EFFICIENT SAT-BASED SOFTWARE ANALYSIS: FROM AUTOMATED TESTING TO AUTOMATED VERIFICATION AND REPAIR

Nazareno Aguirre (joint work with Marcelo Frias et al.)
Departamento de Computación, FCEFQyN
Universidad Nacional de Río Cuarto
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Argentina

AGENDA

- One Analysis Approach
  - SAT Solving
- Two Analysis Techniques
  - Tight Bounds
  - Transcoping
- Three Software Engineering Problems
  - Software Testing
  - Program Verification
  - Program Repair

FORMAL METHODS IN PRACTICE

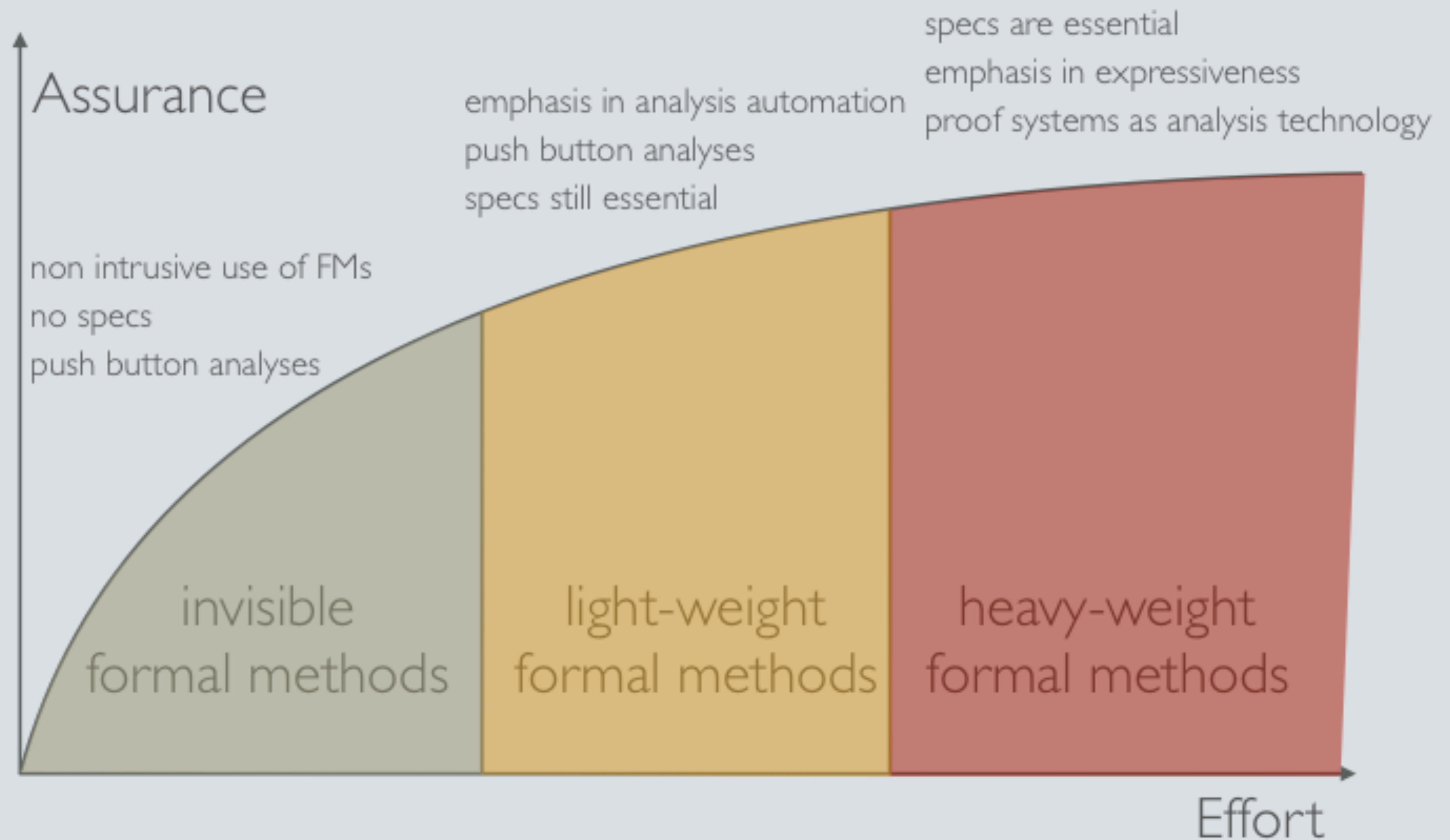
- A classical objection:

"…the cost of full verification is prohibitive…"

[Jackson & Wing 1996]

Two major sources of complexity/cost:

- ***the specification problem:*** *capture the specification of the system and properties of interest in a formal language*
- ***the analysis problem:*** *analyze the consequences of design decisions, the validity of properties, etc.*



A TREND IN FORMAL METHODS

[Rushby et al. 2003-2004]

