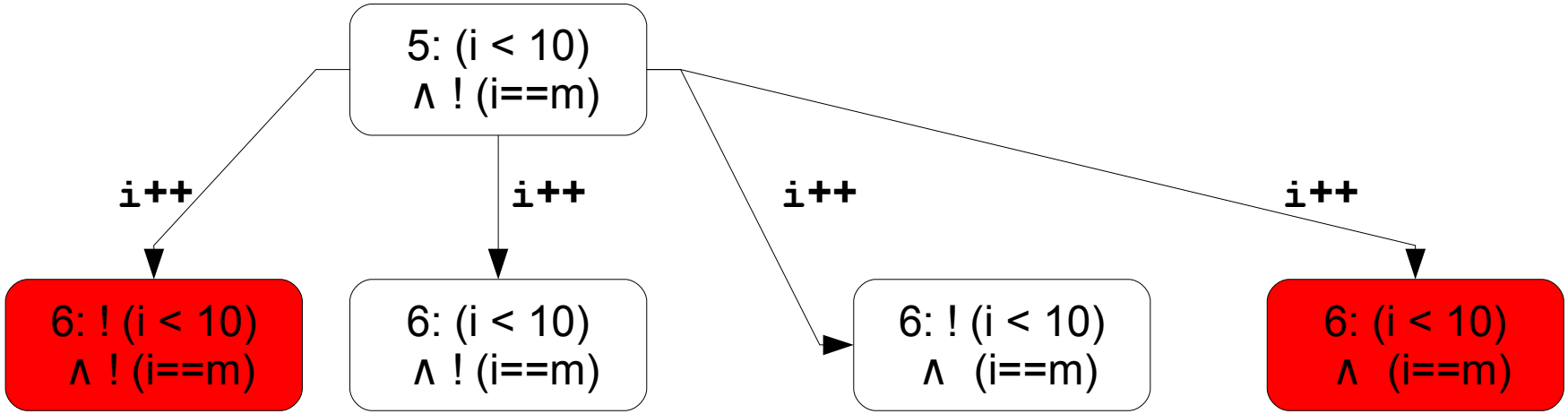
VS.



Old Example VS. New Example



VS.



New Example

```
1. i = 0;
2. m = 10;
3. while (i != m) {
4.    assert (i < 10);
5.
6.    i = i + 1;
7. }
8.
```

Predicate Set:
{ i < 10, i == m,
i + 1 < 10, i + 1 == m }

Simplify (Not shown)
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → ...

New Example

```
1. i = 0;
2. m = 10;
3. while (i != m) {
4.    assert (i < 10);
5.
6.    i = i + 1;
7. }
8.
```

Predicate Set:
{ i < 10, i == 10,
i + 1 < 10, i + 1 == m,
i + 2 < 10, i + 2 == m}

Simplify (Not shown)
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → ...

New Example

```
1. i = 0;
2. m = 10;
3. while (i != m) {
4.    assert (i < 10);
5.
6.    i = i + 1;
7. }
8.
```

Predicate Set:
{ i < 10, i == 10,
i + 1 < 10, i + 1 == m,
i + 2 < 10, i + 2 == m,
i + 3 < 10, i + 3 == m}

Simplify (Not shown)
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → Check Model → Cex feasible? → Generate Predicates from Cex
Abstract → ...

New Example

```
1. i = 0;
2. m = 10;
3. while (i != m) {
4.    assert (i < 10);
5.
6.    i = i + 1;
7. }
8.
```

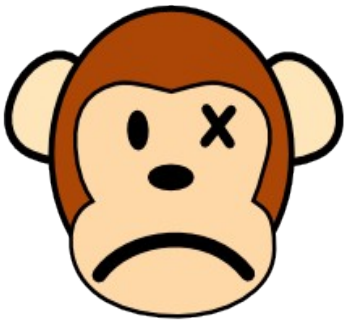
Predicate Set:
{ i < 10, i == m,
i + 1 < 10, i + 1 == m,
i + 2 < 10, i + 2 == m,
i + 3 < 10, i + 3 == m,
i + 4 < 10, i + 4 == m,
...
i + 10 < 10, i + 10 == m}

Loop unrolling leads to state explosion!

Simplify (Not shown)
Abstract → Check Model → Cex feasible? → ...rate ...edicates from Cex
Abstract → Check Model → Cex feasible? → ...enerate Predicates from Cex
Abstract → ...

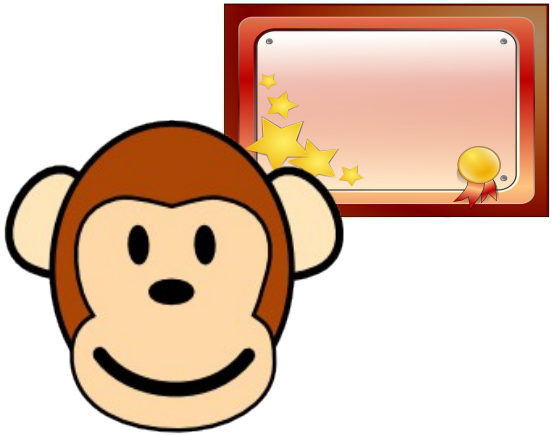
Loop Unrolling

- Due to poor predicate generation
- An efficient proof exists
  - ➔ m == 10, i == m, i < 10
- Tool finds an *inefficient proof*



