Exploiting the SAT Revolution for Automated Software Verification and Synthesis

Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk

Before Joining Manchester

1 7

BSc/MSc in Electrical/ Computer Engineering

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

5

NXP

Set-top Box Software Engineer

6

PhD in Computer Science

8

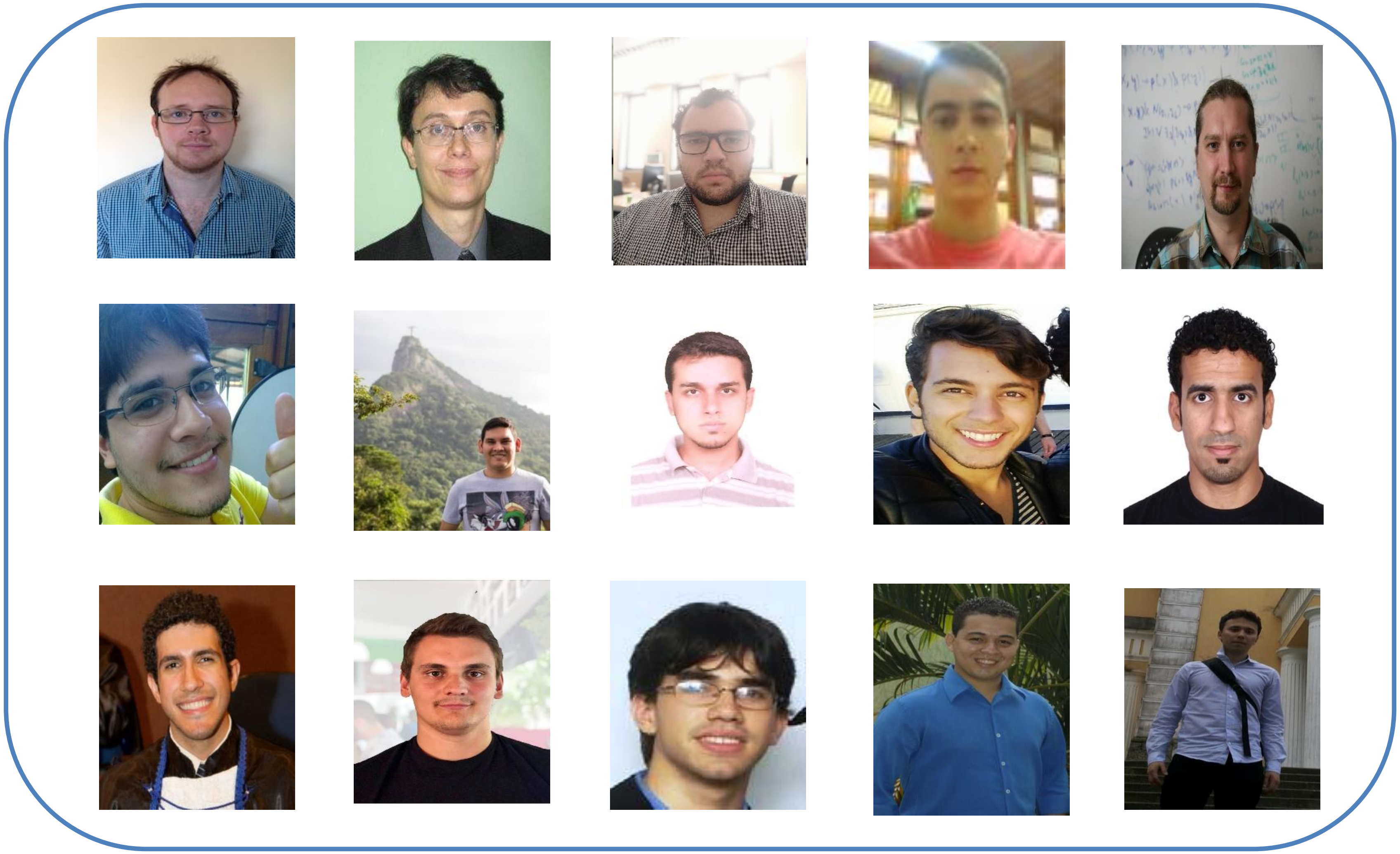
Postdoctoral Researcher

8

diffblue AI for Code

Research Engineer

Research Group on Systems and Software Verification



Collaborators

aws

UNIVERSITY OF OXFORD

UNIVERSITEIT STELLENBOSCH UNIVERSITY

UNIVERSITY OF Southampton
School of Electronics and Computer Science

UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

intel

UFRR

UFAM

SAMSUNG SIDIA

diffblue
AI for Code

arm

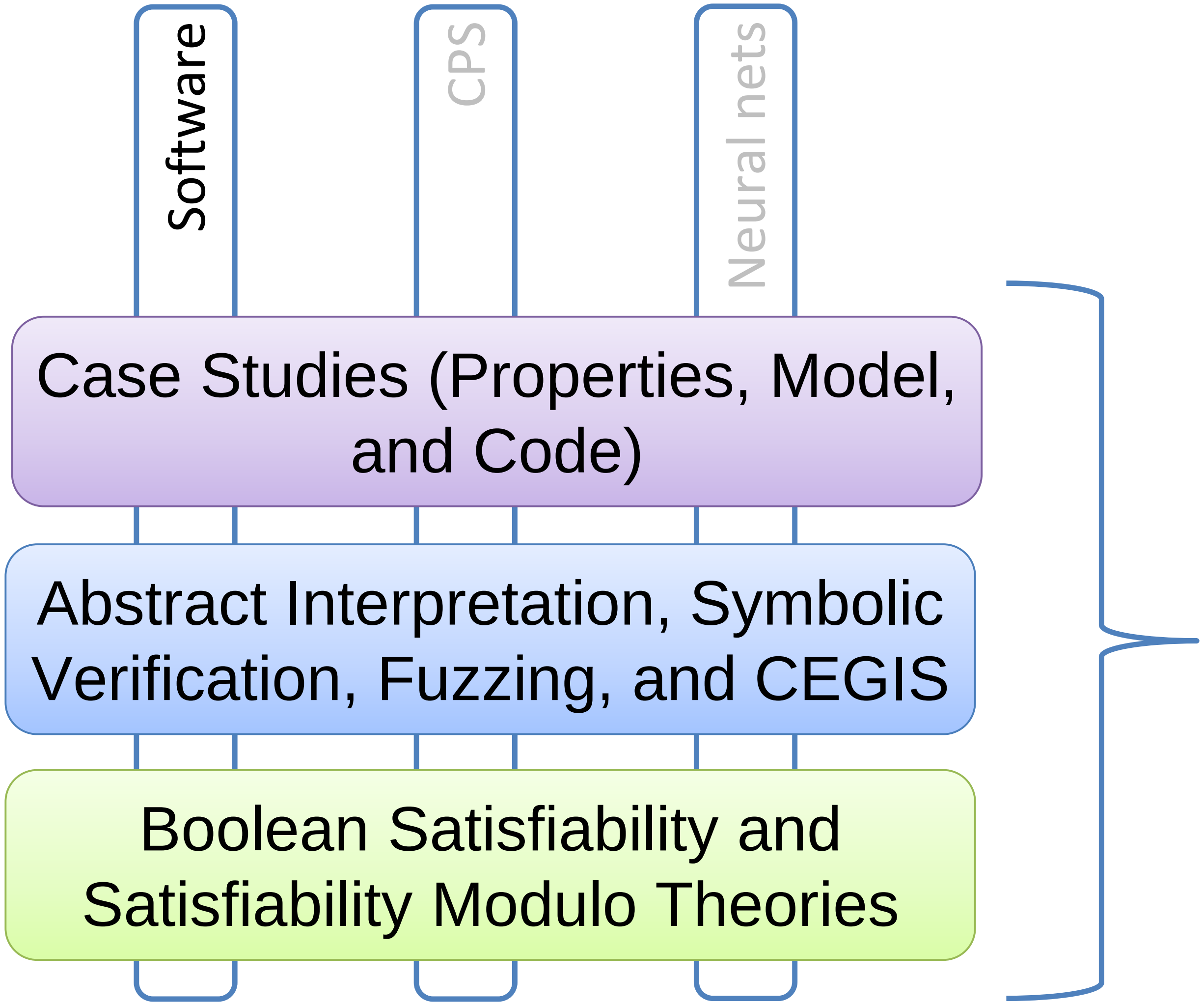
UNIVERSITY OF CAMBRIDGE

Research Objectives

leverage **program analysis/synthesis** to improve **coverage** and reduce **verification time** for finding **vulnerabilities** in software

leverage **program analysis/synthesis** to achieve **correct-by-construction** software systems considering **safety** and **security**

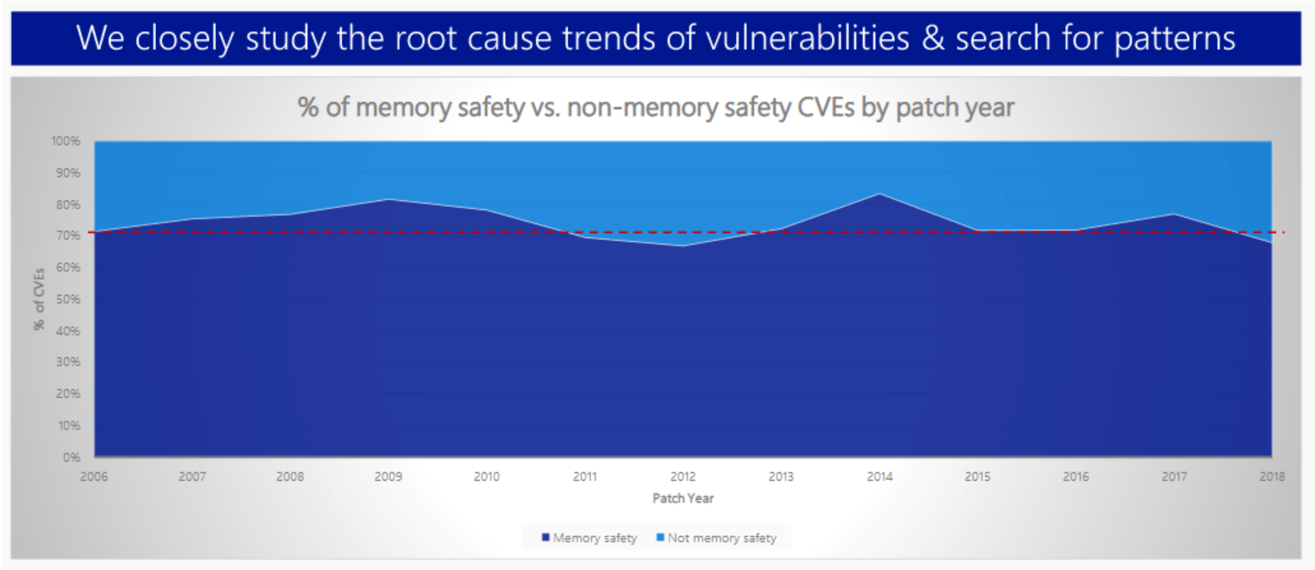
Outline


Software
CPS
Neural nets
Case Studies (Properties, Model, and Code)
Abstract Interpretation, Symbolic Verification, Fuzzing, and CEGIS
Boolean Satisfiability and Satisfiability Modulo Theories

Approaches to formally build verified **trustworthy software systems** to ensure **confidentiality, integrity** and **availability**

70 percent of all security bugs are memory safety issues

"The majority of vulnerabilities are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code. As Microsoft increases its code base and uses more Open Source Software in its code, this problem isn't getting better, it's getting worse (2019)."