PROGRAM VERIFICATION

$\forall s, s' \cdot pre(s) \wedge P(s, s') \Rightarrow post(s')$

$\forall s \cdot pre(s) \Rightarrow P(s)$ terminates

BOUNDED PROGRAM VERIFICAT

limit on the number of objects in program states, the range for numeric types, …

limit on the number of loop iterations and depth of recursive calls

$\{pre(s$

P $_B)$

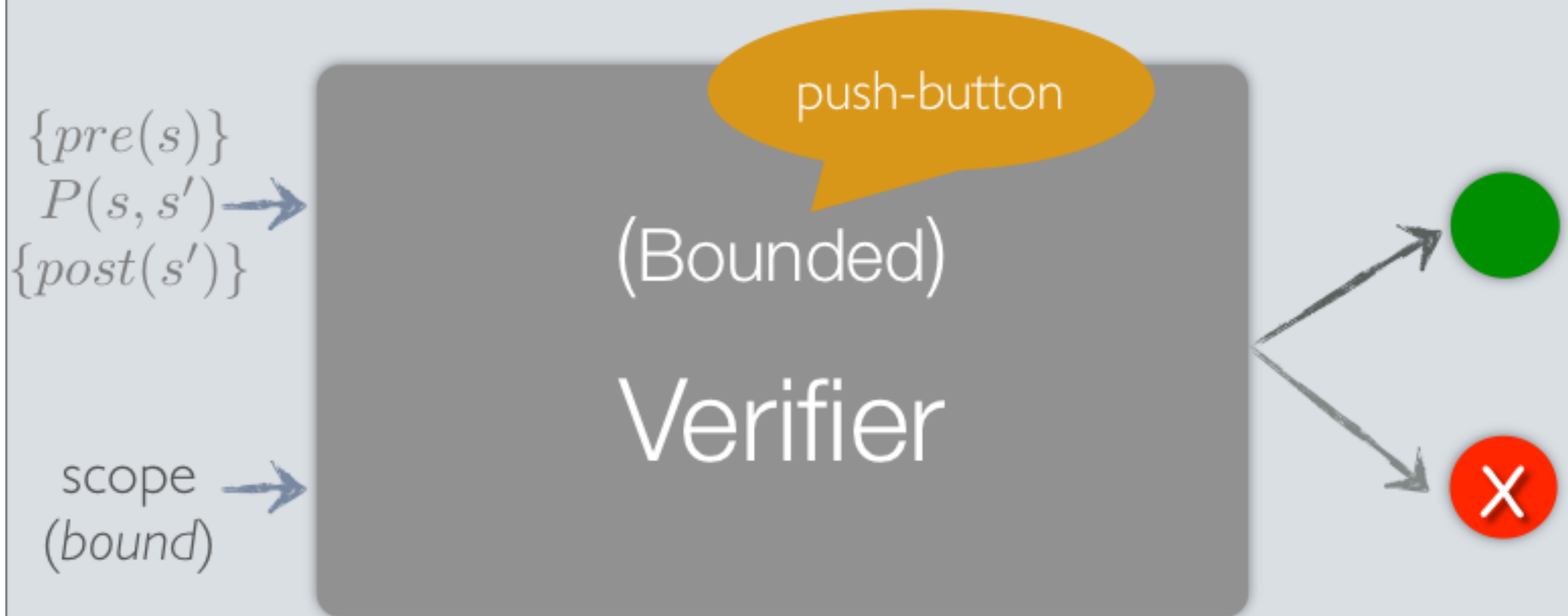
$st(s'_B)\}$

DECIDABLE

?

$\forall s_B, s'_B \cdot pre(s_B) \land P_B(s_B, s'_B) \Rightarrow post(s'_B)$
$\forall s_B \cdot P$ terminates within B iterations

BOUNDED VERIFICATION



SAT-BASED BOUNDED VERIFICATION

THE BOOLEAN SATISFIABILITY PROBLEM

Given a **propositional** formula *A*, is it *satisfiable*?

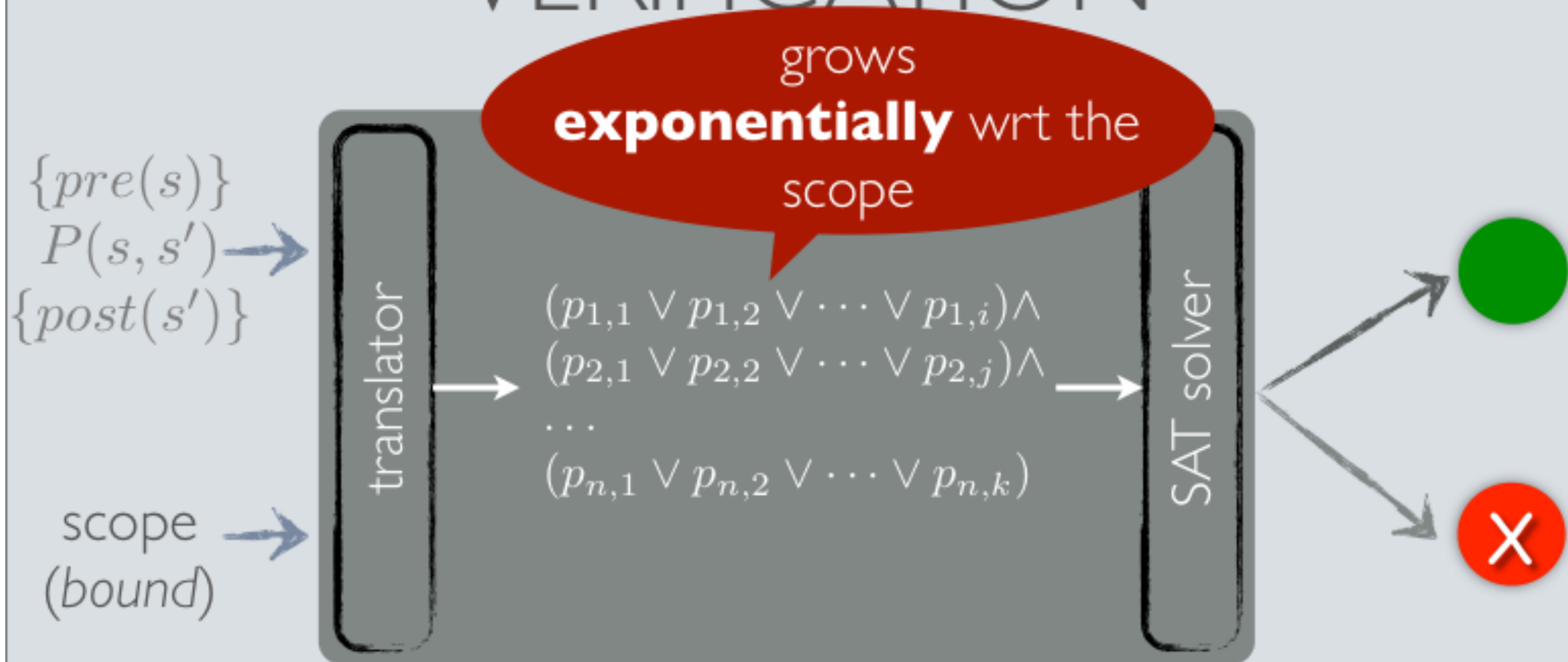
i.e., find a valuation:

$$v : Vars(V) \rightarrow \{\mathsf{T}, \mathsf{F}\}$$

that makes the formula true

- **Decidable** problem
- First problem known to be **NP-complete**
- Several **efficient implementations for SAT** based on Davis-Putnam-Logemann-Loveland (DPLL)