Loop Unrolling

- Due to poor predicate generation
- An efficient proof exists
  - → m == 10, i == m, i < 10
- Tool finds an *inefficient proof*



Solution: Teach Software Model Checker About Efficient Proofs for Common Array Traversals

Outline

1. Introduction
   i. Buffer overflows
   ii. Need for verification
   iii. Preliminary Evaluation
2. Background
   i. Sofware Model Checking with CEGAR
   ii. Why loop unrolling happens
3. Solution: proof templates
4. Tool support and results
5. Conclusions and Future Work

Template : Intuition

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

Template : Intuition

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Observe:
  - ➔ `i` changes in loop body (line 7)

Template : Intuition

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Observe:
  - `i` changes in loop body (line 7)
  - `i` must be bounded in loop body (line 6)

Template : Intuition

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Observe:
  - `i` changes in loop body (line 7)
  - `i` must be bounded in loop body (line 6)
- Thus loop must bound `i`

Template : Intuition

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Observe:
  - `i` changes in loop body (line 7)
  - `i` must be bounded in loop body (line 6)
- Thus loop must bound `i`
- Approach:
  - conjecture a proof that the loop bounds `i`

Template : Example

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

Template : Example

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Examine:
  - *loop exit conditions*
    - `i` bounded by `m`
  - *bounds checks*
    - `i` must be bounded by `10`

Template : Example

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Examine:
  - *loop exit conditions*
    - `i` bounded by `m`
  - *bounds checks*
    - `i` must be bounded by `10`
- Proof:
  - initially, `i` <= `m` <= 10
  - loop invariant: `i` < `m` <= 10
  - thus at line 6, `i` < 10

Template : Example

```
1. i = 0;
2. m = 10;
3.
4.
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Guide tool towards this proof

Template : Example

```
1. i = 0;
2. m = 10;
3. assume (i <= m);
4. assume (m <= 10);
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

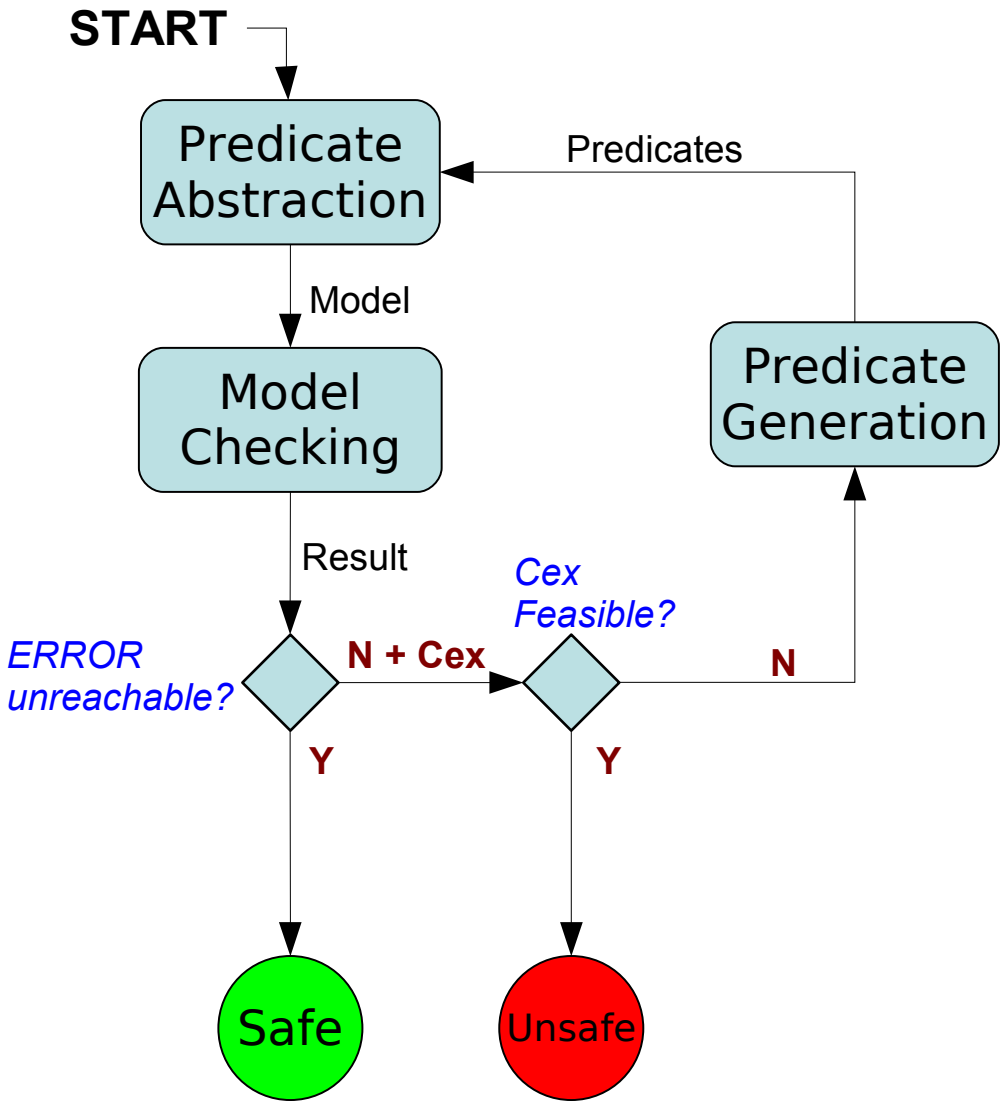
- Guide tool towards this proof
- Supply:
  - assumptions of proof
    - discharge later