https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Security Vulnerabilities

```
int getPassword() {
  char buf[4];
  gets(buf);
  return strcmp(buf, ”SMT”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

- What happens if the user enters “SMT”?
- On a Linux x64 platform running GCC 4.8.2, an input consisting of 24 arbitrary characters followed by *]*, <*ctrl-f*>, and @, will bypass the “Access Denied” message
- A longer input will run over into other parts of the **computer memory**

Exciting research projects concerning software security and automated reasoning:

EnnCore SCorCH ELEGANT

Boolean Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

SAT = {<Φ> : Φ is a satisfiable Boolean formula}

  - Example:
    - o $\Phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
    - o Assignment: $<x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1>$
    - o $\Phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$
    - o $\Phi = (1 \vee \neg(1 \vee 1)) \wedge 1$
    - o $\Phi = (1 \vee 0) \wedge 1$

unit propagation, conflict clauses and non-chronological backtracking

DPLL satisfiability solving

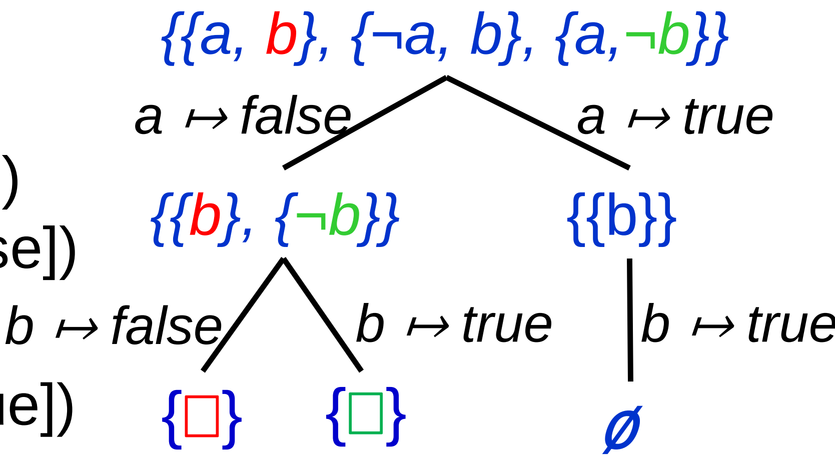
Given a Boolean formula φ in *clausal form* **(an AND of ORs)**

{{a, b}, {¬a, b}, {a,¬b}, {¬a,¬b}}

determine whether a *satisfying assignment* of variables to truth values exists.

Solvers based on Davis-Putnam-Logemann-Loveland algorithm:

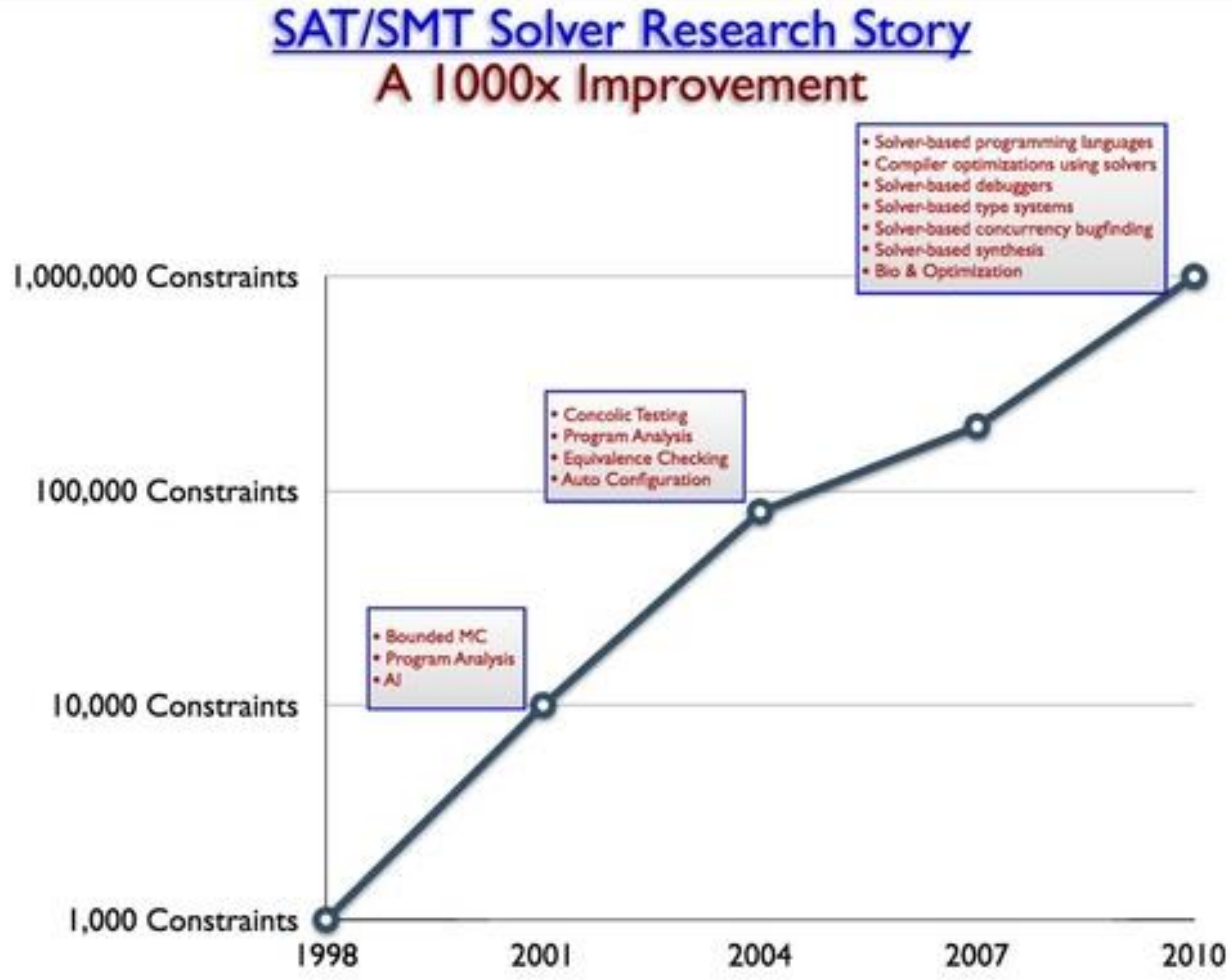
1. If $\varphi = \emptyset$ then SAT
2. if $\square \in \varphi$ then UNSAT
3. If $\varphi = \varphi' \cup \{x\}$ then DPLL($\varphi'[x \mapsto \text{true}]$)
   If $\varphi = \varphi' \cup \{\neg x\}$ then DPLL($\varphi'[x \mapsto \text{false}]$)
4. Pick arbitrary x and return
   DPLL($\varphi[x \mapsto \text{false}]$) ∨ DPLL($\varphi[x \mapsto \text{true}]$)



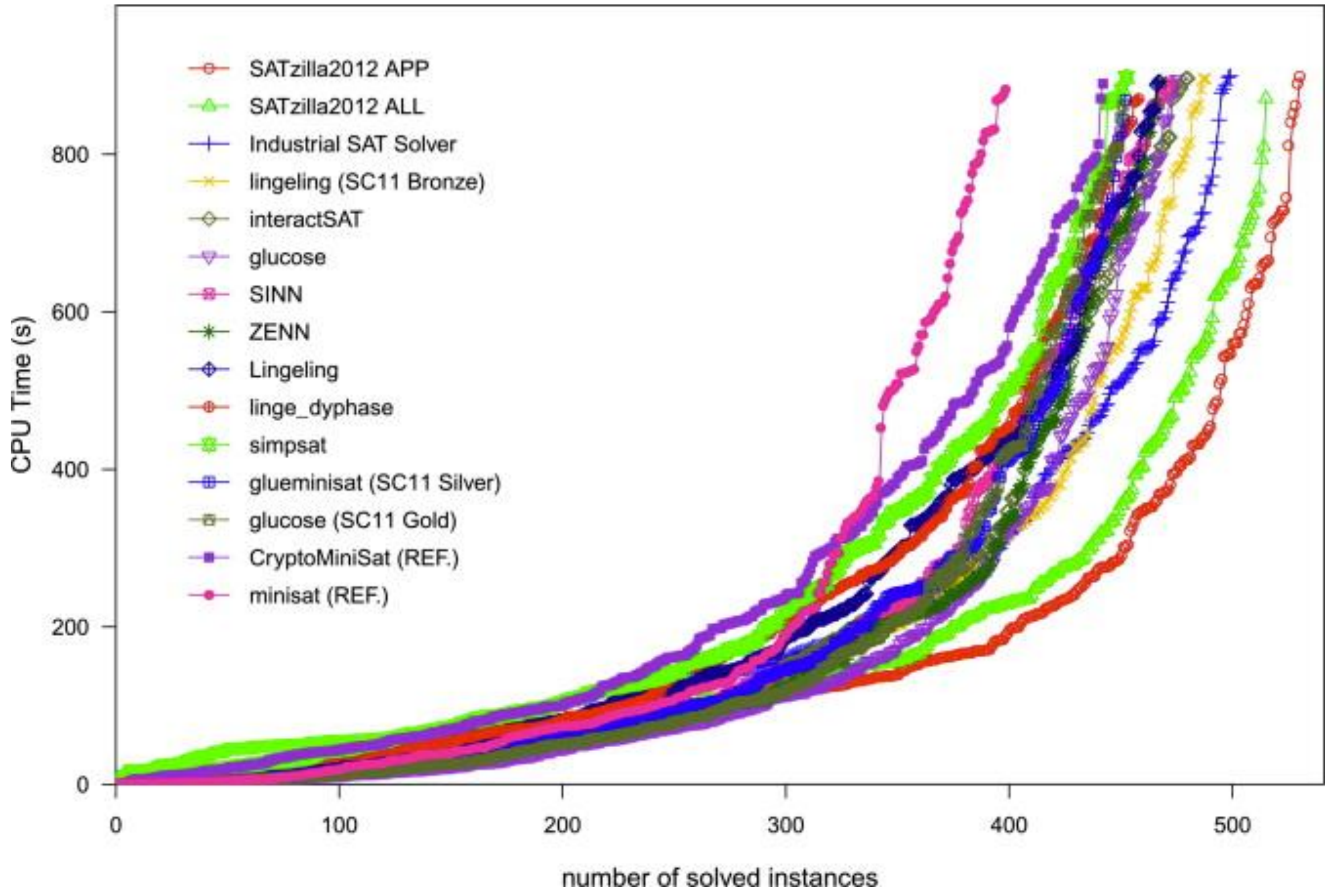
+ NP-complete but many heuristics and optimizations

⇒ can handle problems with 1,000,000's of variables

SAT solving as enabling technology



SAT Competition



Satisfiability Modulo Theories

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories**

Theory	**Example**
Equality	$\mathbf{x_1=x_2 \wedge \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)}$
Bit-vectors	$(b >> i) \,\&\, 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \wedge a[k]=2) \Rightarrow a[j]=2$
Combined theories	$(j \leq k \wedge a[j]=2) \Rightarrow a[i] < 3$

Satisfiability Modulo Theories

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories**

Theory	**Example**
Equality	$x_1=x_2 \land \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	**$(b >> i) \& 1 = 1$**
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \land a[k]=2) \Rightarrow a[j]=2$
Combined theories	$(j \leq k \land a[j]=2) \Rightarrow a[i] < 3$

$(a > 0) \land (b > 0) \Rightarrow (a + b > 0)$

Satisfiability Modulo Theories

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories**

Theory	**Example**
Equality	$x_1=x_2 \land \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	(b >> i) & 1 = 1
Linear arithmetic	$\mathbf{(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)}$
Arrays	$(j = k \land a[k]=2) \Rightarrow a[j]=2$
Combined theories	$(j \leq k \land a[j]=2) \Rightarrow a[i] < 3$

Satisfiability Modulo Theories

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories**

Theory	**Example**
Equality	$x_1=x_2 \wedge \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	$(b >> i) \,\&\, 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Arrays	$\mathbf{(j = k \wedge a[k]=2) \Rightarrow a[j]=2}$
Combined theories	$(j \leq k \wedge a[j]=2) \Rightarrow a[i] < 3$

$i = j \Rightarrow \text{select}(\text{store}\ (a, i, v), j) = v$

$i \neq j \Rightarrow \text{select}(\text{store}\ (a, i, v), j) = \text{select}(a, j)$

Satisfiability Modulo Theories

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories**

Theory	**Example**
Equality	$x_1=x_2 \land \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	(b >> i) & 1 = 1
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \land a[k]=2) \Rightarrow a[j]=2$
Combined theories	$\mathbf{(j \leq k \land a[j]=2) \Rightarrow a[j] < 3}$

SMT-based Verification

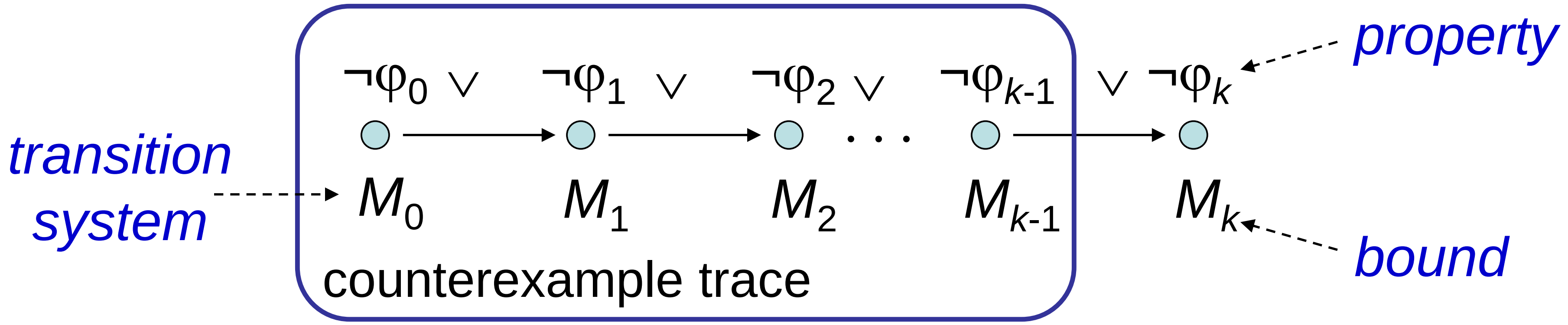
- Given
  - a decidable $\sum$-theory T
  - a quantifier-free formula φ

  φ is T-satisfiable iff $T \cup \{\varphi\}$ is satisfiable, i.e., there exists a structure that satisfies both formula and sentences of T
- Given
  - a set $\Gamma \cup \{\varphi\}$ of first-order formulae over T

  φ is a T-consequence of Γ ($\Gamma \models_T \varphi$) iff every model of $T \cup \Gamma$ is also a model of φ
- Checking $\Gamma \models_T \varphi$ can be reduced in the usual way to checking the T-satisfiability of $\Gamma \cup \{\neg\varphi\}$

Bounded Model Checking (BMC)

Basic idea: check negation of given property up to given depth


$\neg\varphi_0 \vee$ $\neg\varphi_1 \vee$ $\neg\varphi_2 \vee$ $\neg\varphi_{k\text{-}1}$ $\vee$ $\neg\varphi_k$
property
transition system
M_0 M_1 M_2 $M_{k\text{-}1}$ M_k
bound
counterexample trace

- Transition system *M* unrolled *k* times
  - for programs: loops, recursion, …
- Translated into verification condition ψ such that

ψ satisfiable iff φ has counterexample of max. depth *k*

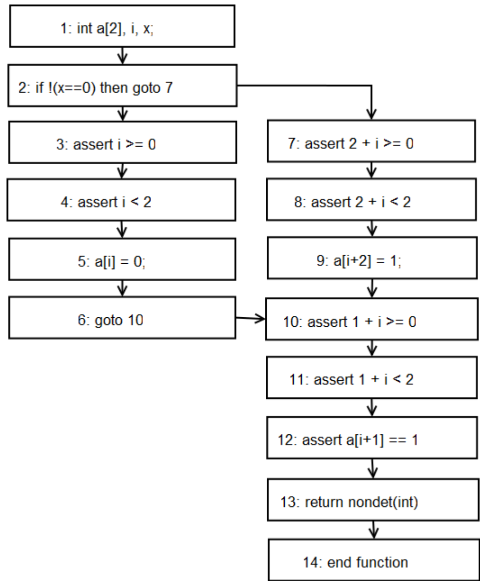
BMC has been applied successfully to verify HW and SW

Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

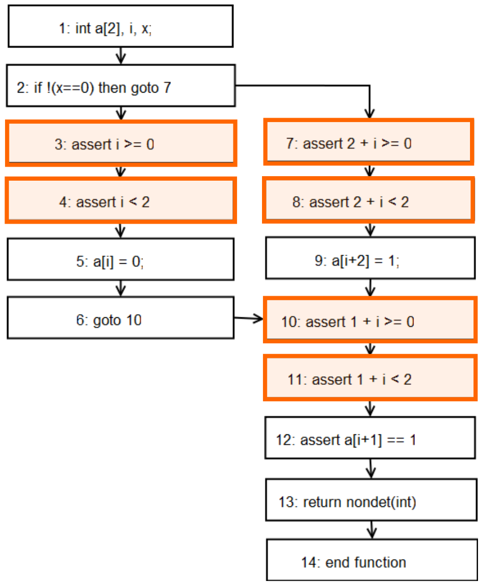


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

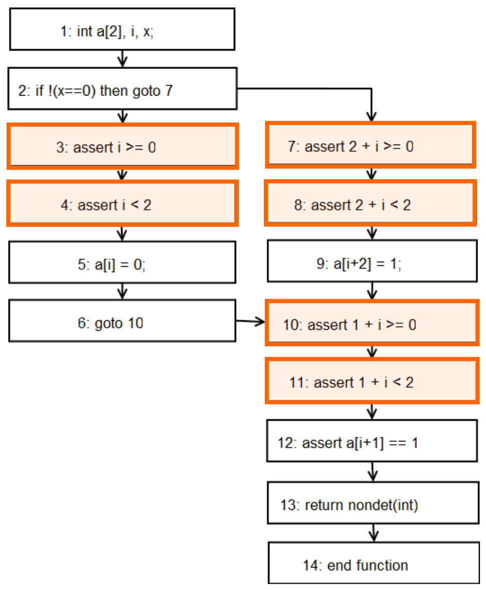


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```



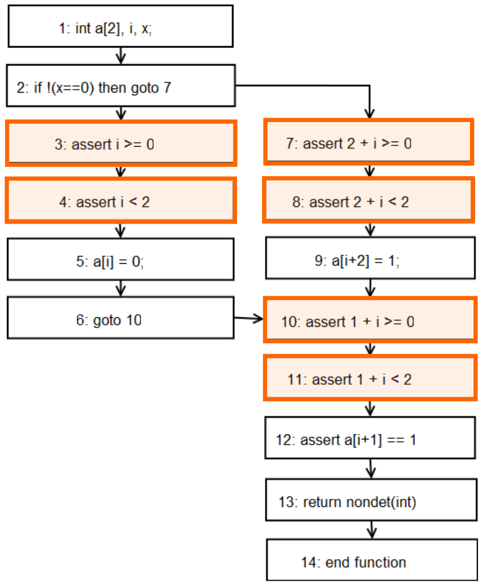
Software BMC

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ crucial



Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  crucial
- front-end converts unrolled and **optimized program into SSA**

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

$$
\begin{aligned}
g_1 &= x_1 == 0 \\
a_1 &= a_0 \text{ WITH } [i_0 := 0] \\
a_2 &= a_0 \\
a_3 &= a_2 \text{ WITH } [2+i_0 := 1] \\
a_4 &= g_1 \; ? \; a_1 : a_3 \\
t_1 &= a_4 [1+i_0] == 1
\end{aligned}
$$

Software BMC

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  } crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Software BMC

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  } crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories

$$C := \left[\begin{array}{l} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{array}\right]$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{array}\right]$$

Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  $\Big\}$ crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of *C* ∧ ¬*P*

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

$$C := \left[\begin{array}{l} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{array}\right]$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{array}\right]$$

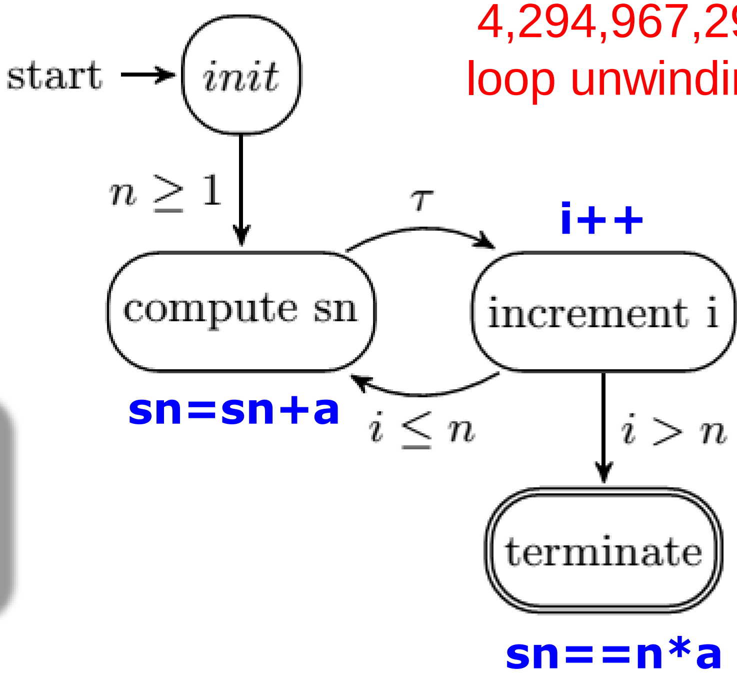
Difficulties in proving the correctness of programs with loops in BMC

- BMC techniques can falsify properties up to a given depth *k*
  - prove correctness if an upper bound of *k* is known (**unwinding assertion**)
    - » BMC tools typically fail to verify programs that contain bounded and unbounded loops

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1$$

the loop will be unfolded 2^{n-1} times (in the worst case, 2^{32-1} times on 32 bits integer)

4,294,967,295 loop unwindings



Induction-Based Verification for Software

***k*-induction** checks loop-free programs...

- **base case (*base*$_k$):** find a counter-example with up to *k* loop unwindings (plain BMC)
- **forward condition (*fwd*$_k$):** check that *P* holds in all states reachable within *k* unwindings
- **inductive step (*step*$_k$):** check that whenever *P* holds for *k* unwindings, it also holds after next unwinding
  - havoc state
  - run *k* iterations
  - assume invariant
  - run final iteration

⇒ iterative deepening if inconclusive

Induction-Based Verification for Software