SAT-BASED BOUNDED VERIFICATION



SAT-BASED BOUNDED VERIFICATION

$\{AVL(t)\}$
$t.insert(x)$
$\{AVL(t')\}$

up to 7 nodes
[0,7] int
up to 7 loop unrolls

translator

546798 variables

$(p_{1,1} \vee p_{1,2} \vee \cdots \vee p_{1,i}) \wedge$
$(p_{2,1} \vee p_{2,2} \vee \cdots \vee p_{2,j}) \wedge$
$\ldots$
$(p_{n,1} \vee p_{n,2} \vee \cdots \vee p_{n,k})$

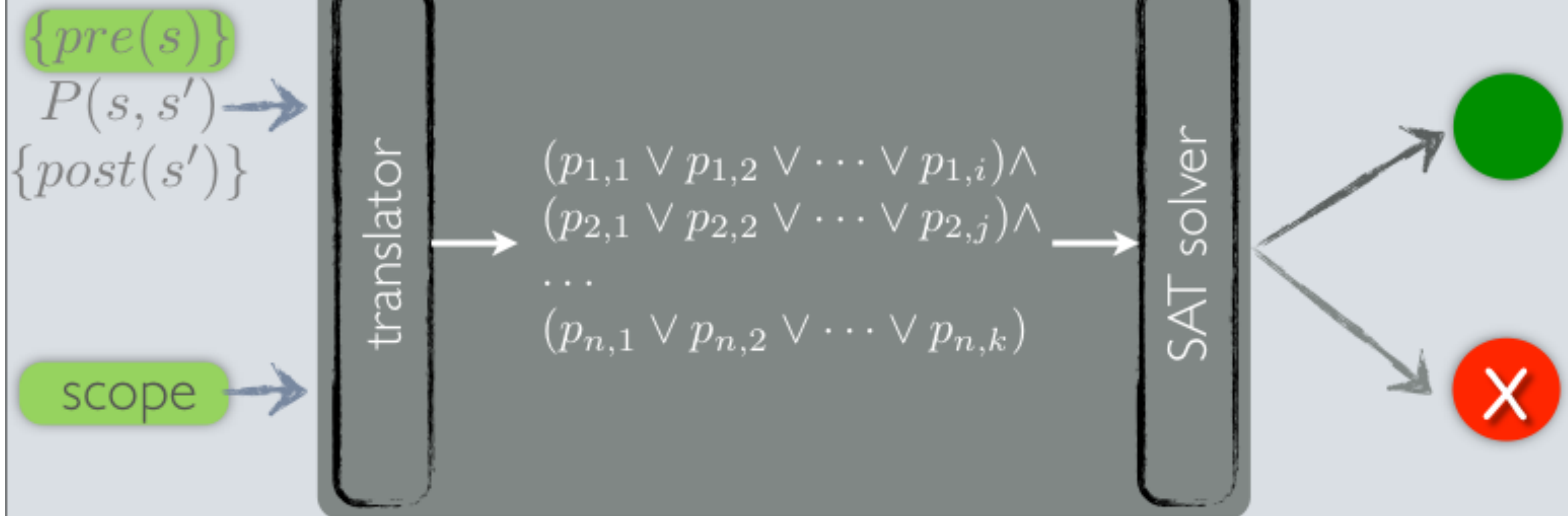
SAT solver

>5 hours

captures all
executions of *insert* on valid trees,
with up to 7 iterations, handling
trees of up to 7 nodes

TIGHT BOUNDS

BOUNDED VERIFICATION



[1] Galeotti, Rosner, López Pombo & Frias, *Analysis of Invariants for Efficient Bounded Verification*. ISSTA 2010.

EXAMPLE: REMOVEFIRST IN A

EXAMPLE: REMOVEFIRST IN A SINGLYLINKEDLIST

$$\{acyclicList(l) \wedge \neg empty(l)\}$$
$$l.\mathsf{removeFirst}()$$
$$\{acyclicList(l')\}$$