Template : Example

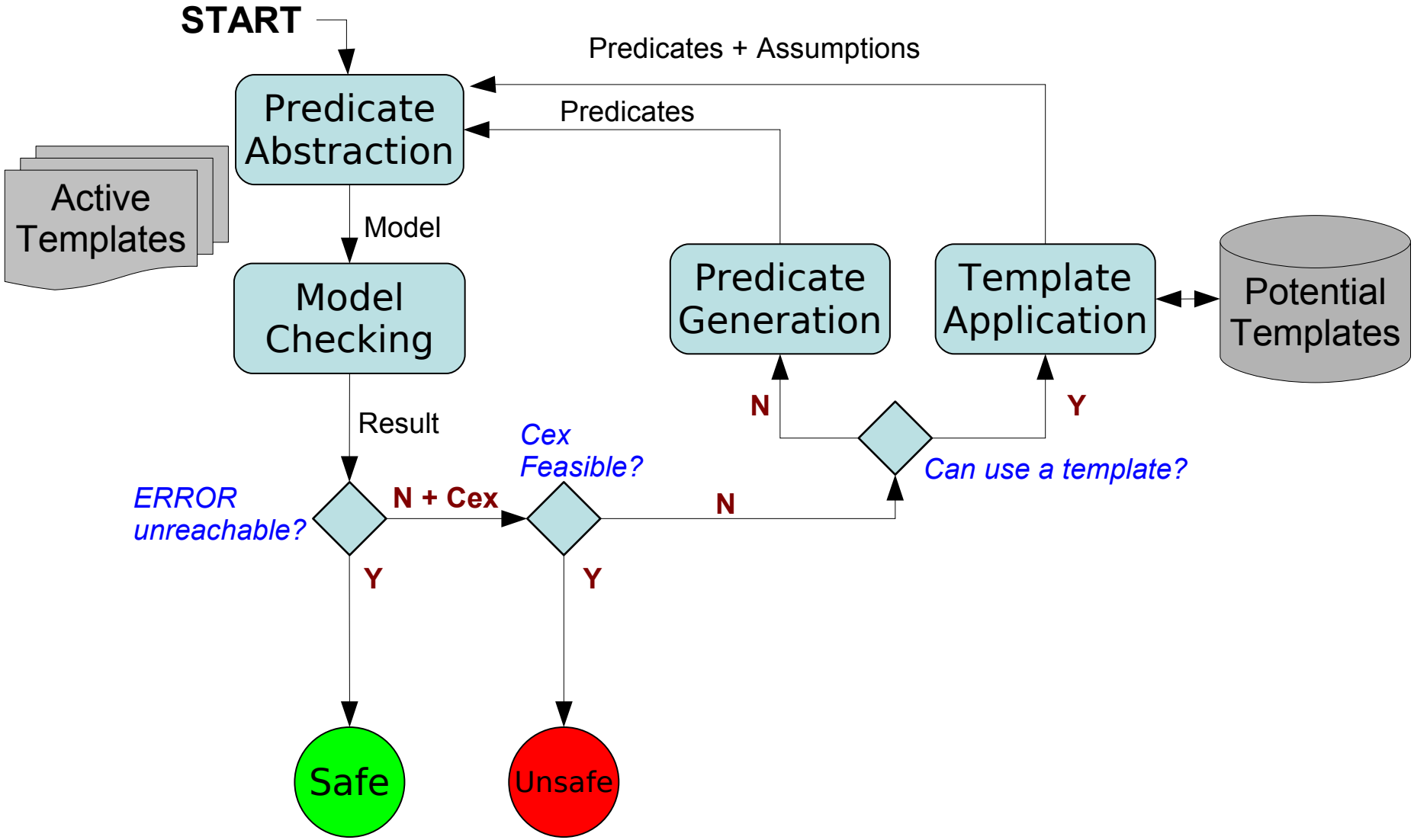
```
1. i = 0;
2. m = 10;
3. assume (i <= m);
4. assume (m <= 10);
5. while (i != m) {
6.    assert (i < 10);
7.    i = i + 1;
8. }
```

- Guide tool towards this proof
- Supply:
  - assumptions of proof
    - discharge later
  - predicates appearing in proof
    - `i <= m`
    - `m <= i`
    - `m <= 10`