```
k=1
while k<=max_iterations do
    if $base_{P,\phi,k}$ then
        return trace s[0..k]
    else
        k=k+1
        if $fwd_{P,\phi,k}$ then
          return true
        else if $step_{P',\phi,k}$ then
          return true
    end if
end
return unknown
```

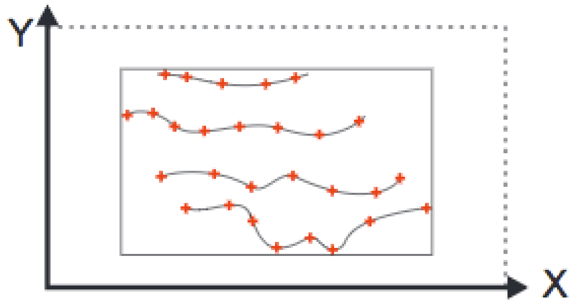
```
unsigned int x=*;
while(x>0) x--;
assume(x<=0);
assert(x==0);
```

```
unsigned int x=*;
while(x>0) x--;
assert(x<=0);
assert(x==0);
```

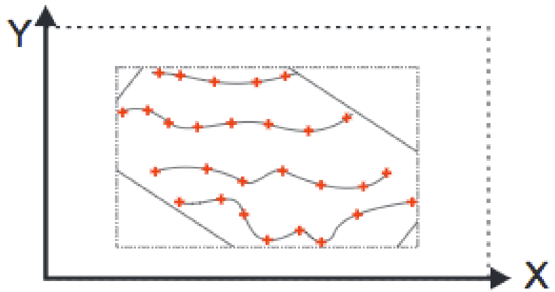
```
unsigned int x=*;
assume(x>0);
while(x>0) x--;
assume(x<=0);
assert(x==0);
```

Automatic Invariant Generation

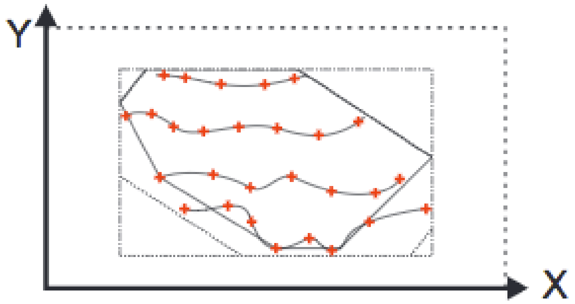
- infer invariants using **intervals**, **octagons**, and **convex polyhedral** constraints for the inductive step
  - *e.g., a* ≤ *x* ≤ *b*; *x* ≤ *a, x-y* ≤ *b*; and *ax* + *by* ≤ *c*



intervals



octagons



convex polyhedral

- use existing libraries to discover linear/polynomial relations among integer/real variables to infer **loop invariants**
  - compute **pre-** and **post-conditions**

Verifying Multi-threaded Programs

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

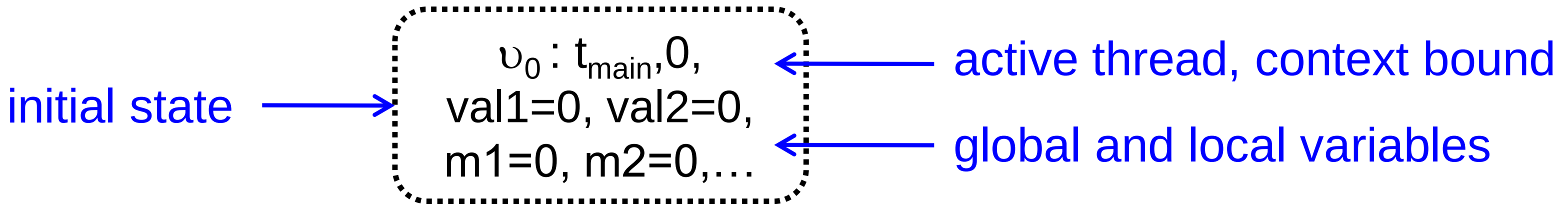
- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings

```
void *threadA(void *arg) {
  lock(&mutex);
  x++;
  if (x == 1) lock(&lock);
  unlock(&mutex); (CS1)
  lock(&mutex);   (CS3)
  x--;
  if (x == 0) unlock(&lock);
  unlock(&mutex);
}
```

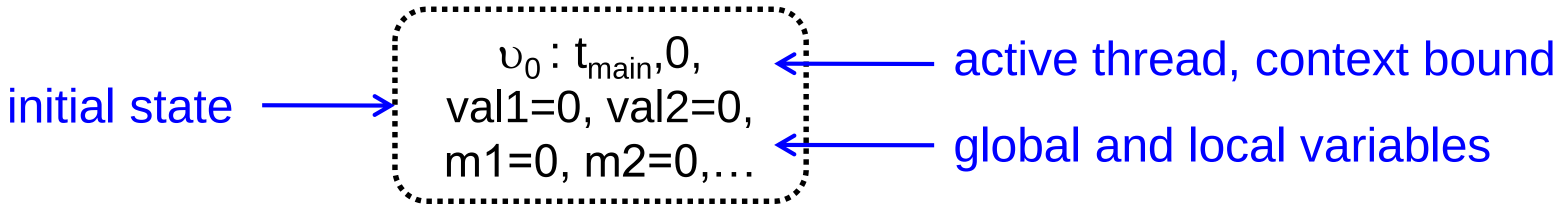
Deadlock