Scope: 0..1 list, 0..3 nodes

PROGRAM STATE STRUCTURE

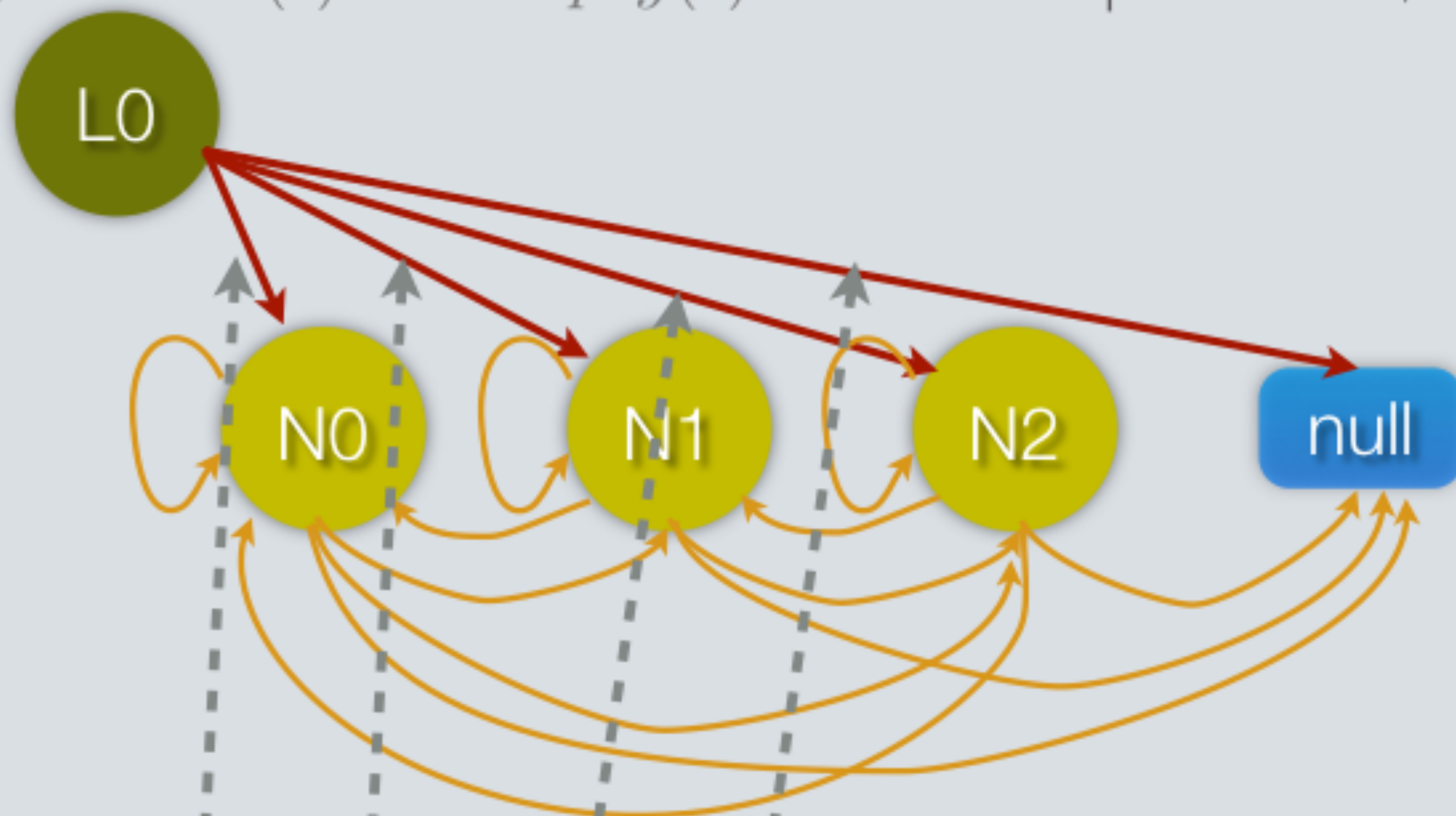
$pre : acyclicList(l) \wedge \neg empty(l)$

Scope: 0..1 list, 0..3 nodes

PROGRAM STATE STRUCTURE

$pre : acyclicList(l) \wedge \neg empty(l)$

Scope: 0..1 list, 0..3 nodes



SAT representation

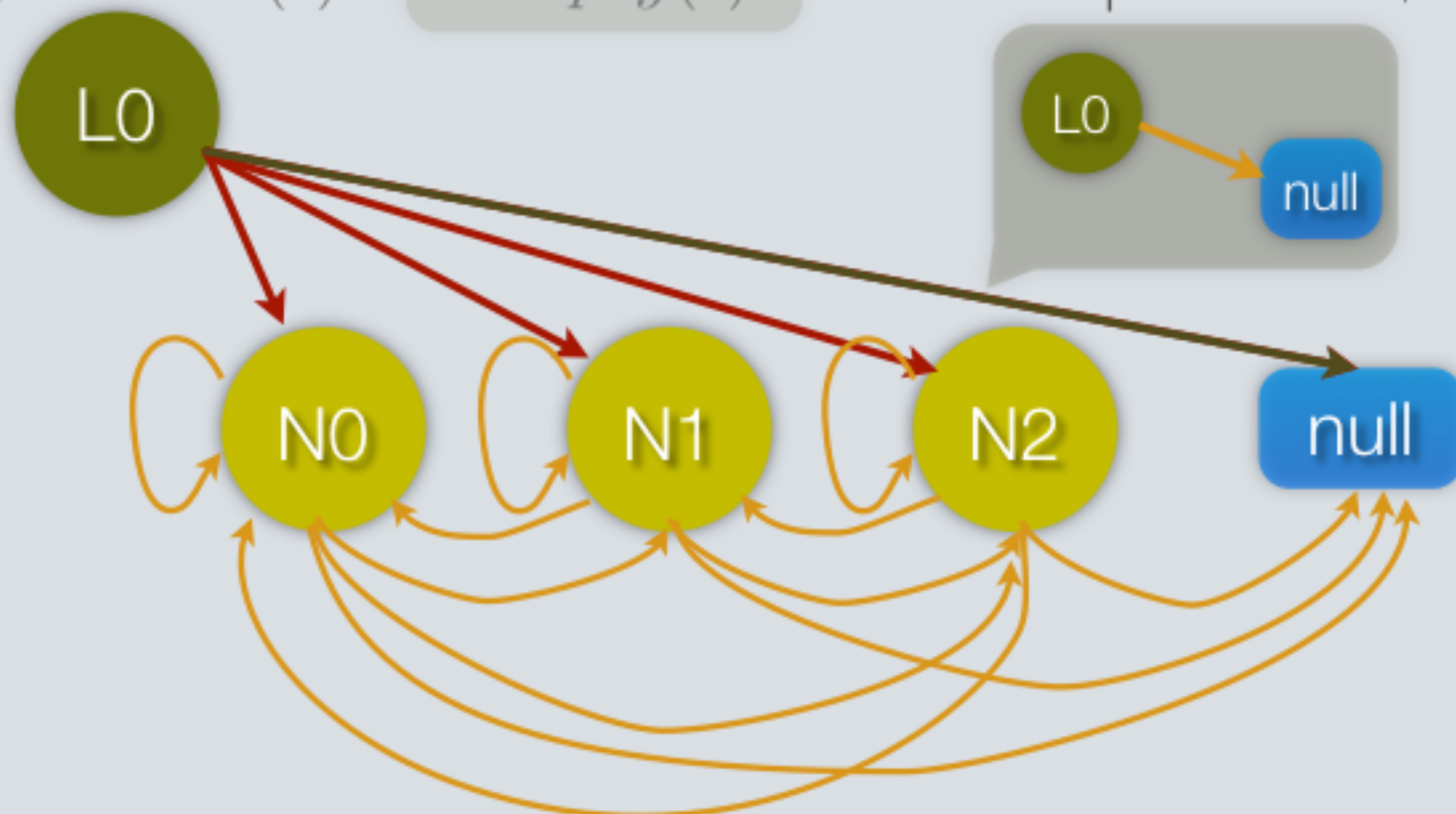
head	N0	N1	N2	null
L0	p_{L0N0}	p_{L0N1}	p_{L0N2}	p_{L0null}