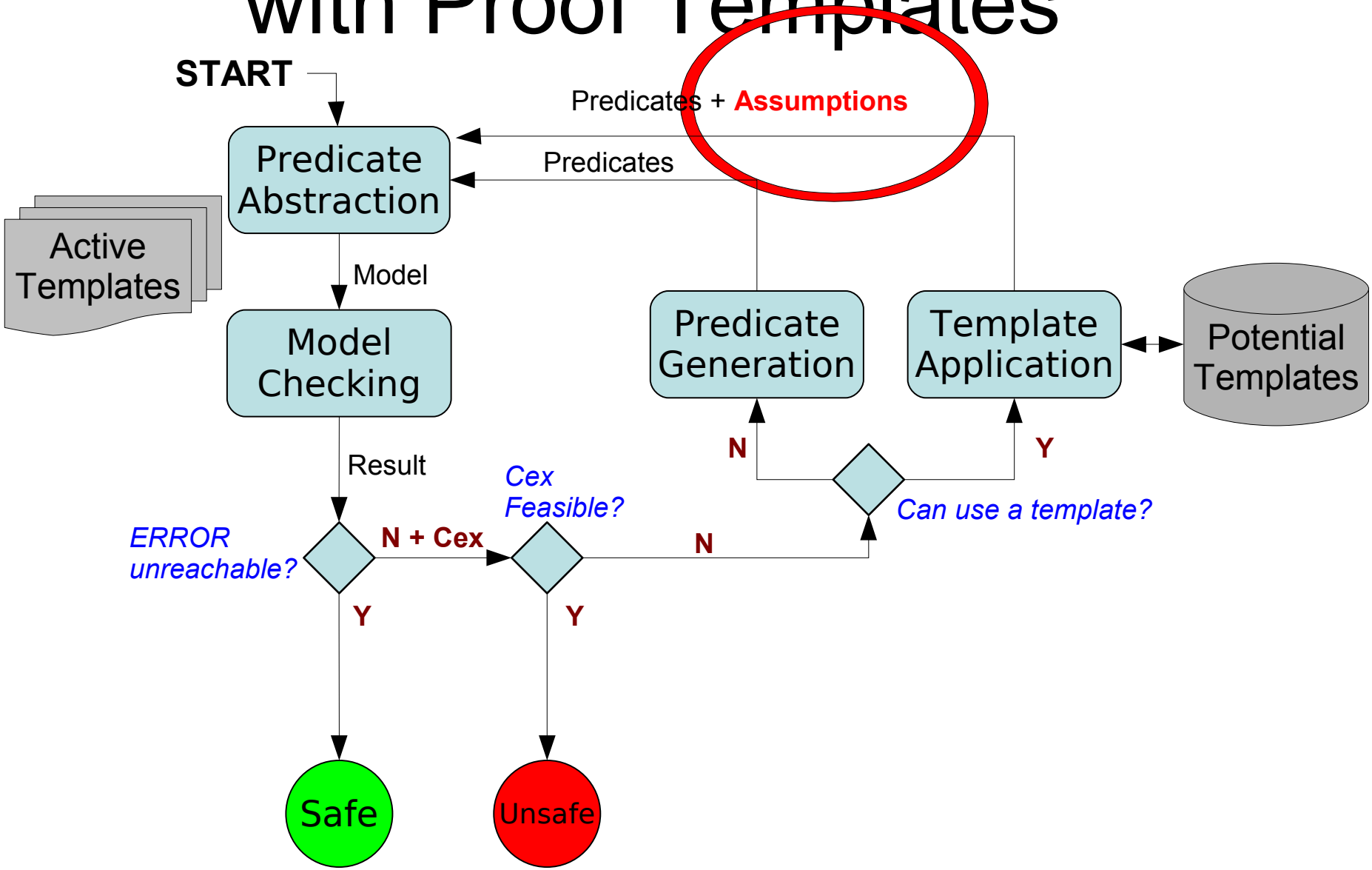
CEGAR Software Model Checking



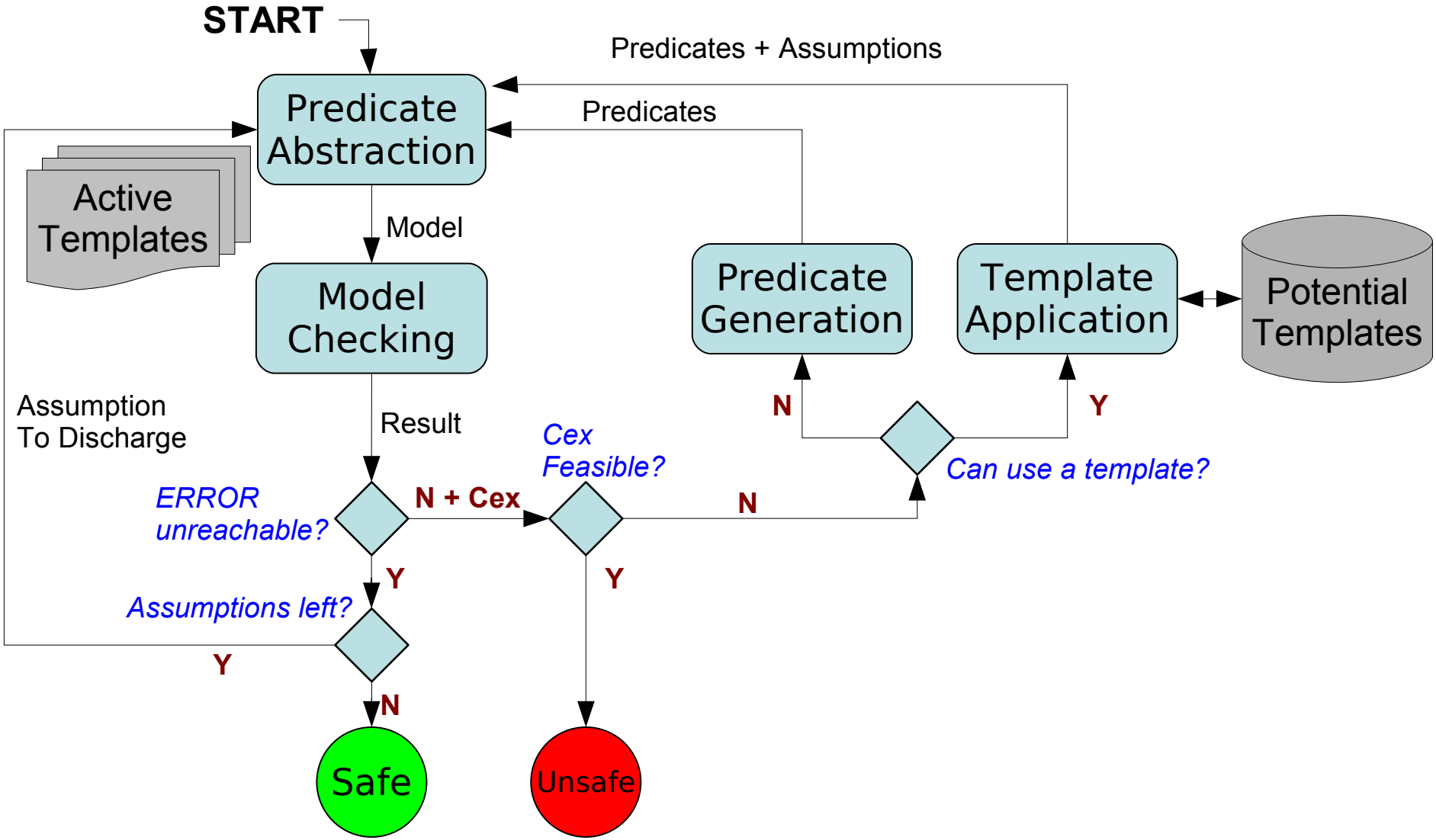
CEGAR Software Model Checking with Proof Templates