```
void *threadB(void *arg) {
  lock(&mutex);
  y++;
  if (y == 1) lock(&lock); (CS2)
  unlock(&mutex);
  lock(&mutex);
  y--;
  if (y == 0) unlock(&lock);
  unlock(&mutex);
}
```

Lazy Exploration of the Reachability Tree


$\upsilon_0 : t_{main}, 0,$
val1=0, val2=0,
m1=0, m2=0,…
initial state
active thread, context bound
global and local variables

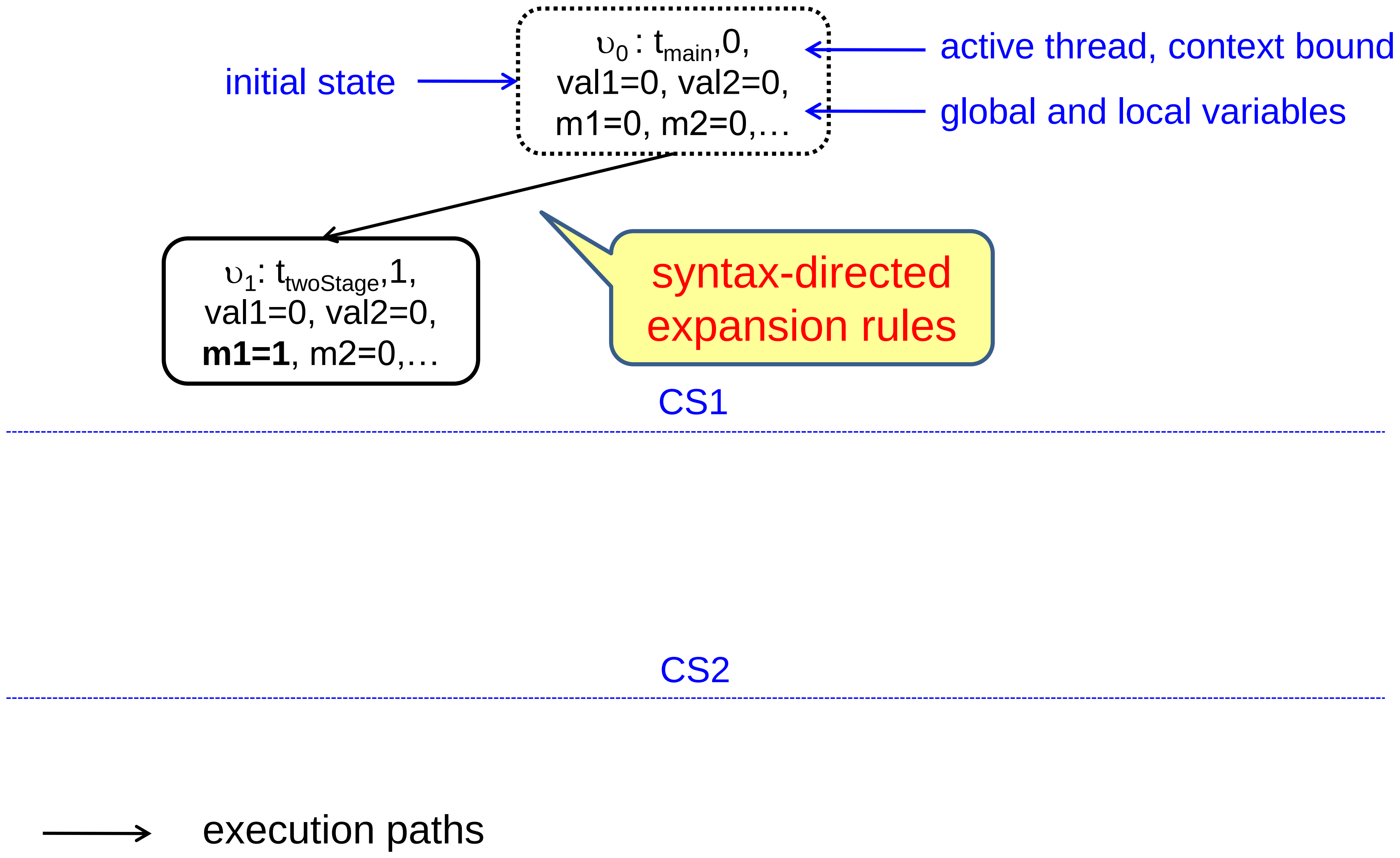
Lazy Exploration of the Reachability Tree


υ_0 : t_{main},0,
val1=0, val2=0,
m1=0, m2=0,…
initial state
active thread, context bound
global and local variables

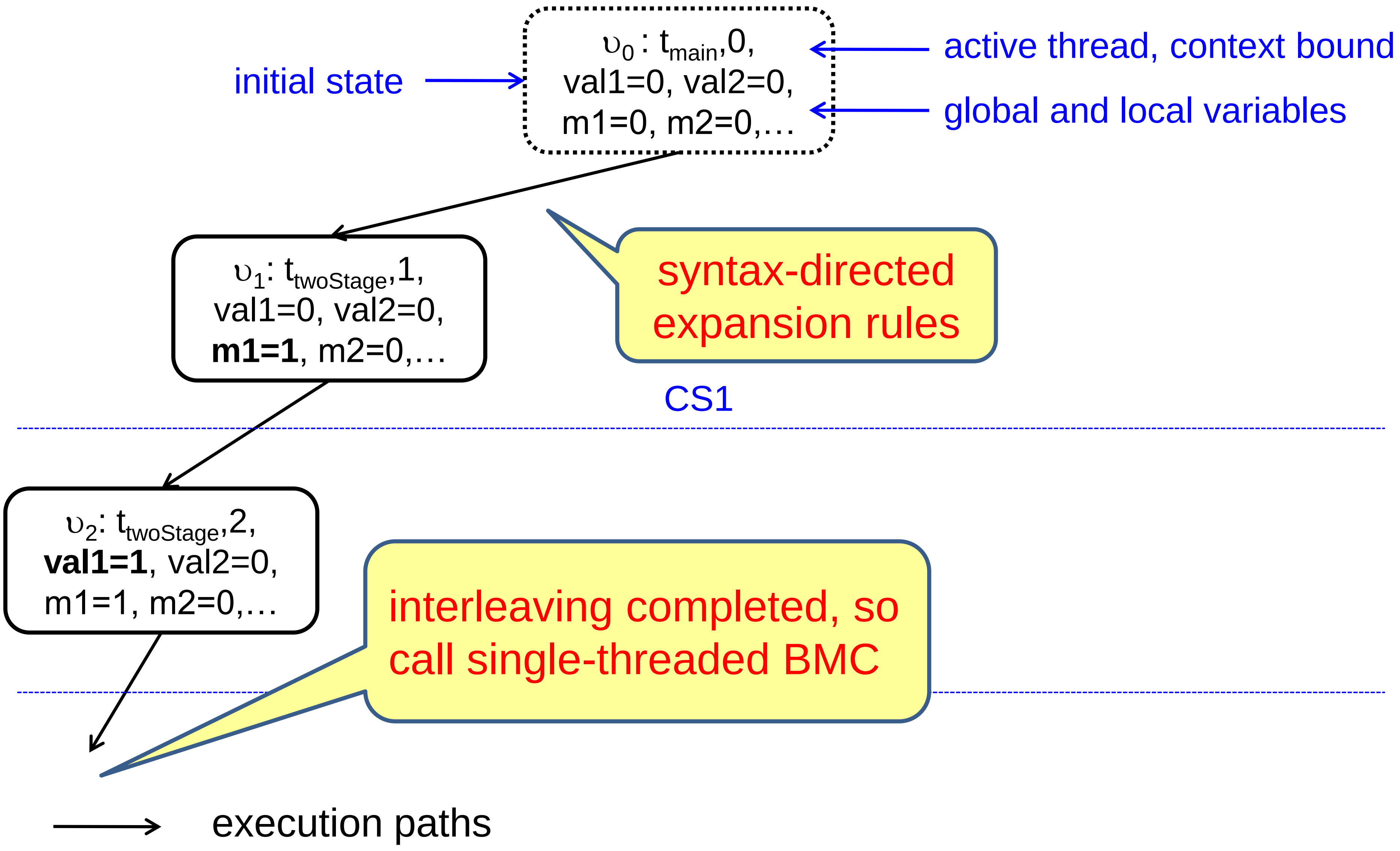
CS1

CS2

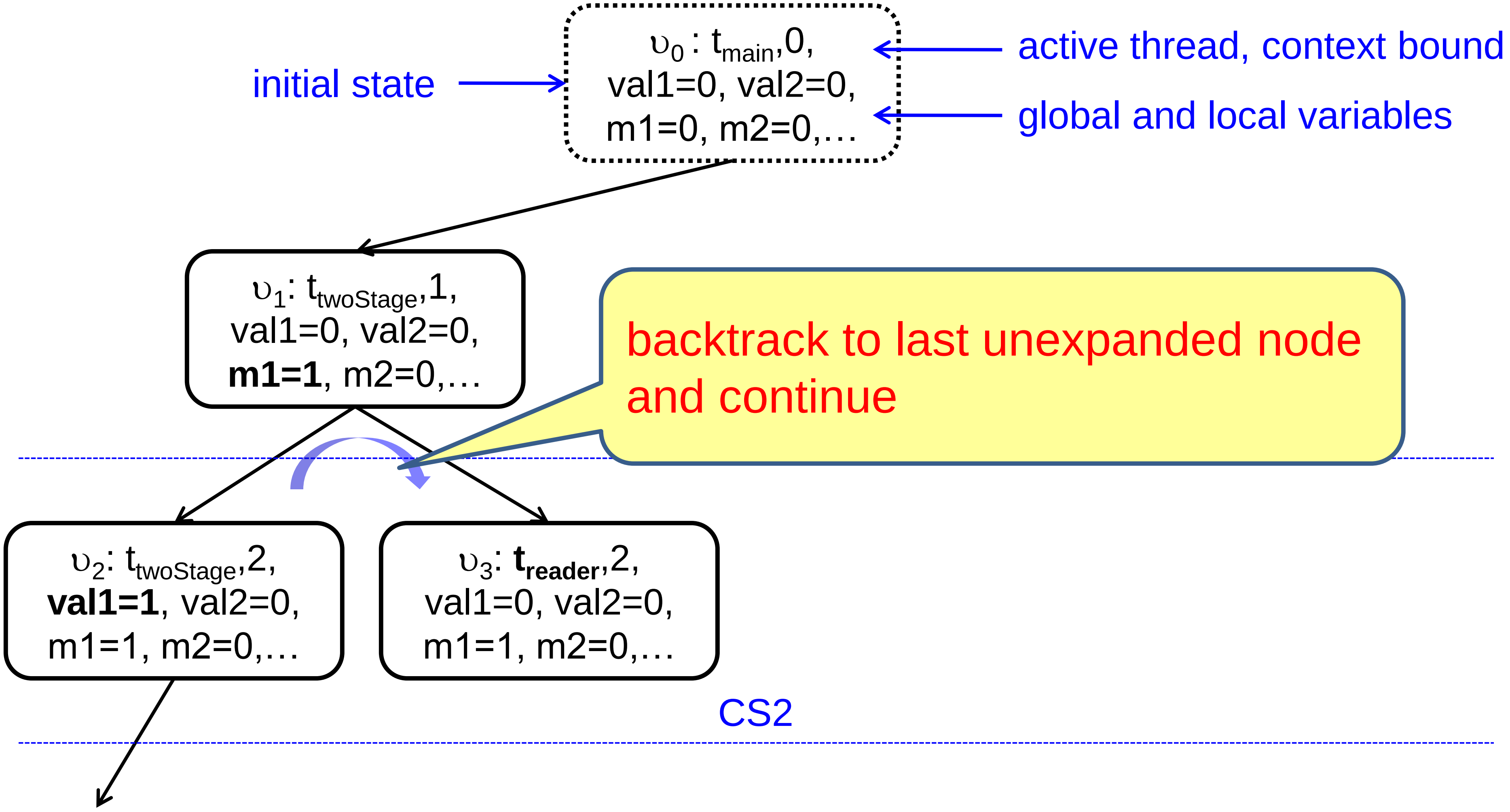
Lazy Exploration of the Reachability Tree


initial state
υ_0 : t_{main},0,
val1=0, val2=0,
m1=0, m2=0,…
active thread, context bound
global and local variables
υ_1: $t_{twoStage}$,1,
val1=0, val2=0,
m1=1, m2=0,…
syntax-directed expansion rules
CS1
CS2
execution paths

Lazy Exploration of the Reachability Tree


initial state
υ_0 : t_{main},0,
val1=0, val2=0,
m1=0, m2=0,…
active thread, context bound
global and local variables
υ_1: $t_{twoStage}$,1,
val1=0, val2=0,
m1=1, m2=0,…
syntax-directed expansion rules
CS1
υ_2: $t_{twoStage}$,2,
val1=1, val2=0,
m1=1, m2=0,…
interleaving completed, so call single-threaded BMC
execution paths

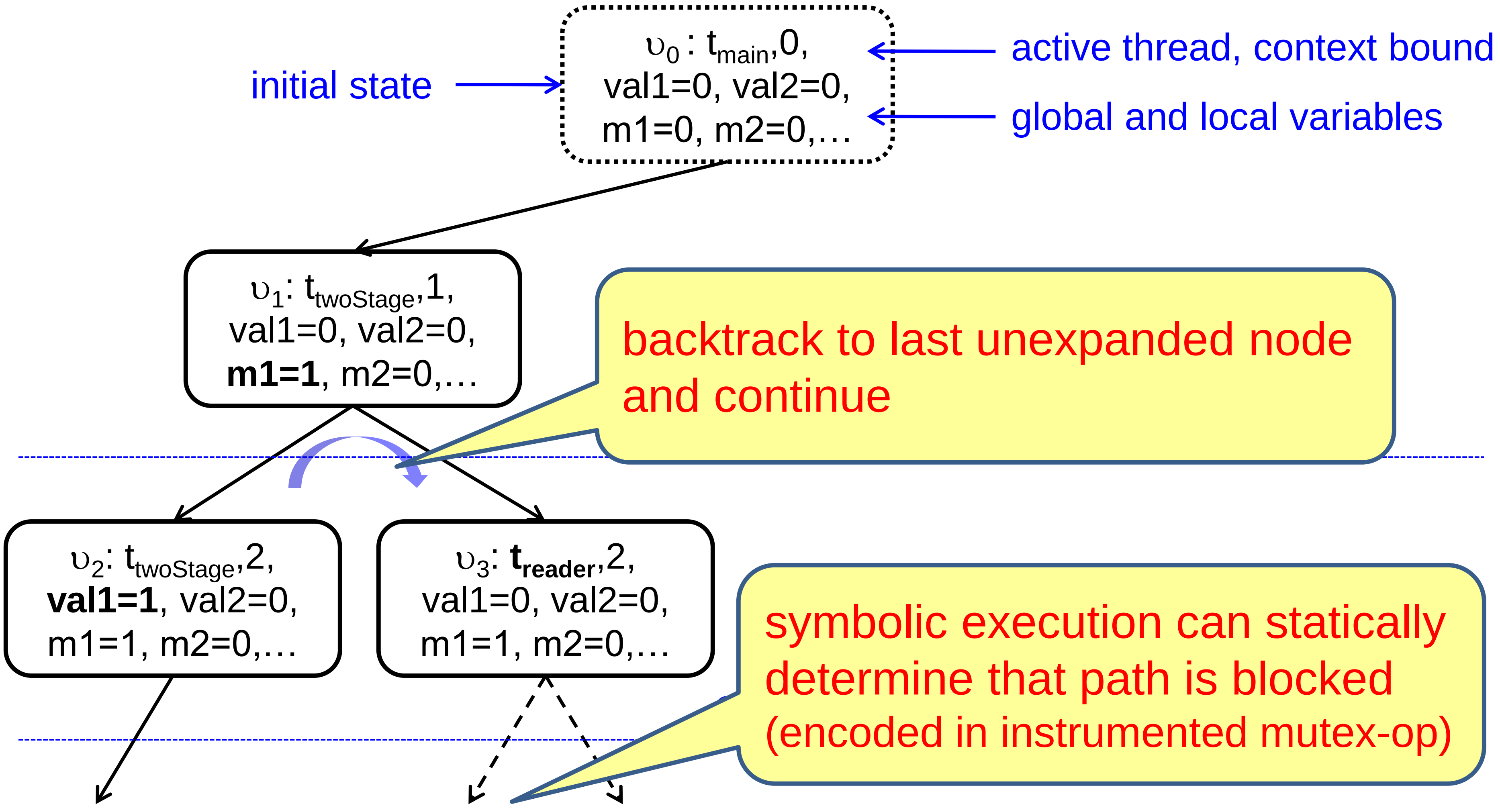
Lazy Exploration of the Reachability Tree


initial state
active thread, context bound
global and local variables
υ_0 : t_{main},0, val1=0, val2=0, m1=0, m2=0,…
υ_1: $t_{twoStage}$,1, val1=0, val2=0, **m1=1**, m2=0,…
backtrack to last unexpanded node and continue
υ_2: $t_{twoStage}$,2, **val1=1**, val2=0, m1=1, m2=0,…
υ_3: $\mathbf{t_{reader}}$,2, val1=0, val2=0, m1=1, m2=0,…
CS2

$\longrightarrow$ execution paths

$\dashrightarrow$ blocked execution paths (*eliminated*)

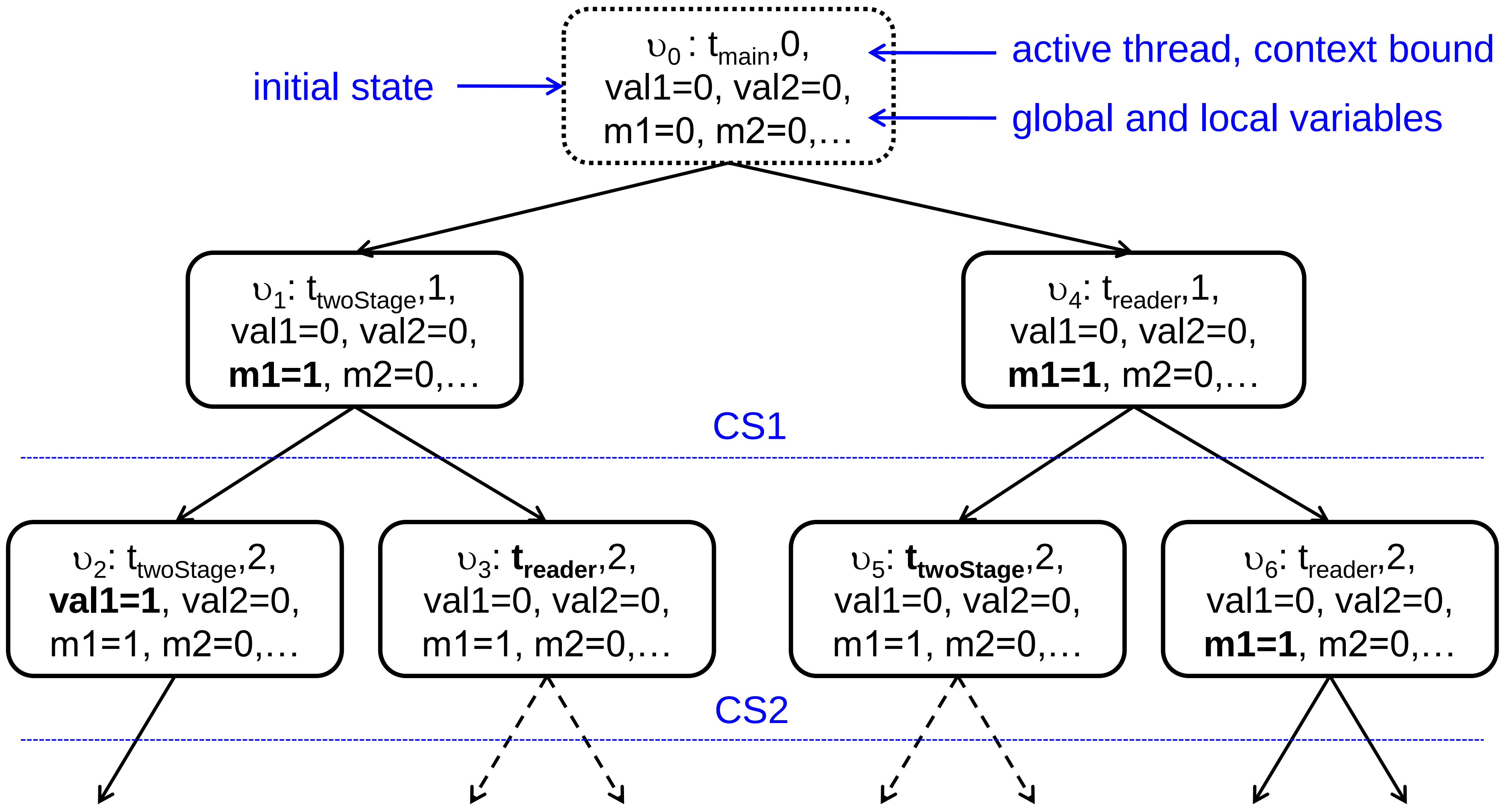
Lazy Exploration of the Reachability Tree


initial state
active thread, context bound
global and local variables
υ_0 : t_{main},0, val1=0, val2=0, m1=0, m2=0,…
υ_1: $t_{twoStage}$,1, val1=0, val2=0, **m1=1**, m2=0,…
backtrack to last unexpanded node and continue
υ_2: $t_{twoStage}$,2, **val1=1**, val2=0, m1=1, m2=0,…
υ_3: $\mathbf{t_{reader}}$,2, val1=0, val2=0, m1=1, m2=0,…
symbolic execution can statically determine that path is blocked (encoded in instrumented mutex-op)

$\longrightarrow$ execution paths

$\dashrightarrow$ blocked execution paths (*eliminated*)

Lazy Exploration of the Reachability Tree



initial state
active thread, context bound