next	N0	N1	N2	null
N0	p_{N0N0}	p_{N0N1}	p_{N0N2}	p_{N0null}
N1	p_{N1N0}	p_{N1N1}	p_{N1N2}	p_{N1null}
N2	p_{N2N0}	p_{N2N1}	p_{N2N2}	p_{N2null}

PROPERTIES OF THE INITIAL STATES

$pre : acyclicList(l) \wedge \neg empty(l)$

Scope: 0..1 list, 0..3 nodes

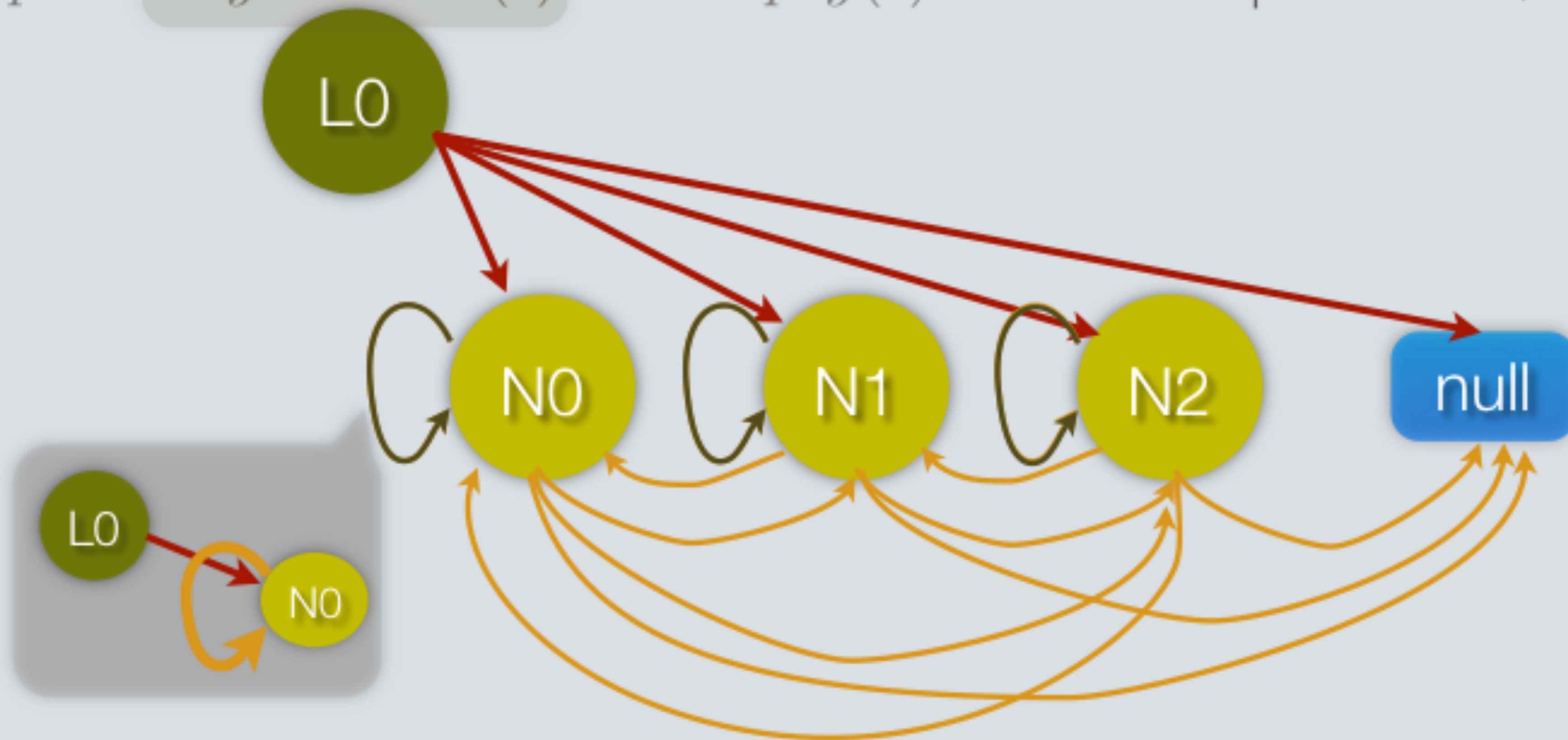


head	N0	N1	N2	null
L0	p_{L0N0}	p_{L0N1}	p_{L0N2}	false

next	N0	N1	N2	null
N0	p_{N0N0}	p_{N0N1}	p_{N0N2}	p_{N0null}
N1	p_{N1N0}	p_{N1N1}	p_{N1N2}	p_{N1null}
N2	p_{N2N0}	p_{N2N1}	p_{N2N2}	p_{N2null}