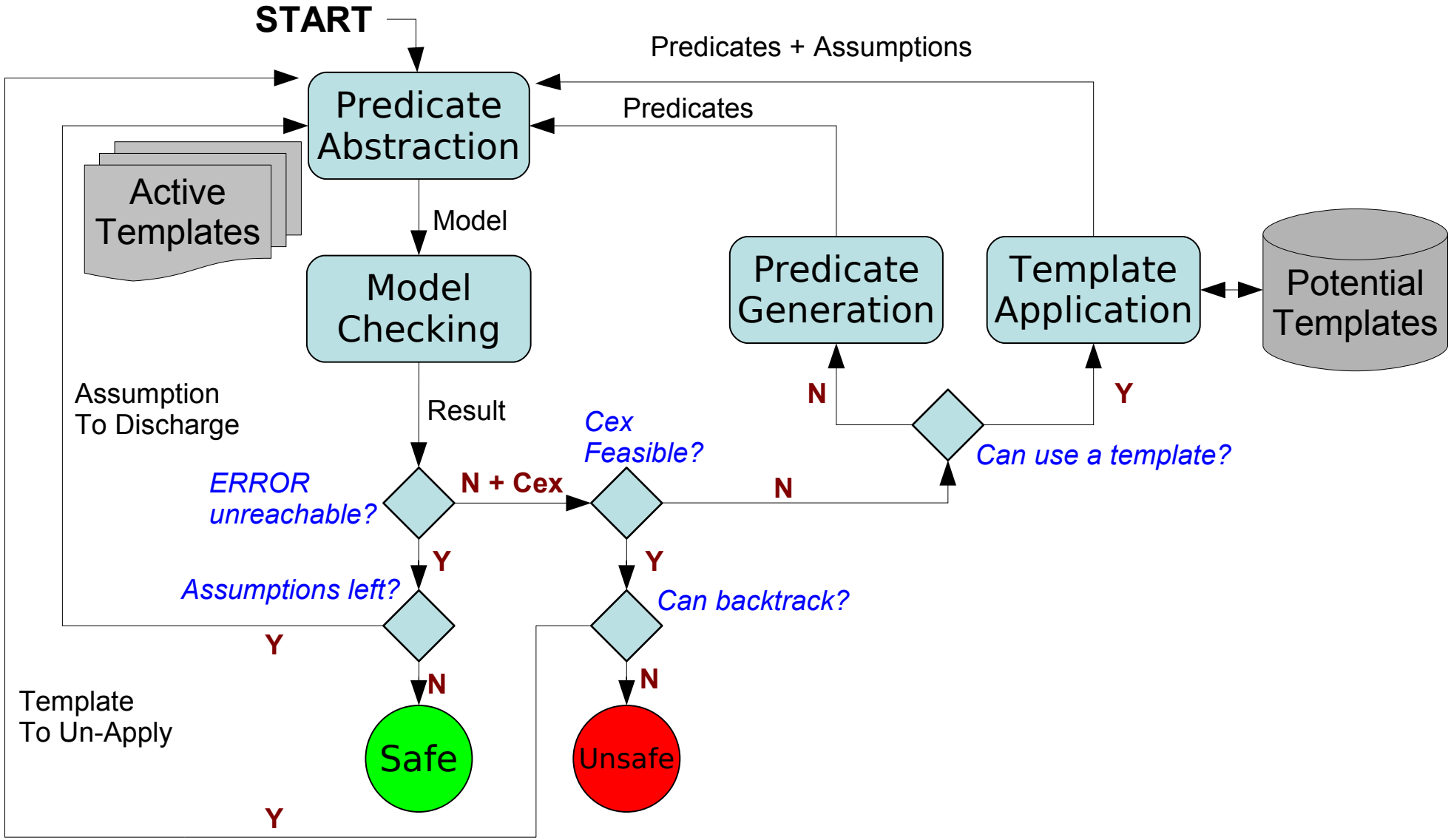
CEGAR Software Model Checking with Proof Templates