global and local variables
υ_0 : t_{main},0, val1=0, val2=0, m1=0, m2=0,…
υ_1: $t_{twoStage}$,1, val1=0, val2=0, **m1=1**, m2=0,…
υ_4: t_{reader},1, val1=0, val2=0, **m1=1**, m2=0,…
CS1
υ_2: $t_{twoStage}$,2, **val1=1**, val2=0, m1=1, m2=0,…
υ_3: $\mathbf{t_{reader}}$,2, val1=0, val2=0, m1=1, m2=0,…
υ_5: $\mathbf{t_{twoStage}}$,2, val1=0, val2=0, m1=1, m2=0,…
υ_6: t_{reader},2, val1=0, val2=0, **m1=1**, m2=0,…
CS2

$\longrightarrow$ execution paths

$\dashrightarrow$ blocked execution paths (*eliminated*)

Lazy exploration of interleavings

- Main steps of the algorithm:

  1. Initialize the stack with the initial node ν_0 and the initial path $\pi_0 = \langle \upsilon_0 \rangle$

  2. If the stack is empty, terminate with "no error".

  3. Pop the current node υ and current path π off the stack and compute the set υ' of successors of υ using rules R1-R8.

  4. If υ' is empty, derive the VC φ_k^{π} for π and call the SMT solver on it. If φ_k^{π} is satisfiable, terminate with "error"; otherwise, goto step 2.

  5. If υ' is not empty, then for each node $\upsilon \in \upsilon'$, add ν to π, and push node and extended path on the stack. goto step 3.

computation path

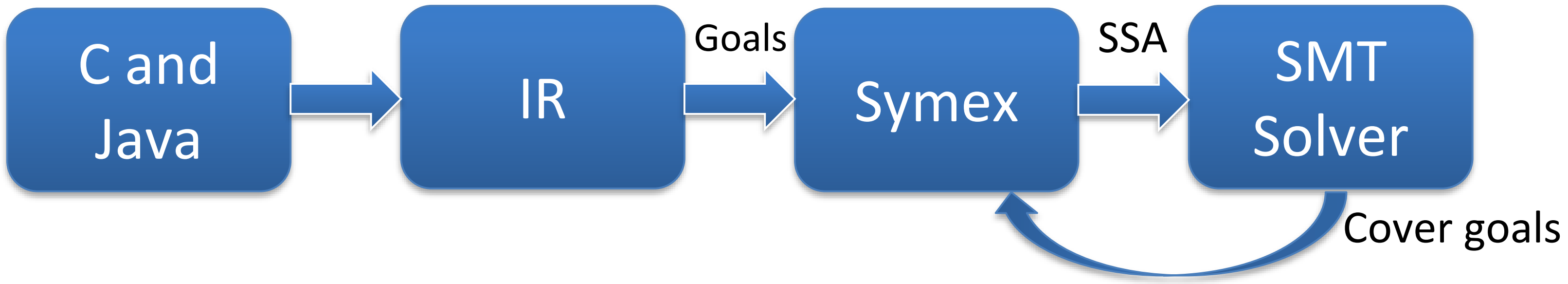
$\pi = \{\upsilon_1, \ldots \upsilon_n\}$

$$\varphi_k^{\pi} = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{\text{constraint s}} \wedge \overbrace{\neg \phi_k}^{\text{property}}$$

bound

BMC / SE for Coverage Test Generation

- Translate the program to an **intermediate representation** (IR)
- Add goals indicating the **coverage**
  - *location, branch, decision, condition and path*
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover different goals
- Interpret the solution to figure out the **input conditions**
- Spit those input conditions out as a test case



Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

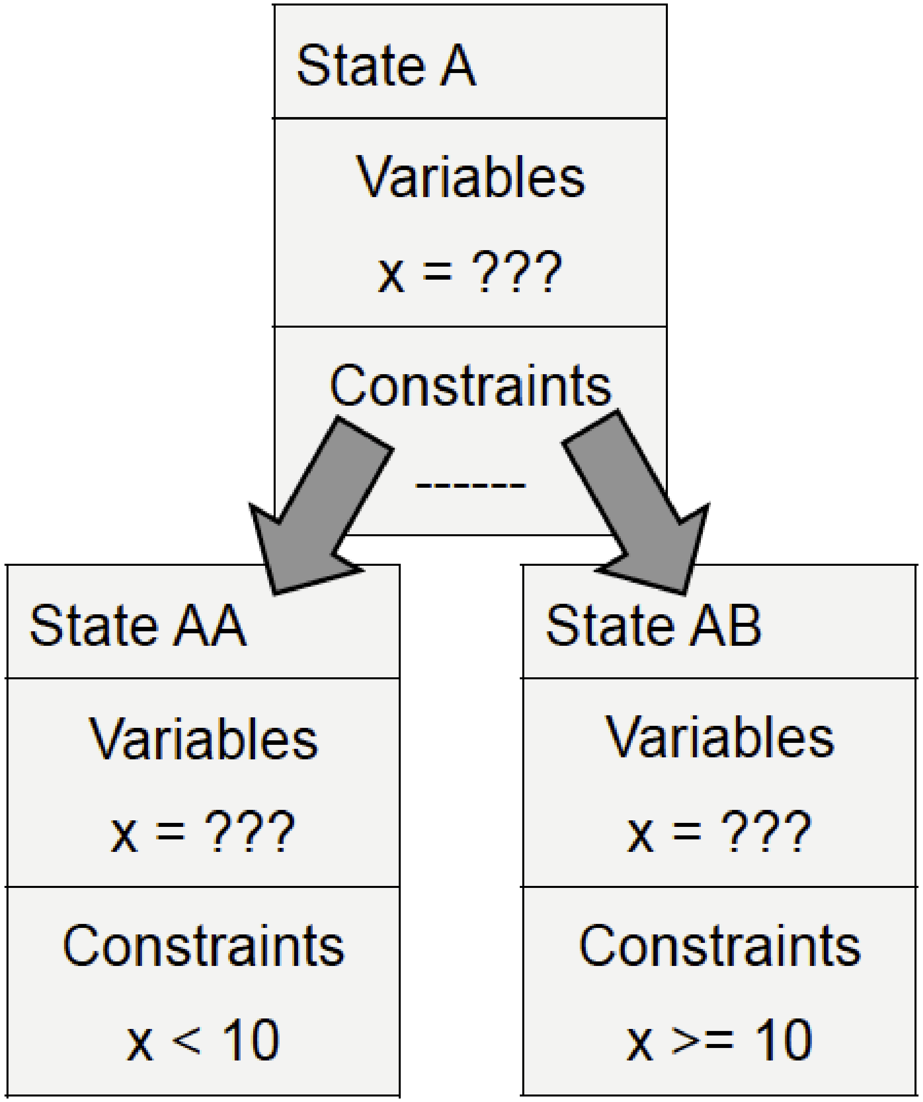
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

State A
Variables x = ???
Constraints ------

Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



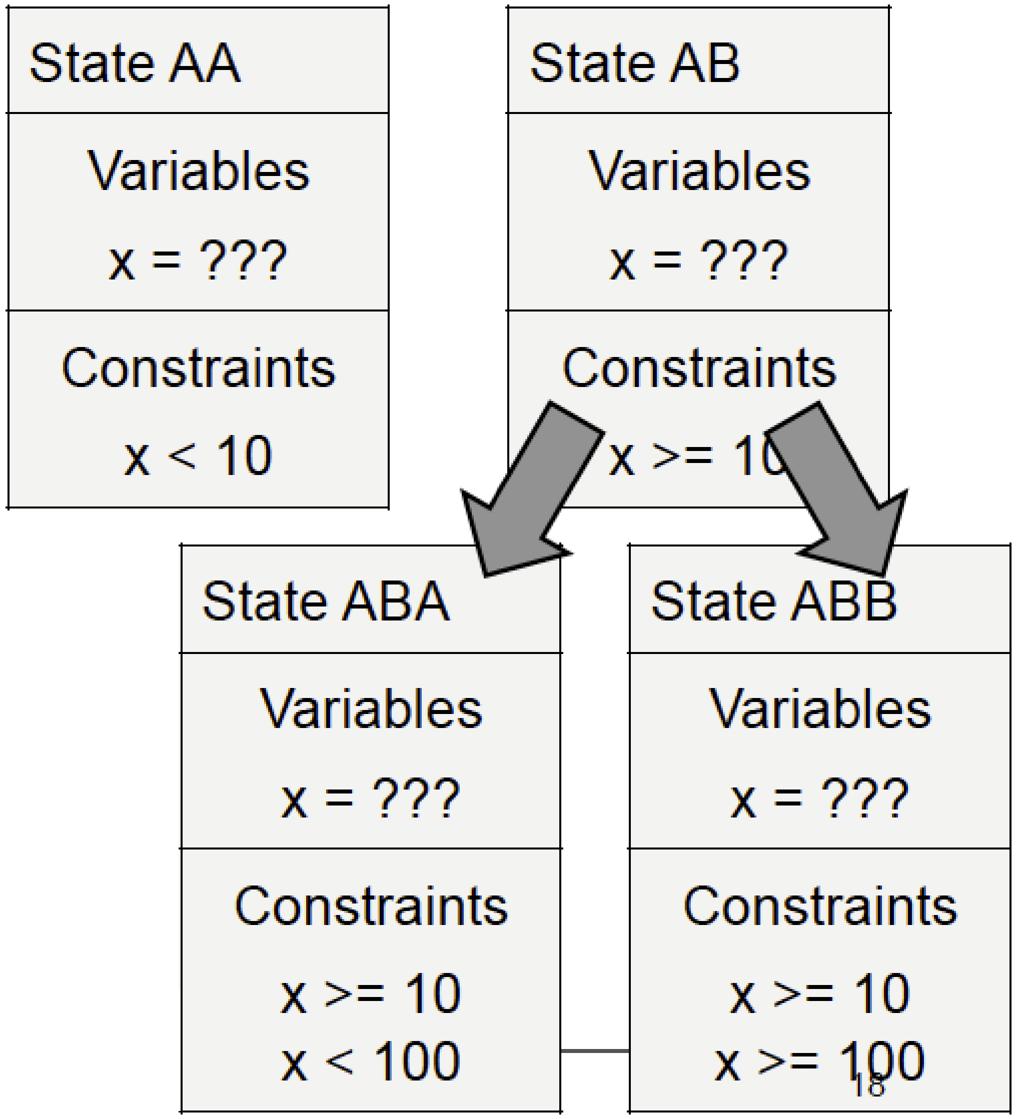
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

State AA	State AB
Variables	Variables
x = ???	x = ???
Constraints	Constraints
x < 10	x >= 10

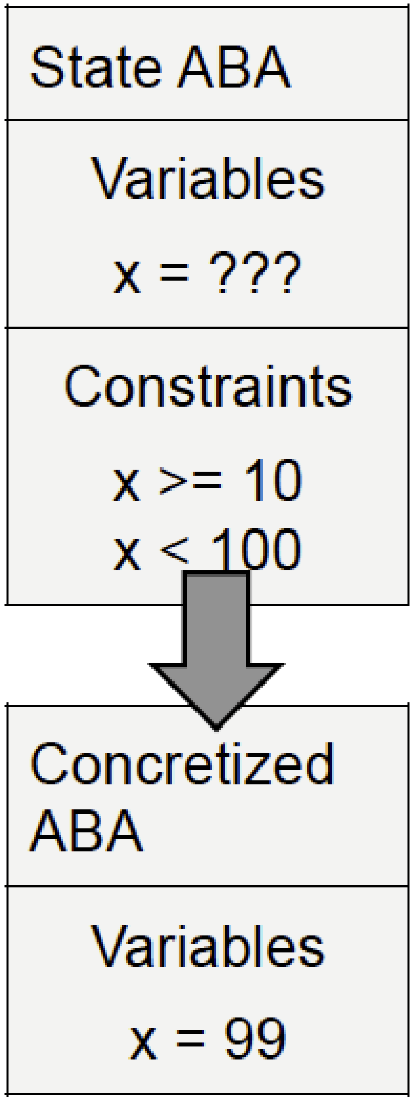
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



Coverage Test Generation for Security

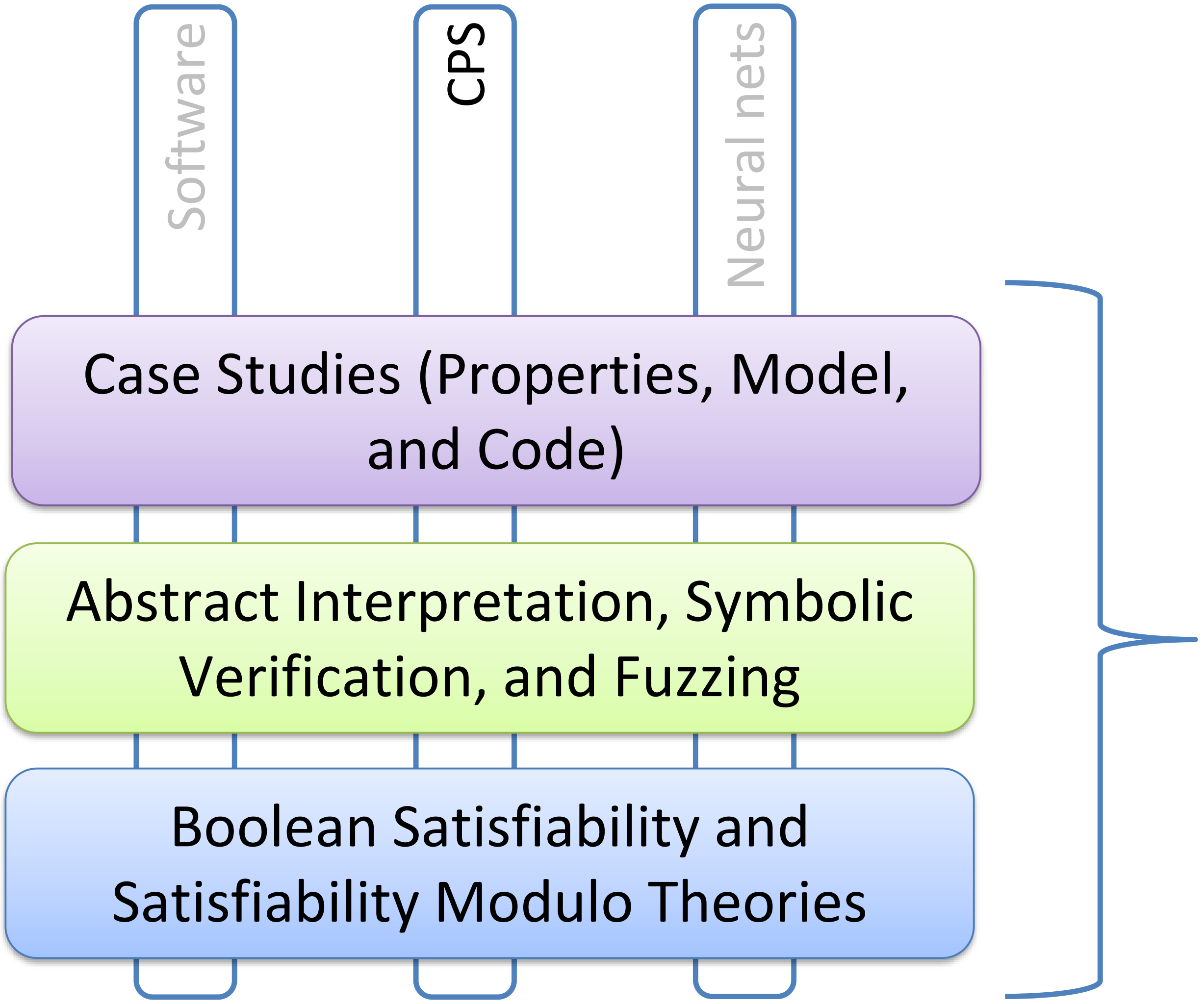
```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