PROPERTIES OF THE INITIAL STATES

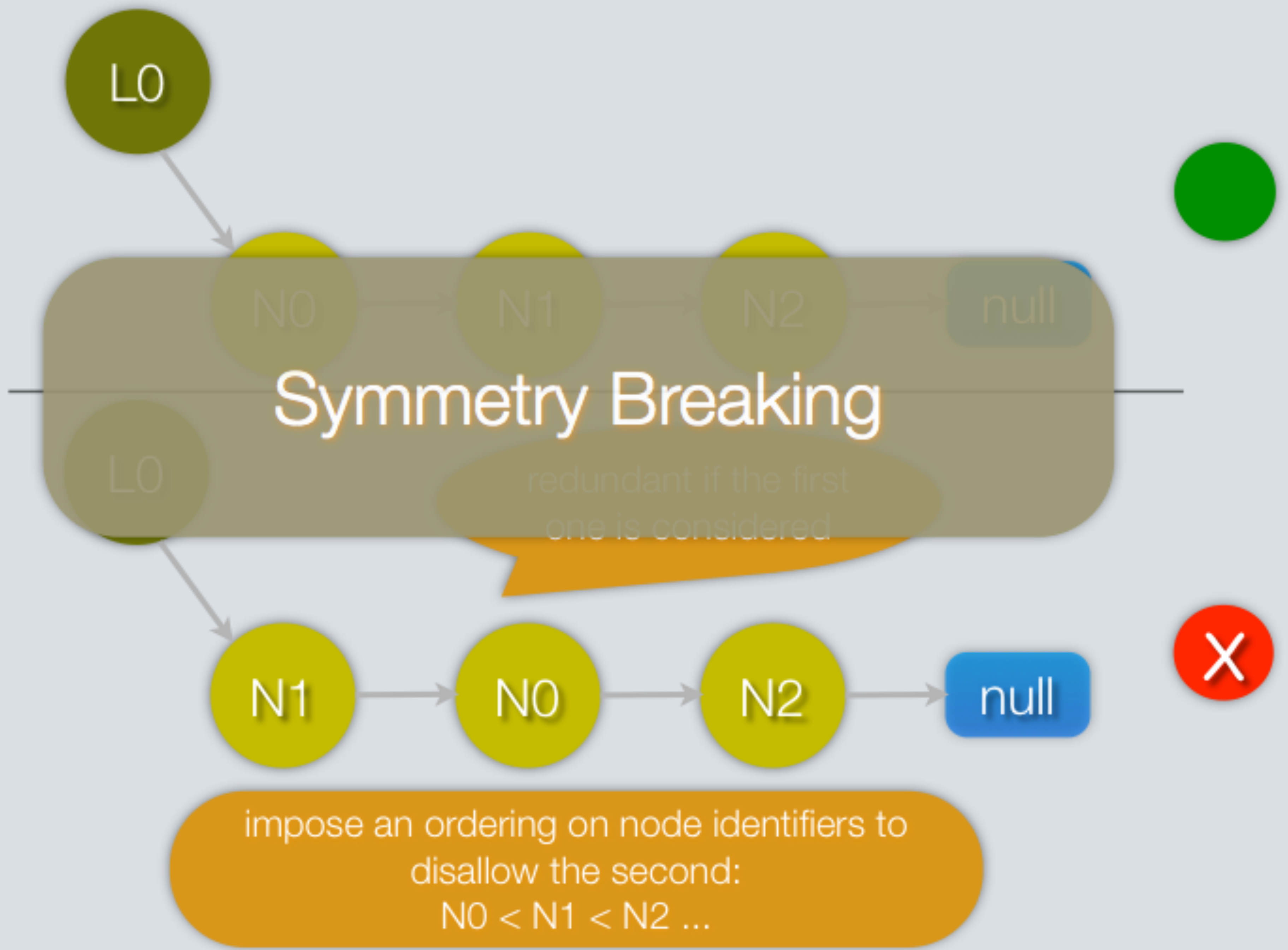
$pre : acyclicList(l) \wedge \neg empty(l)$ Scope: 0..1 list, 0..3 nodes



head	N0	N1	N2	null
L0	p_{L0N0}	p_{L0N1}	p_{L0N2}	p_{L0null}

next	N0	N1	N2	null
N0	false	p_{N0N1}	p_{N0N2}	p_{N0null}
N1	p_{N1N0}	false	p_{N1N2}	p_{N1null}
N2	p_{N2N0}	p_{N2N1}	false	p_{N2null}