CEGAR Software Model Checking with Proof Templates



CEGAR Software Model Checking with Proof Templates



Proof Structure

```
P: { ... }



while (i != m) {
    Q: { ... }
    assert (i <= N);
    R: { ... }
 }
```

Proof Structure : Assume Phase

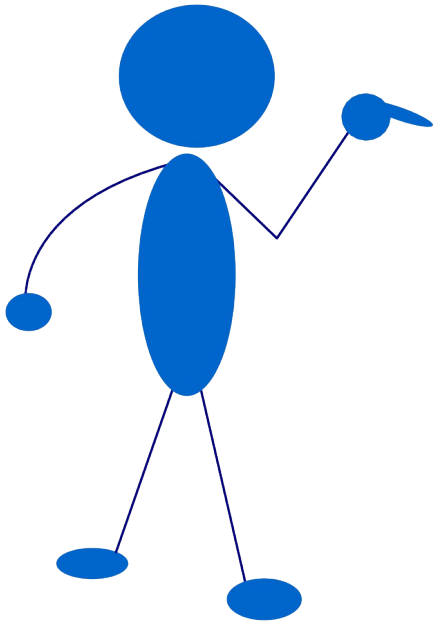
```
P: { ... }
assume (i <= m);
assume (m + c <= N);
while (i != m) {
    Q: { ... }
    assert ((i <= m + c) && (m + c <= N));
    R: { ... }
 }
```

Proof Structure : Discharge Phase

```
P: { ... }
assert (i <= m);
assert (m + c <= N);
while (i != m) {
   Q: { ... }

   R: { ... }
 }
```

Generalizations

- ✔ Less structured loops
- ✔ String traversals
- ✔ Two-variable traversals

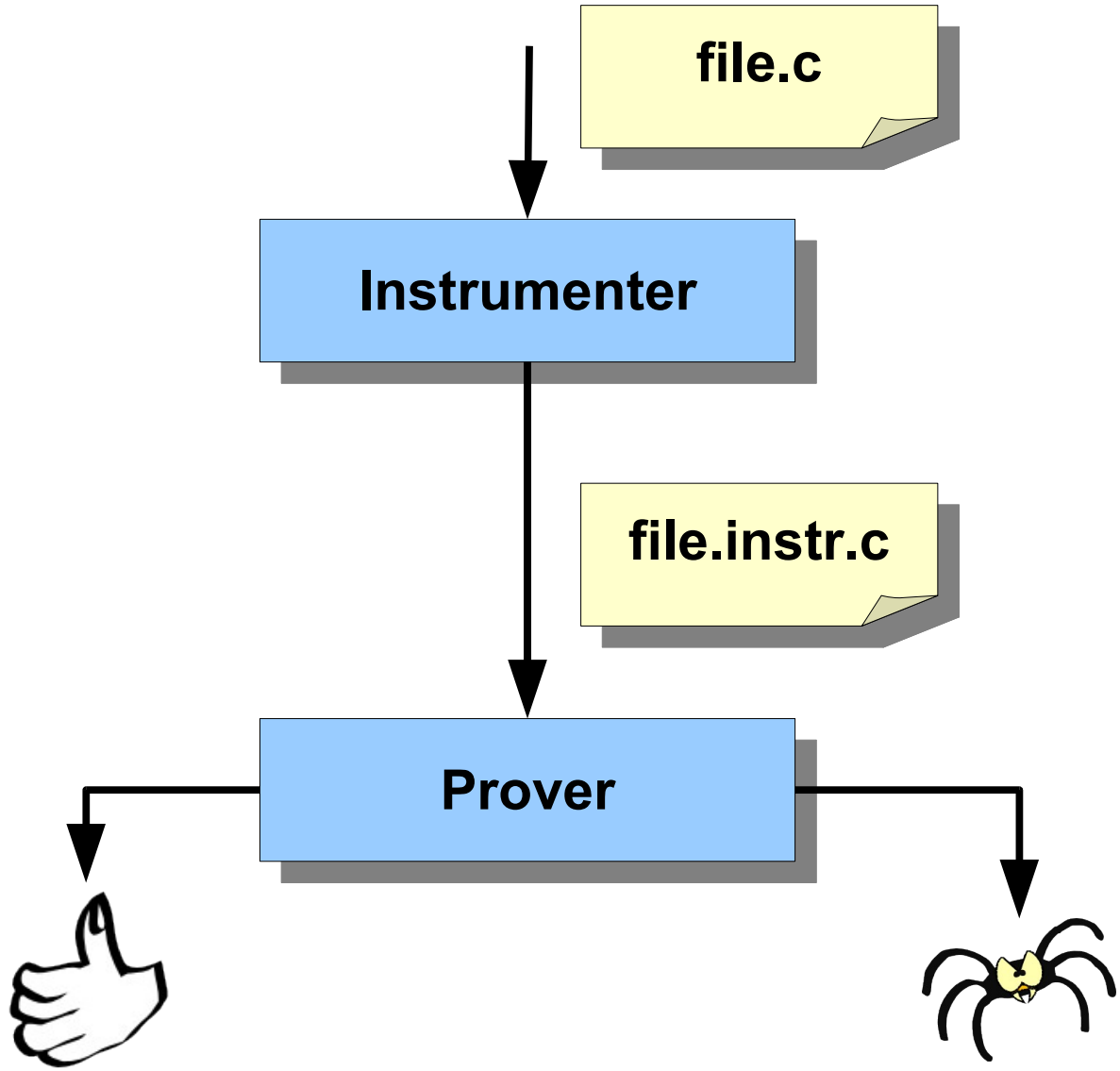


```
void foo () {
  char src[1024], dest[1024];
  char ch;
  int i=0, j=0;

  src[1023] = '\0';
  if (src[i] == '*') i++;
  while (1) {
    ch = src[i];
    if (ch=='\0' || ch==',')
      break;
    assert (j < 1024);
    dest[j] = ch;
    j++;
    i++;
  }
}
```