Achievements

- Distinguished Paper Award at ACM ICSE’11 (acceptance rate 14%)
- Best Paper Award at SBC SBESC’15 (acceptance rate 24%)
- 25 awards from the international competitions on software verification (SV-COMP) 2012-2020 and testing (Test-COMP) 2019-2020
  - Overall
  - Falsification Overall
  - Cover-Error

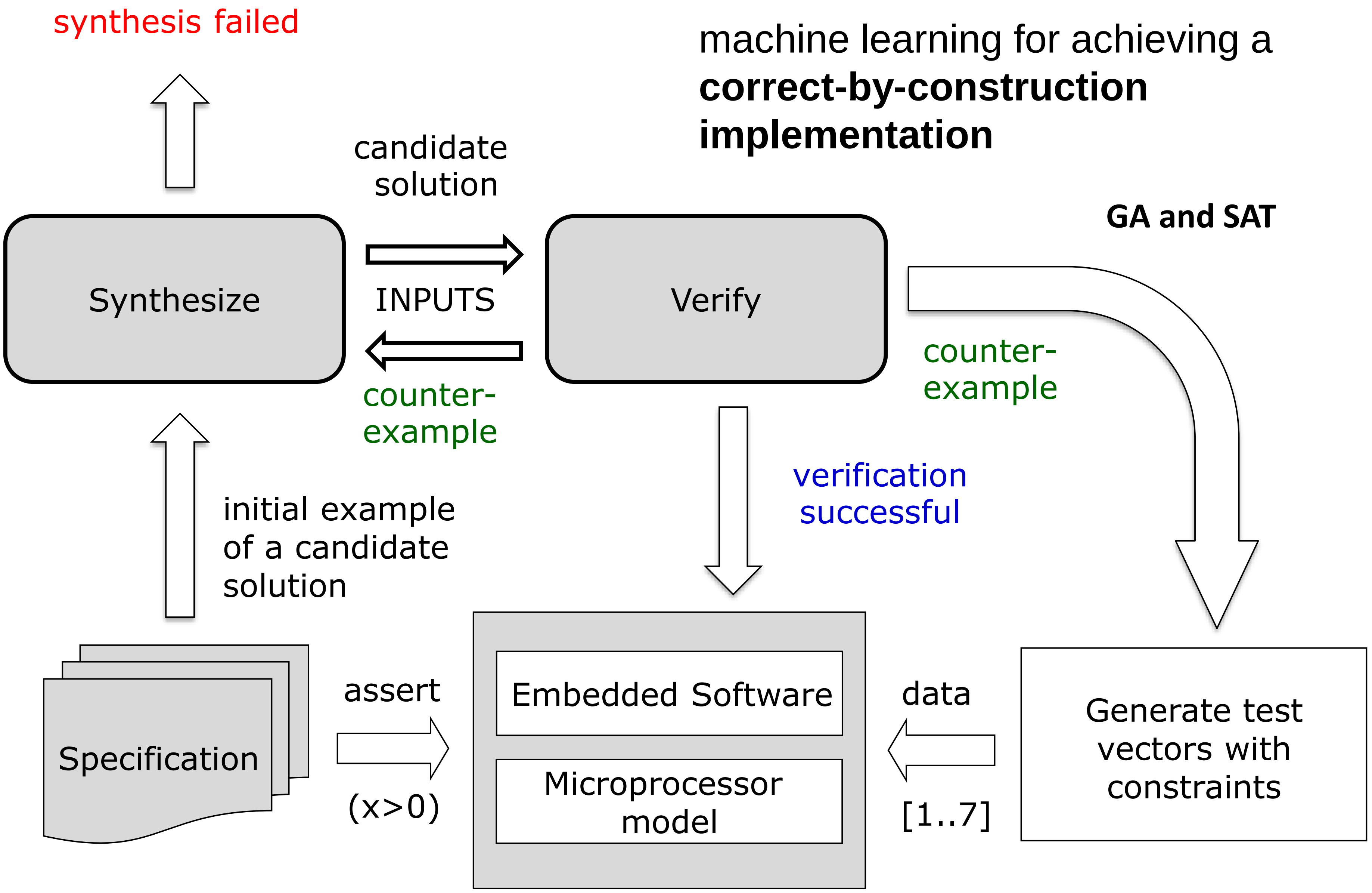
Outline


Software
CPS
Neural nets
Case Studies (Properties, Model, and Code)
Abstract Interpretation, Symbolic Verification, and Fuzzing
Boolean Satisfiability and Satisfiability Modulo Theories

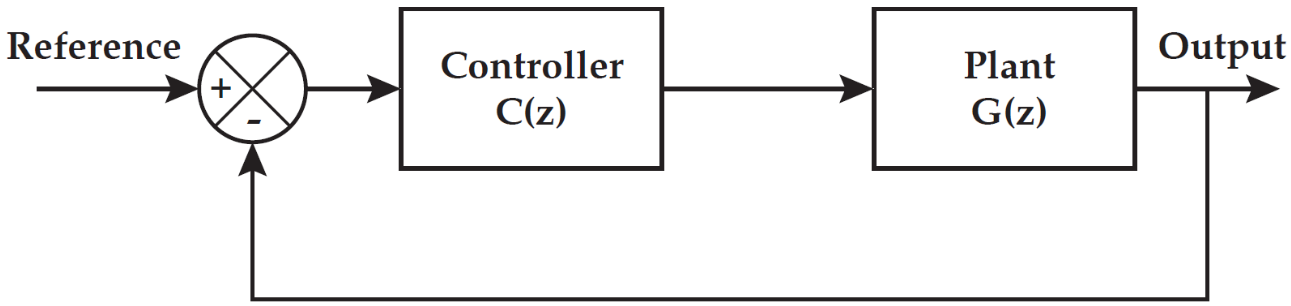
Approaches to formally build verified **trustworthy software systems** to ensure **confidentiality, integrity and availability**

Counter-Example Guided Inductive Synthesis (CEGIS)

machine learning for achieving a **correct-by-construction implementation**

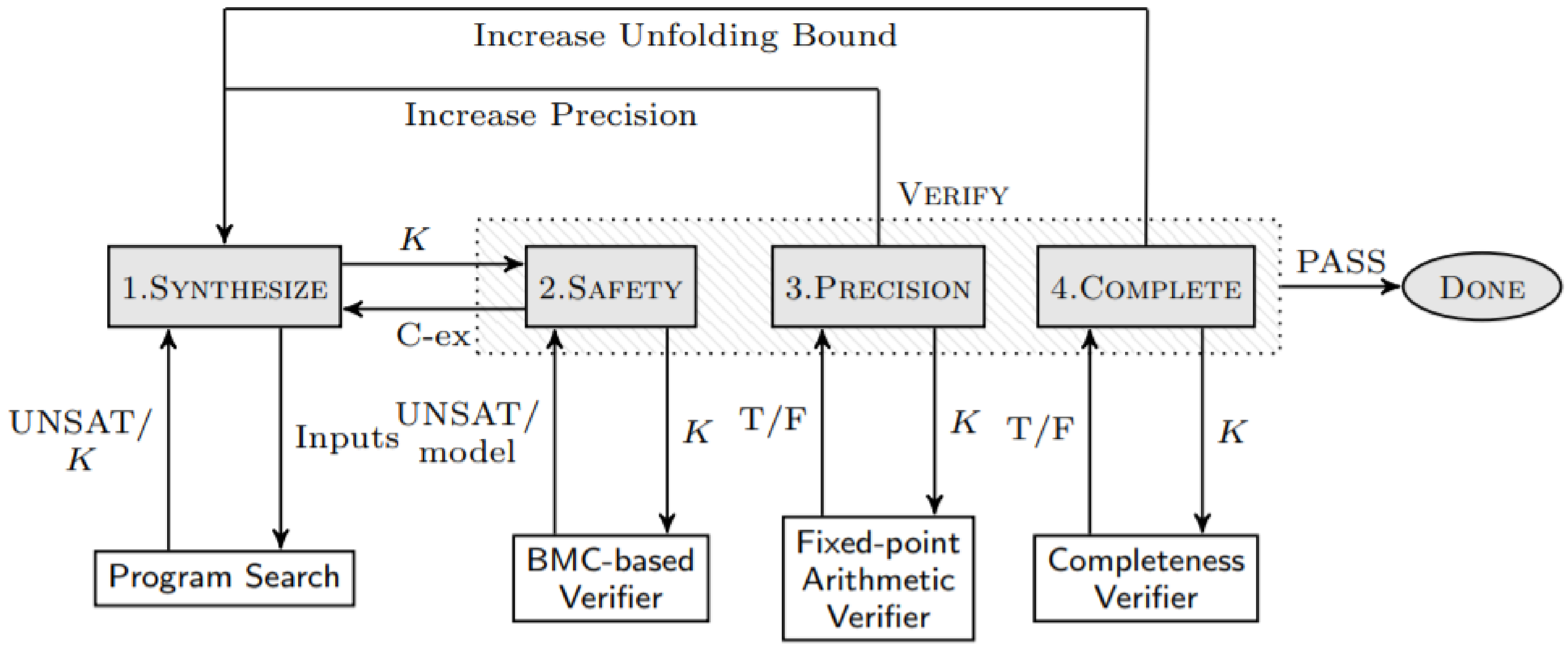


Typical Closed-Loop Control System



- Digital controller and plant representation
  - **state-space:** matrices *A*, *B*, *C*, and *D*
  - **transfer-function:** coefficients b_0, b_1,…,b_m and a_0, a_1,…,a_m
- Stability of closed-loop systems
  - presents a bounded response for any bounded excitation
- Safety of closed-loop systems
  - defines a requirement on the model states
- Numerical errors (truncation and rounding)

CEGIS with multi-staged verification for digital controller synthesis



We synthesise the digital controller *K* for physical plants represented as time-invariant models

Synthesizing Control Software

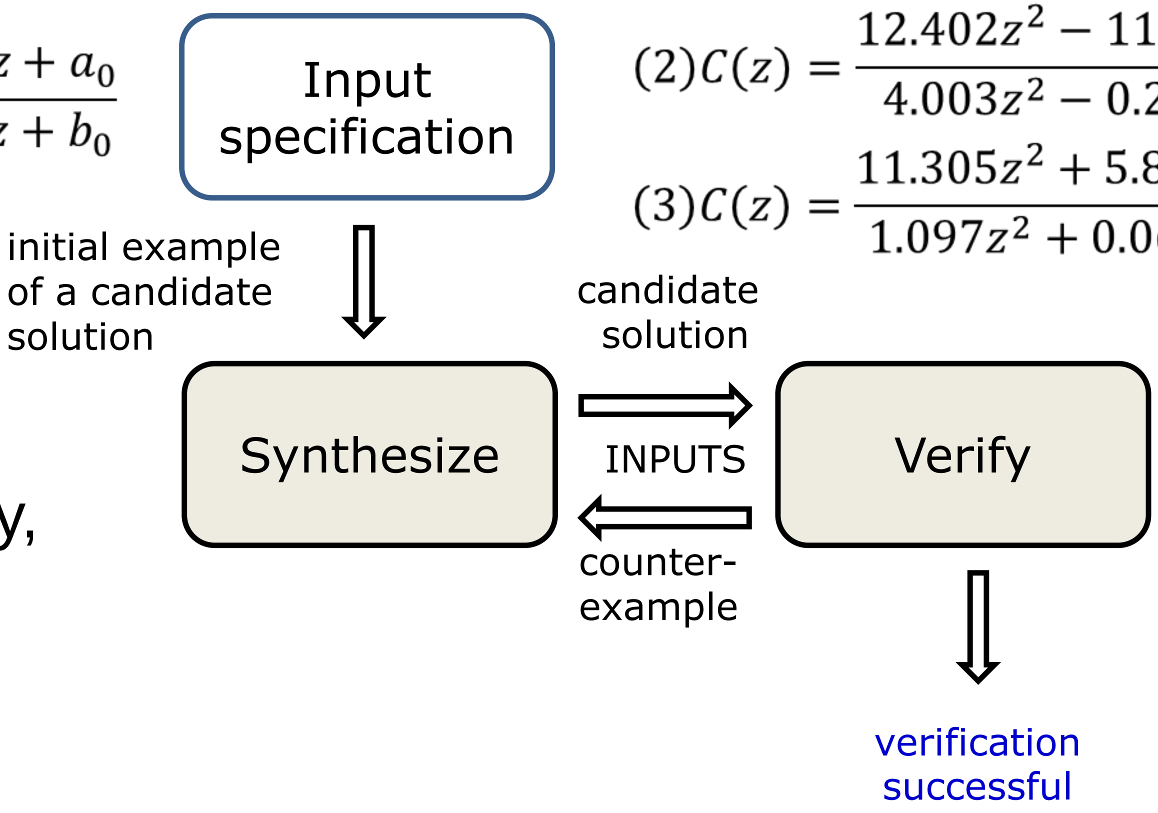
- **Counterexample guided induction synthesis** automates the controller design that is **correct-by-construction**

$$(1)\ C(z) = \frac{0.026}{z + 1.002}$$

$$C(z) = \frac{a_2 z^2 + a_1 z + a_0}{b_2 z^2 + b_1 z + b_0}$$

$$(2) C(z) = \frac{12.402z^2 - 11.439z + 0.596}{4.003z^2 - 0.287z + 0.015}$$

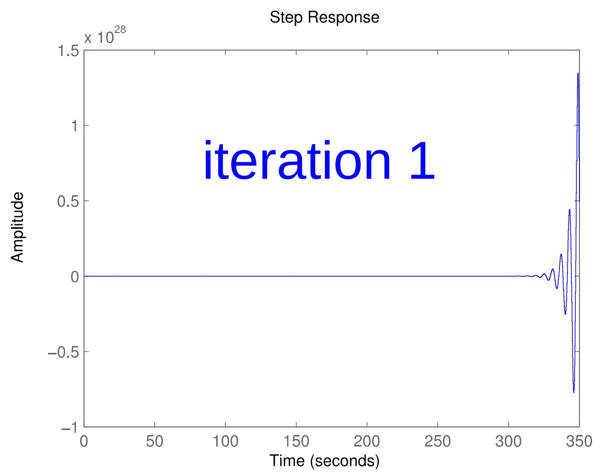
$$(3) C(z) = \frac{11.305z^2 + 5.864z + 4.901}{1.097z^2 + 0.063z + 0.128}$$



stability, safety, performance specifications

Synthesizing Control Software

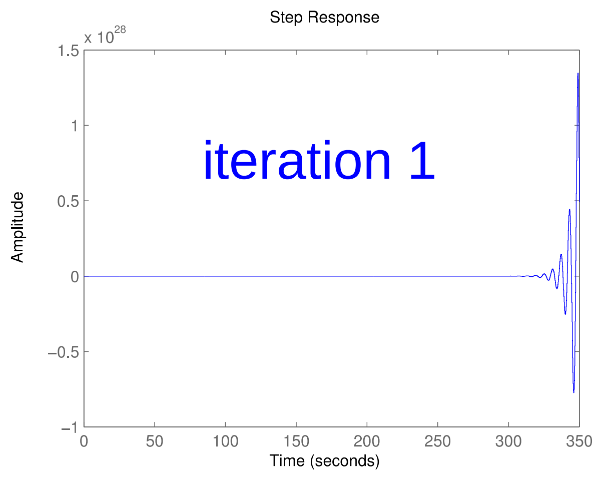
- Step responses for a **closed-loop control system** with **FWL effects** and for each synthesize iteration

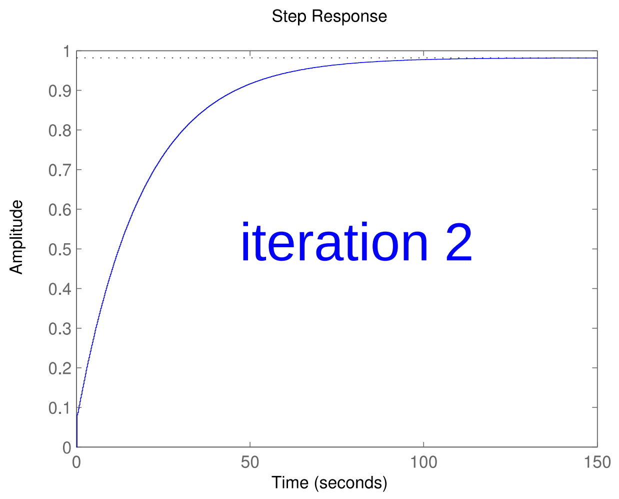


A digital system is **stable** *iff* all of its poles are inside the z-plane unitary circle

Synthesizing Control Software

- Step responses for a **closed-loop control system** with **FWL effects** and for each synthesize iteration