ISOMORPHIC STRUCTURES



PROPERTIES OF THE INITIAL STATES

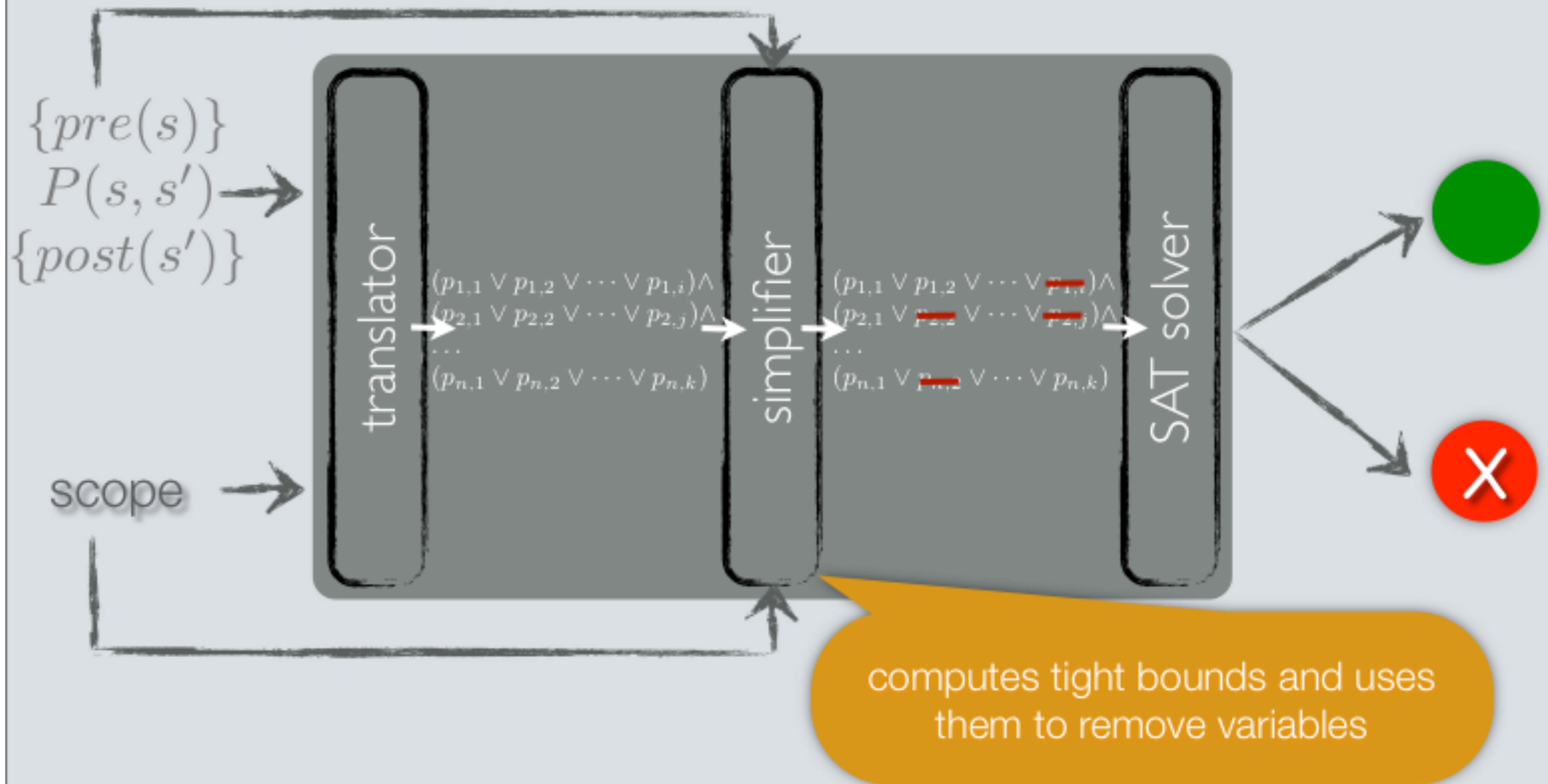
$pre : acyclicList(l) \wedge \neg empty(l) \wedge$ symmetry breaking

head	N0	N1	N2	null
L0	p_{L0N0}	false	false	false

next	N0	N1	N2	null
N0	false	p_{N0N1}	false	p_{N0null}
N1	false	false	p_{N1N2}	p_{N1null}
N2	false	false	false	p_{N2null}