Outline

1. Introduction
   i. Buffer overflows
   ii. Need for verification
   iii. Preliminary Evaluation
2. Background
   i. Sofware Model Checking with CEGAR
   ii. Why loop unrolling happens
3. Solution: proof templates
4. Tool support and results
5. Conclusions and Future Work

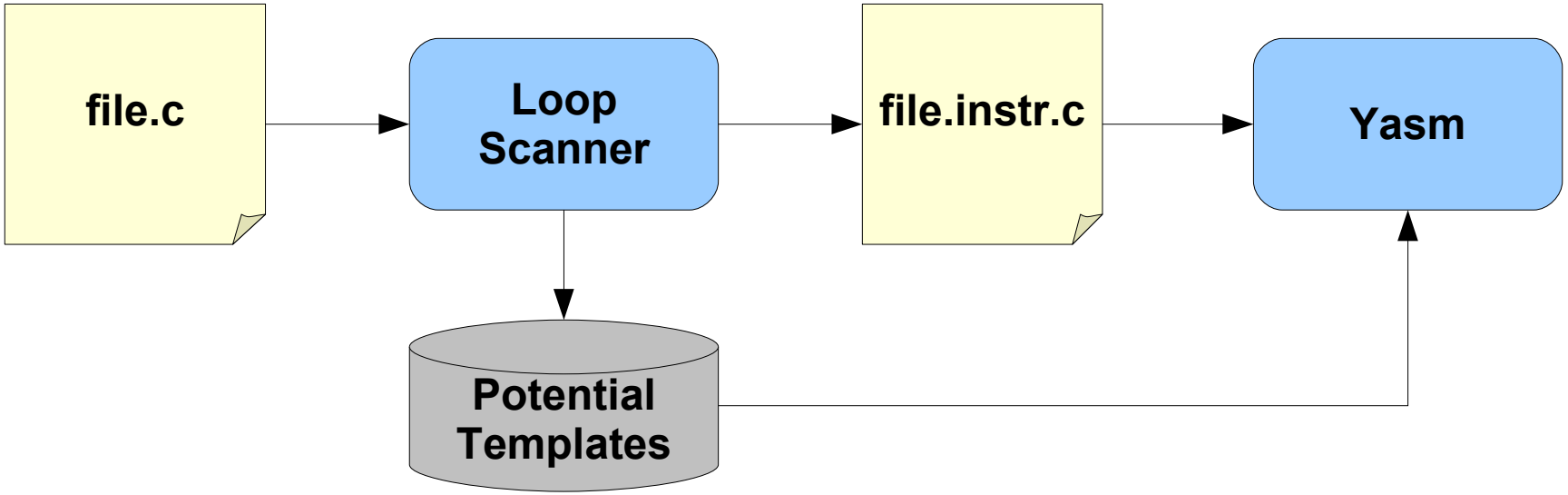
Our Approach to Verification




```
char A [100];
...
assert (0 <= i < 100);
A[i] = x;
```

Invoke prover once for *each* assertion

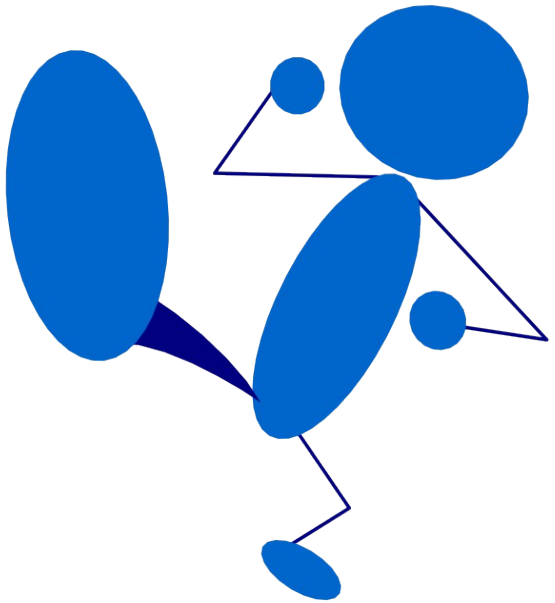
PtYasm



- Loop scanner
  - extension to CIL
  - finds potential template instances and parameters
  - adds instrumentation
- Yasm
  - software model checker
  - template-aware
  - implements backtracking

Experiments

- 59 array bounds-checking testcases
  - derived from the Verisec suite
  - simplified syntax
  - one loop and one assertion per testcase
- Evaluate PtYasm relative to:
  - Yasm
  - BLAST
  - SatAbs



Experimental Process

- Run each tool on each testcase
  - 10 minute timeout
  - tool passes if it proves the assertion safe within the timeout