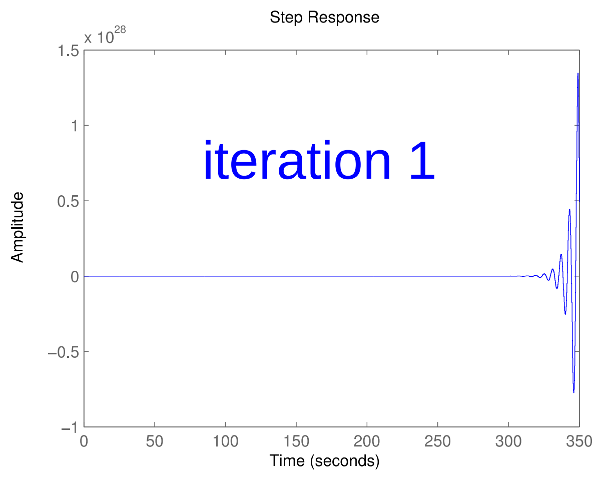


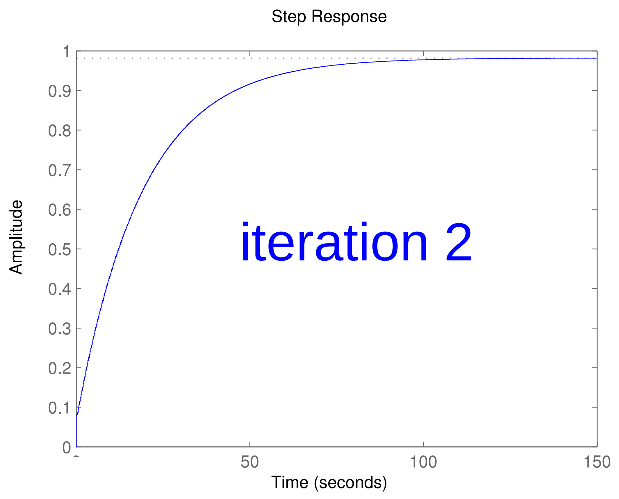


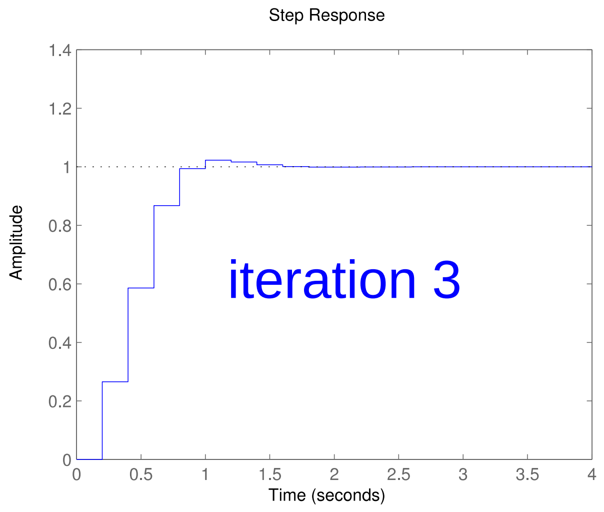
A digital system is **stable** *iff* all of its poles are inside the z-plane unitary circle

Synthesizing Control Software

- Step responses for a **closed-loop control system** with **FWL effects** and for each synthesize iteration

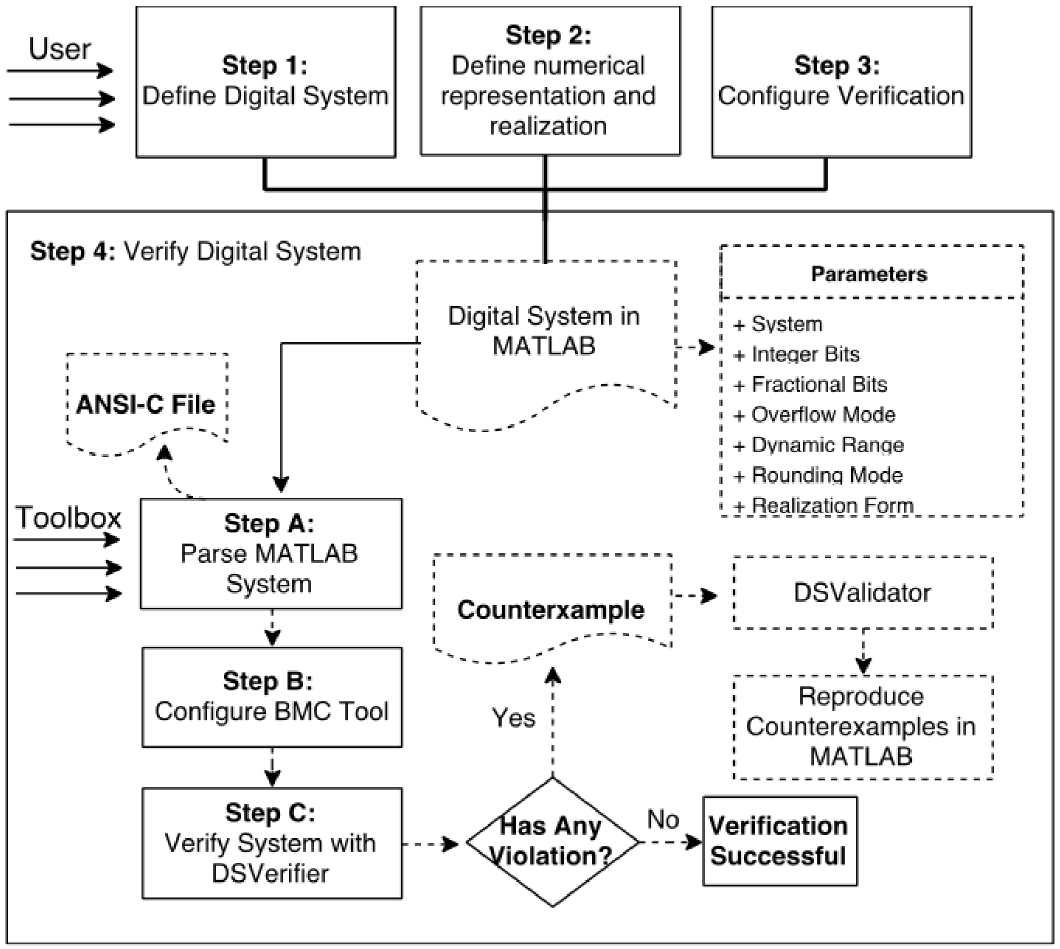






A digital system is **stable** *iff* all of its poles are inside the z-plane unitary circle

DSVerifier Toolbox: BMC tool to check design errors in digital systems with MATLAB

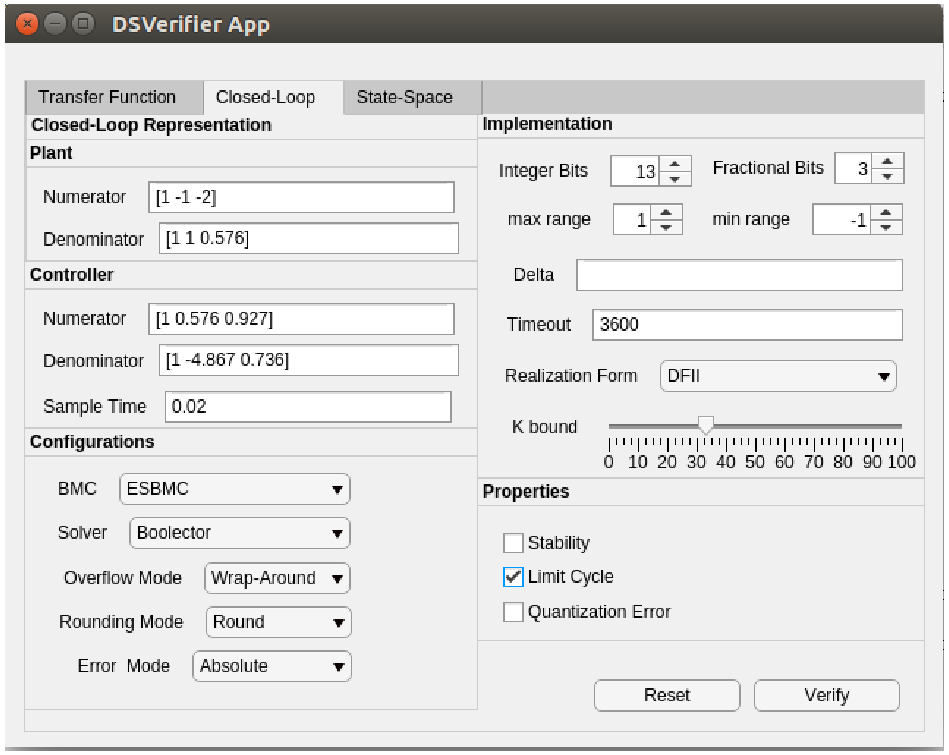

User
Step 1: Define Digital System
Step 2: Define numerical representation and realization
Step 3: Configure Verification
Step 4: Verify Digital System
Digital System in MATLAB
Parameters
+ System
+ Integer Bits
+ Fractional Bits
+ Overflow Mode
+ Dynamic Range
+ Rounding Mode
+ Realization Form
ANSI-C File
Toolbox
Step A: Parse MATLAB System
Step B: Configure BMC Tool
Step C: Verify System with DSVerifier
Counterexample
DSValidator
Yes
Reproduce Counterexamples in MATLAB
Has Any Violation?
No
Verification Successful

CBMC

ESBMC

DSVerifier Toolbox: Illustrative Example

- The different **numerical representations** for a given digital system can yield different **verification results**



DSVerifier App

Transfer Function | Closed-Loop | State-Space

Closed-Loop Representation

Plant

Numerator: [1 -1 -2]

Denominator: [1 1 0.576]

Controller

Numerator: [1 0.576 0.927]

Denominator: [1 -4.867 0.736]

Sample Time: 0.02

Configurations

BMC: ESBMC

Solver: Boolector

Overflow Mode: Wrap-Around

Rounding Mode: Round

Error Mode: Absolute

Implementation

Integer Bits: 13 | Fractional Bits: 3

max range: 1 | min range: -1

Delta:

Timeout: 3600

Realization Form: DFII

K bound: 0 10 20 30 40 50 60 70 80 90 100

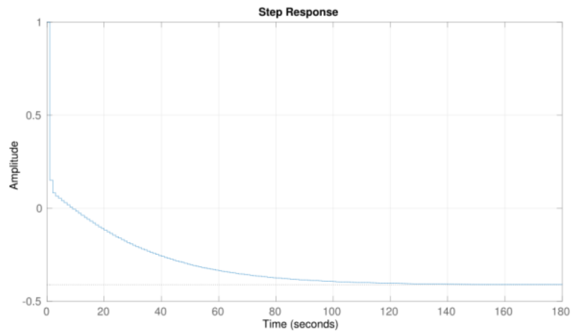
Properties

☐ Stability

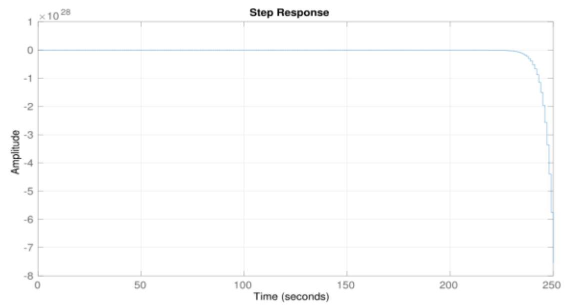
☑ Limit Cycle

☐ Quantization Error

Reset | Verify



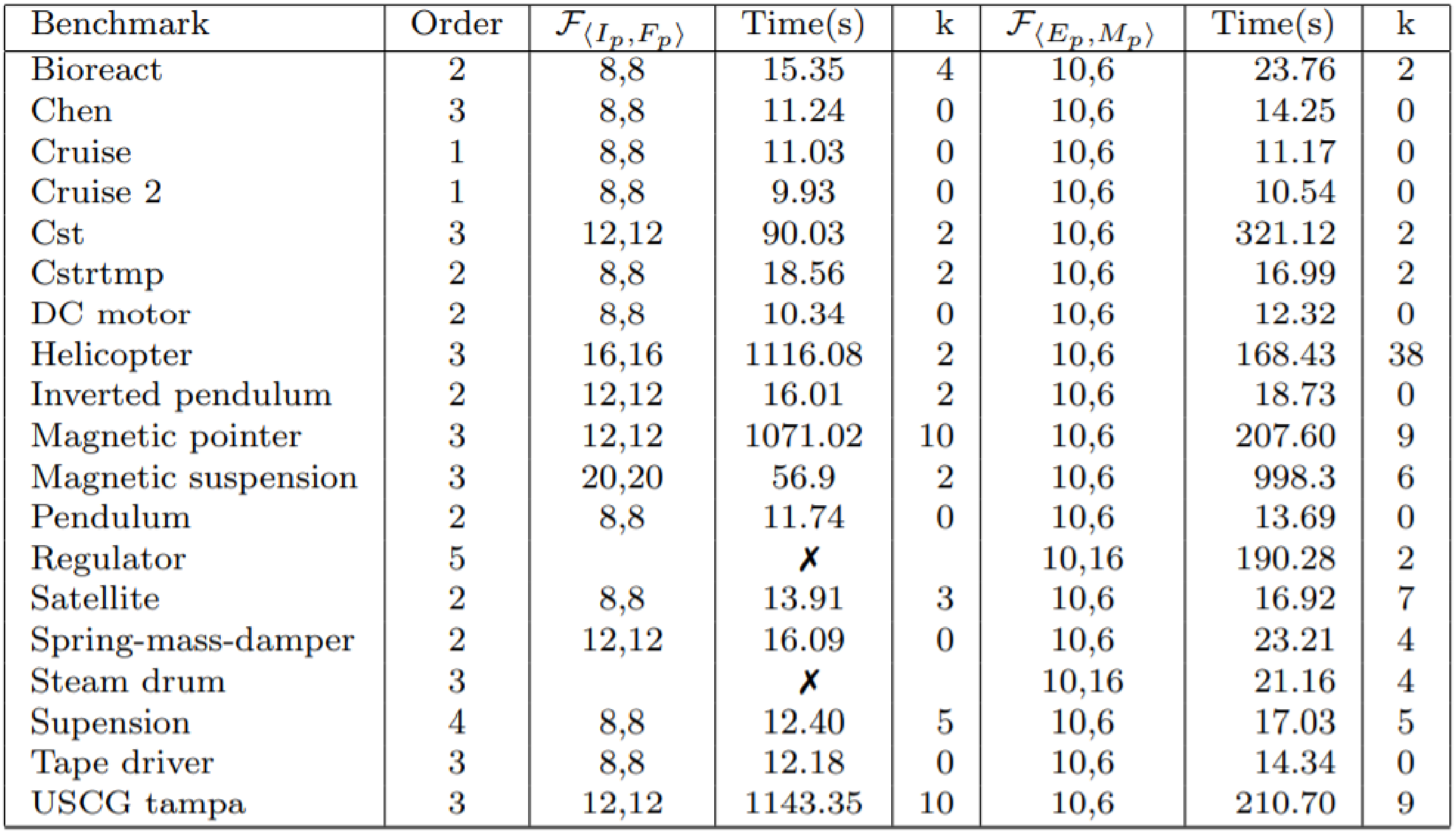
successful verification: **stable** system using <2,13>



http://dsverifier.org/

failed verification: **unstable** system using <13,2>

Synthesis times for fixed- and floating-point controllers



Benchmark	Order	$\mathcal{F}_{\langle I_p,F_p\rangle}$	Time(s)	k	$\mathcal{F}_{\langle E_p,M_p\rangle}$	Time(s)	k
Bioreact	2	8,8	15.35	4	10,6	23.76	2
Chen	3	8,8	11.24	0	10,6	14.25	0
Cruise	1	8,8	11.03	0	10,6	11.17	0
Cruise 2	1	8,8	9.93	0	10,6	10.54	0
Cst	3	12,12	90.03	2	10,6	321.12	2
Cstrtmp	2	8,8	18.56	2	10,6	16.99	2
DC motor	2	8,8	10.34	0	10,6	12.32	0
Helicopter	3	16,16	1116.08	2	10,6	168.43	38
Inverted pendulum	2	12,12	16.01	2	10,6	18.73	0
Magnetic pointer	3	12,12	1071.02	10	10,6	207.60	9
Magnetic suspension	3	20,20	56.9	2	10,6	998.3	6
Pendulum	2	8,8	11.74	0	10,6	13.69	0
Regulator	5		✗		10,16	190.28	2
Satellite	2	8,8	13.91	3	10,6	16.92	7
Spring-mass-damper	2	12,12	16.09	0	10,6	23.21	4
Steam drum	3		✗		10,16	21.16	4
Supension	4	8,8	12.40	5	10,6	17.03	5
Tape driver	3	8,8	12.18	0	10,6	14.34	0
USCG tampa	3	12,12	1143.35	10	10,6	210.70	9

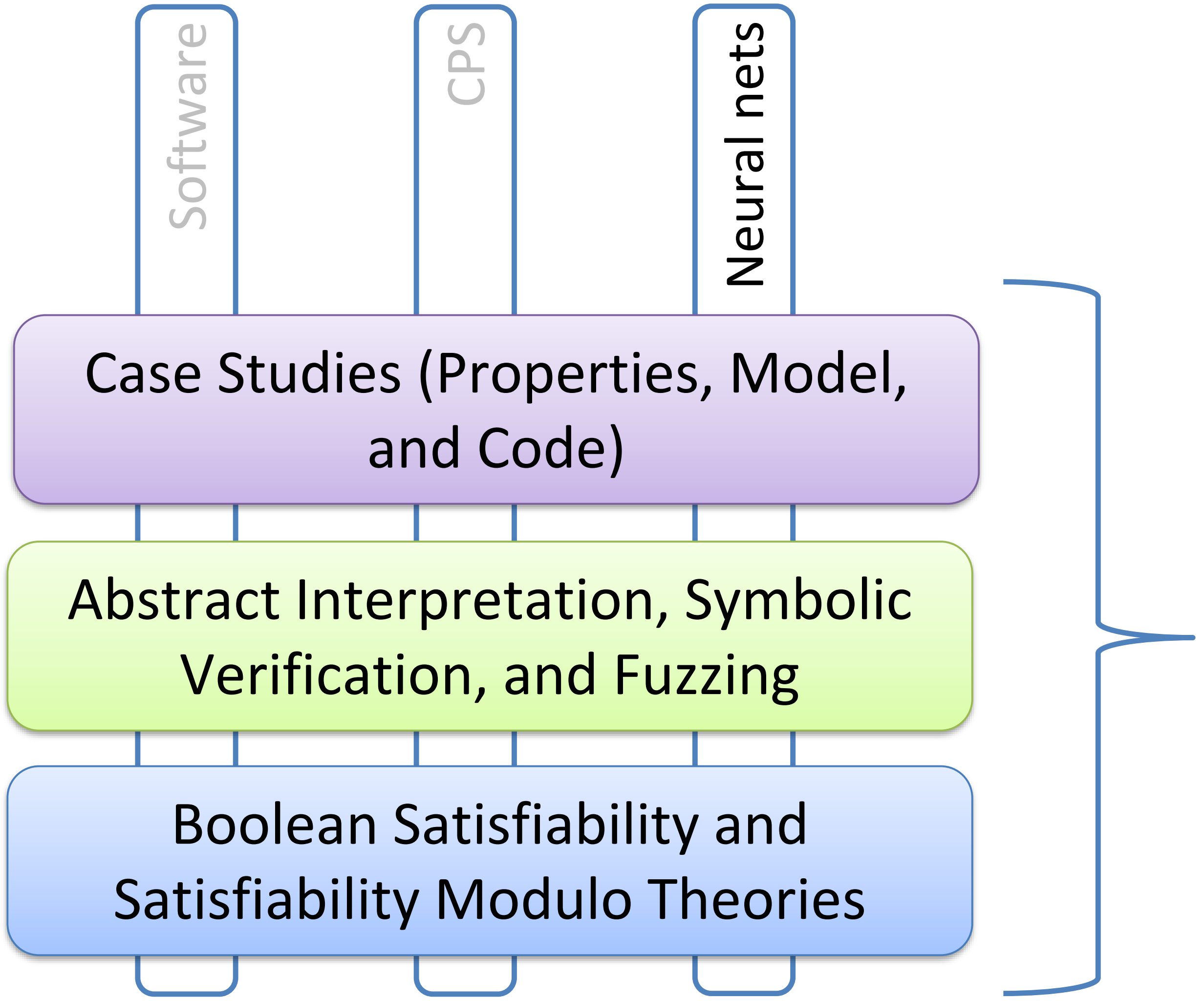
ISSTA 2017, HSCC 2017 and 2018, CAV 2018, ASE 2018, Acta 2020

Future Work

Our synthesis engine might benefit from using techniques ranging from machine learning to more robust formulations for generating candidates in the synthesis scheme

Extend our verification and synthesis methodology to support multiple-input multiple-output (MIMO) systems

Outline


Software
CPS
Neural nets
Case Studies (Properties, Model, and Code)
Abstract Interpretation, Symbolic Verification, and Fuzzing
Boolean Satisfiability and Satisfiability Modulo Theories

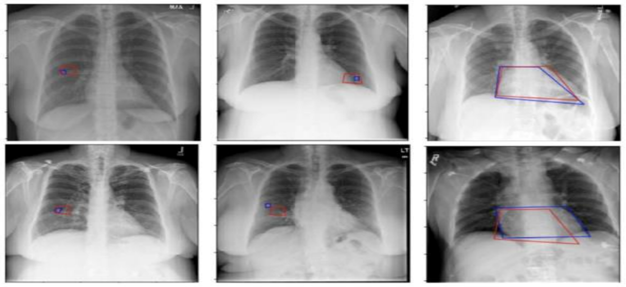
Approaches to formally build verified **trustworthy software systems** to ensure **confidentiality, integrity and availability**

Neural Networks (NN)

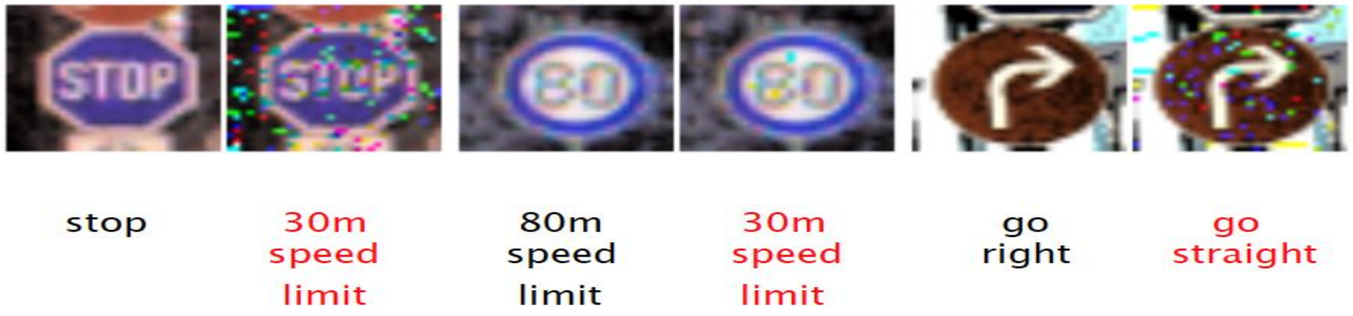
- NNs are computing systems capable of **learning tasks from examples**

Recognize traffic signs and objects

Identify regions to be inspected

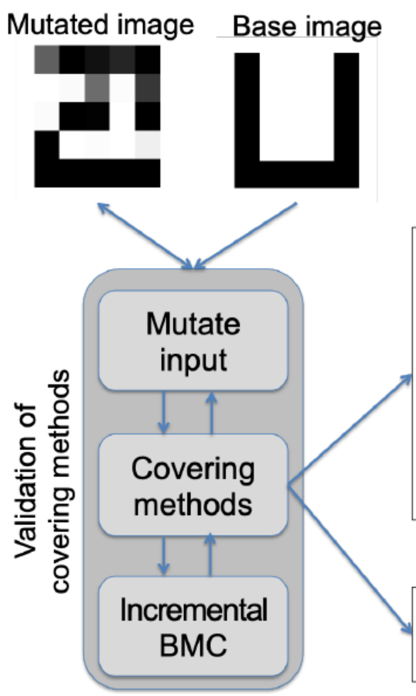


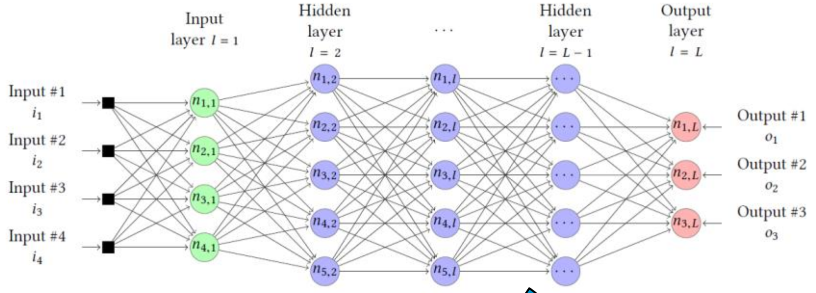
- NNs are known to be vulnerable to **adversarial attacks**



Validation of Covering Methods

Generate executions of an ANN implementation that lead to neuron activation





Source Code

```
void gpu_float2half_rn(
int size, const value_type *buffIn, half1 *buffOut)
{
  int grid_size = (size + BLOCK_SIZE − 1) /
BLOCK_SIZE;
  float2half_rn_kernel<value_type><<<grid_size,
BLOCK_SIZE>>> (size, buffIn, buffOut);
  checkCudaErrors(cudaDeviceSynchronize());
…
}
```

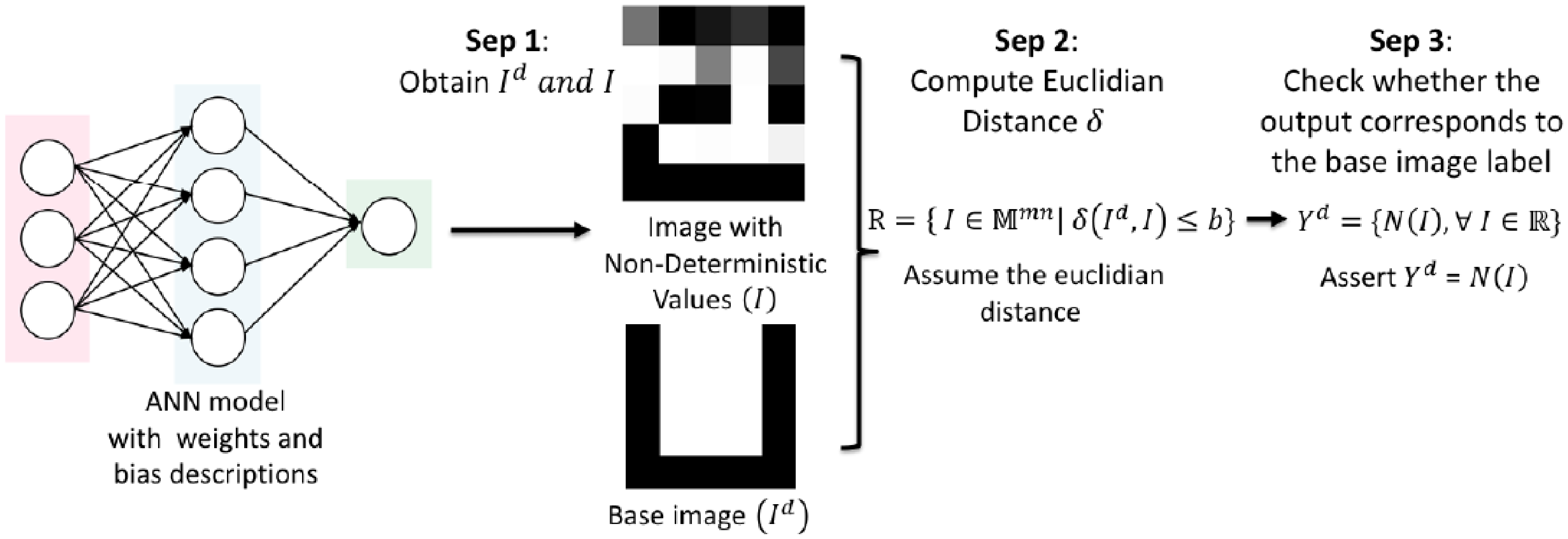
Coverage metrics

```
__ESBMC_assert(neuronCoverageSS() > 0.8, "At least 80%
of all neurons must be SS-Covered.");
…
```

numerical errors and disagreements between DNN implementations and their quantized versions

Verification of Adversarial Case

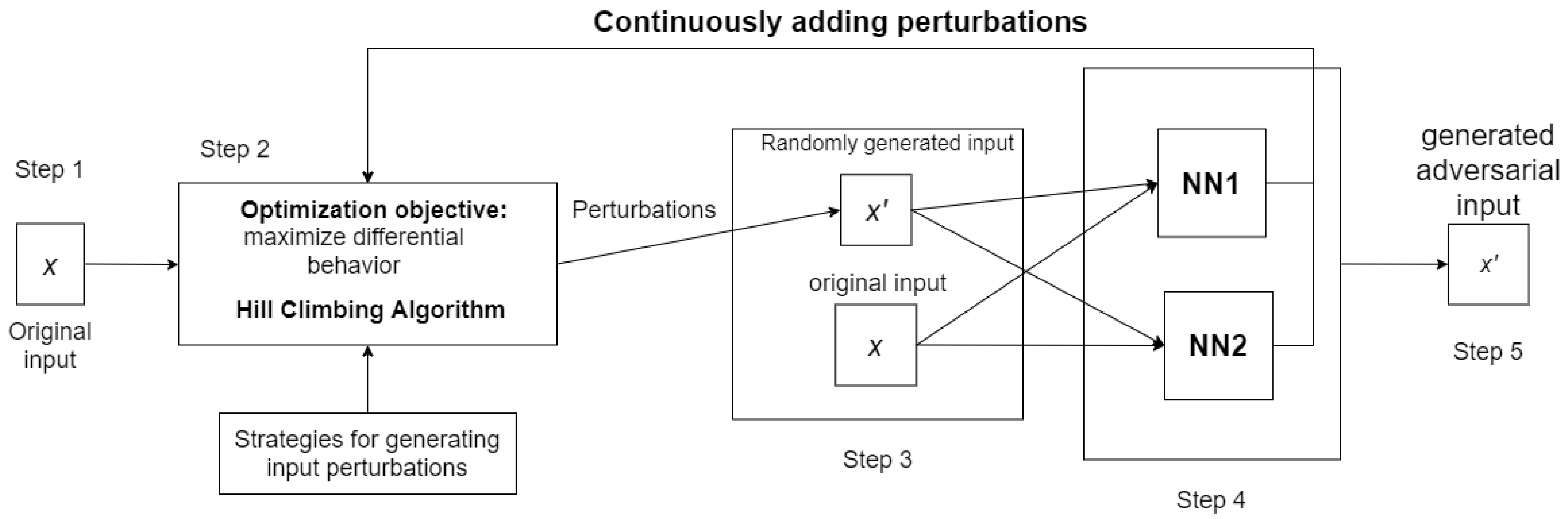
Obtain an adversarial input that can lead the ANN to failures, e.g., misclassifying an image



ANN model with weights and bias descriptions
Sep 1: Obtain I^d *and* I
Image with Non-Deterministic Values (I)
Base image (I^d)
Sep 2: Compute Euclidian Distance δ
$R = \{ I \in \mathbb{M}^{mn} \mid \delta(I^d, I) \leq b \}$
Assume the euclidian distance
Sep 3: Check whether the output corresponds to the base image label
$Y^d = \{N(I), \forall I \in \mathbb{R}\}$
Assert $Y^d = N(I)$

Generating Adversarial Inputs Using A Black-box Differential Technique

DAEGEN queries the NNs with given input and makes perturbations on the input based on observations obtained from the previous queries


Continuously adding perturbations
Step 1
x
Original input
Step 2
Optimization objective: maximize differential behavior
Hill Climbing Algorithm
Strategies for generating input perturbations
Perturbations
Randomly generated input
x'
original input
x
Step 3
NN1
NN2
Step 4
generated adversarial input
x'
Step 5

Future Work

Investigate fault localization and repair techniques to explain errors and make the ANN implementation robust against small noises present in the ANN inputs

Revisit the adversarial case generation using abstract interpretation techniques to speed up the verification process

Research Mission

Automated **verification** and **synthesis** to ensure the **safety** and **security** in **neural-based architectures**

↓

Methods, algorithms, and tools to write safe and secure software systems