Example Testcase

```
void foo () {
  char src[1024], dest[1024];
  char ch;
  int i=0, j=0;

  src[1023] = '\0';
  if (src[i] == '*') i++;
  while (1) {
    ch = src[i];
    if (ch=='\0' || ch==',')
      break;
    assert (j < 1024);
    dest[j] = ch;
    j++;
    i++;
  }
}
```

Testcases Verified

	Testcases
Total	59
PtYasm	49
Yasm	17
BLAST	19
SatAbs	22

- ✔ PtYasm verifies over twice as many testcases as the next best tool

Testcases Verified

	Testcases	String	Two-variable
Total	59	14	10
PtYasm	49	11	7
Yasm	17	0	0
BLAST	19	0	0
SatAbs	22	0	0

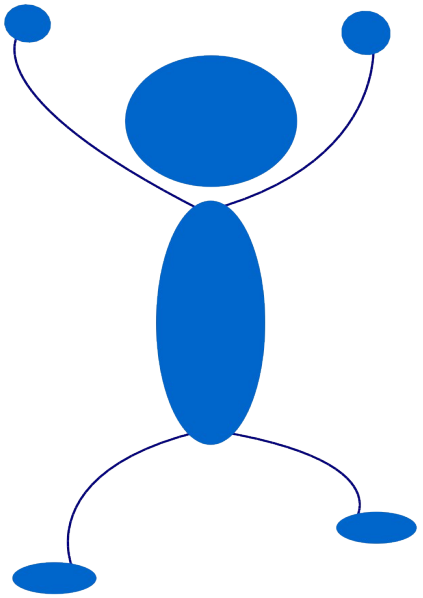
- ✔ PtYasm verifies over twice as many testcases as the next best tool
- ✔ Only PtYasm verifies string or two-variable testcases

Outline

1. Introduction
   i. Buffer overflows
   ii. Need for verification
   iii. Preliminary Evaluation
2. Background
   i. Sofware Model Checking with CEGAR
   ii. Why loop unrolling happens
3. Solution: proof templates
4. Tool support and results
5. Conclusions and Future Work

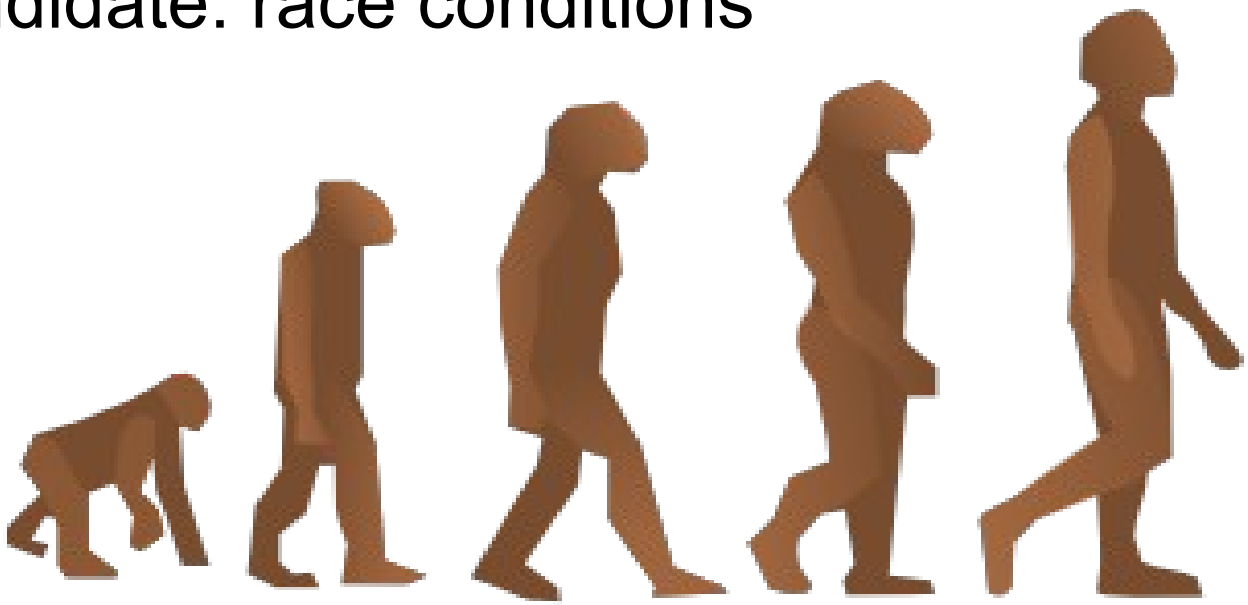
Conclusions

- Proof templates:
  - ✔ Enable verification without loop unrolling
  - ✔ Make previously infeasible analyses feasible
  - ✔ Are a promising foundation for enhancing software model checkers
    - ✔ Especially for buffer overflows!



Future Work

- Extend to multiple loops
- Other proof templates for array bounds checking
- Proof templates for other domains
  - Candidate: race conditions




QUESTIONS?

References

- T. E. Hart, K. Ku, D. Lie, M. Chechik, and A. Gurfinkel. Augmenting Counterexample-Guided Abstraction Refinement with Proof Templates. In Proc. ASE'08.
- T. E. Hart, K. Ku, D. Lie, M. Chechik, and A. Gurfinkel. PtYasm: Software Model Checking with Proof Templates. In Proc. ASE'08 --- Tool Demonstrations Track.
- K. Ku, T. E. Hart, M. Chechik, and D. Lie. A Buffer Overflow Benchmark for Software Model Checkers. In Proc. ASE'07.