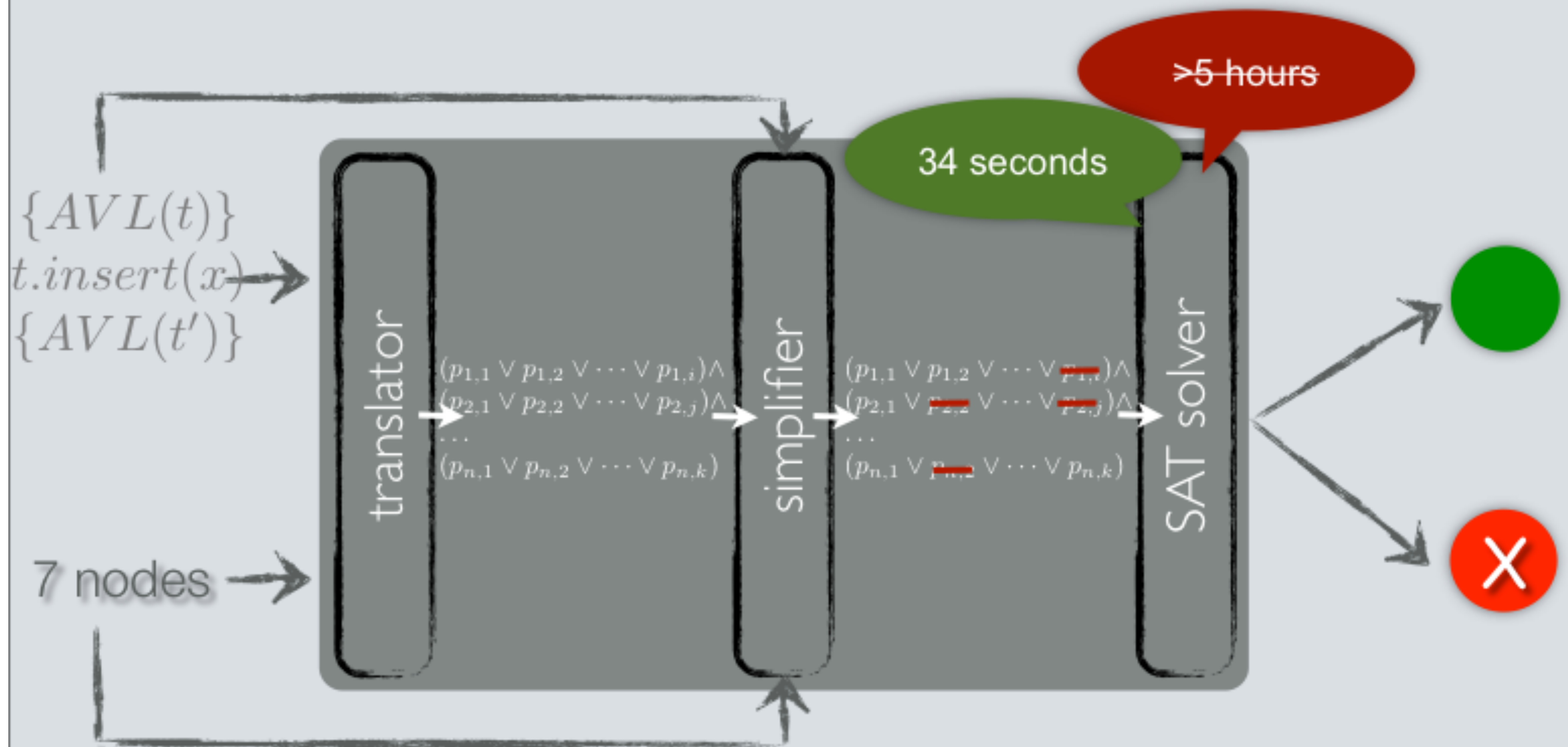
TIGHT BOUNDS IN SAT-BASED

BOUNDED VERIFICATION



TIGHT BOUNDS IN SAT-BASED BOUNDED VERIFICATION

BOUNDED VERIFICATION



How can the bounds be computed?

TIGHT BOUNDS COMPUTATION

Top-down approach

$\exists l \cdot acyclicList(l) \wedge p_{N0,N0}$?

$\exists l \cdot acyclicList(l) \wedge p_{N0,N1}$?

start with all variables in the tight bound

tight bound

X

p_{N0N0}	p_{N0N1}	p_{N0N2}	p_{N0null}
p_{N1N0}	p_{N1N1}	p_{N1N2}	p_{N1null}
p_{N2N0}	p_{N2N1}	p_{N2N2}	p_{N2null}

make a SAT query per variable (variable feasibility)

next	N0	N1	N2	null
N0	p_{N0N0}	p_{N0N1}	p_{N0N2}	p_{N0null}
N1	p_{N1N0}	p_{N1N1}	p_{N1N2}	p_{N1null}
N2	p_{N2N0}	p_{N2N1}	p_{N2N2}	p_{N2null}

Galeotti et al., *Analysis of Invariants for Efficient Bounded Verification*. ISSTA 2010.

TIGHT BOUNDS COMPUTATION

Bottom-up approach

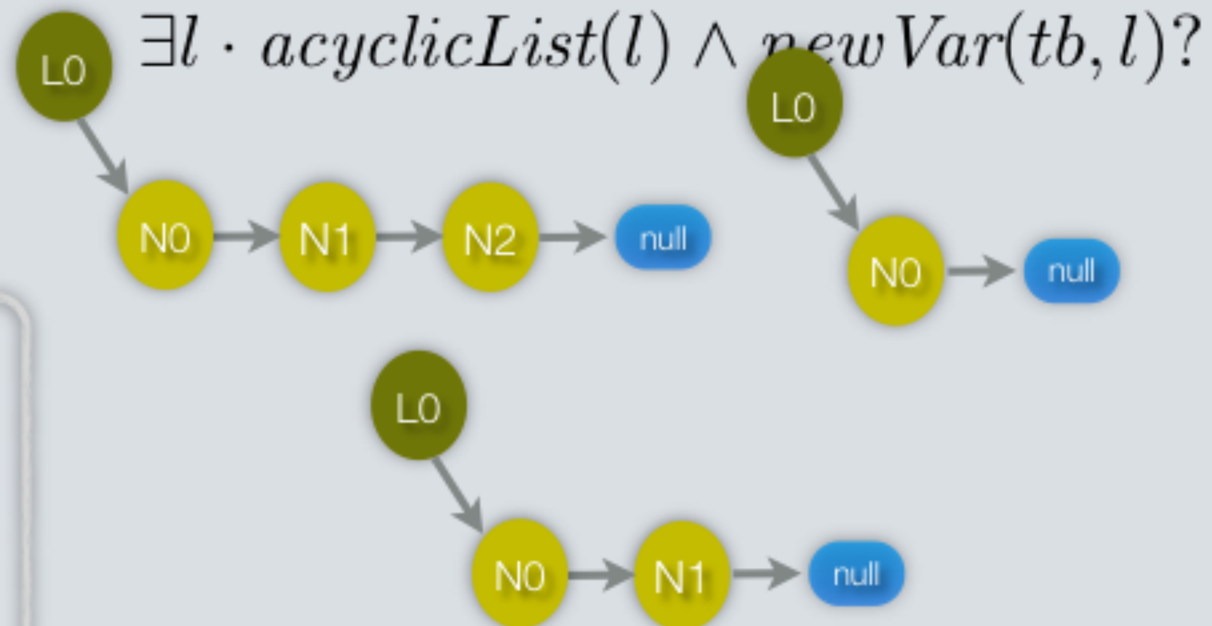
ask for structures that "extend" the bound, until no further bound-extending structure can be produced

start with **no** variable in the tight bound

tight bound

false	pN0N1	false	pN0null
false	false	pN1N2	pN1null
false	false	false	pN2null

when done, remaining variables are infeasible



next	N0	N1	N2	null
N0	pN0N0	pN0N1	pN0N2	pN0null
N1	pN1N0	pN1N1	pN1N2	pN1null
N2	pN2N0	pN2N1	pN2N2	pN2null

Ponzio, Aguirre, Frias, Visser, *Field-Exhaustive Testing*. FSE 2016.

TIGHT BOUNDS COMPUTATION

TOP-DOWN VS. BOTTOM-UP

Top Down

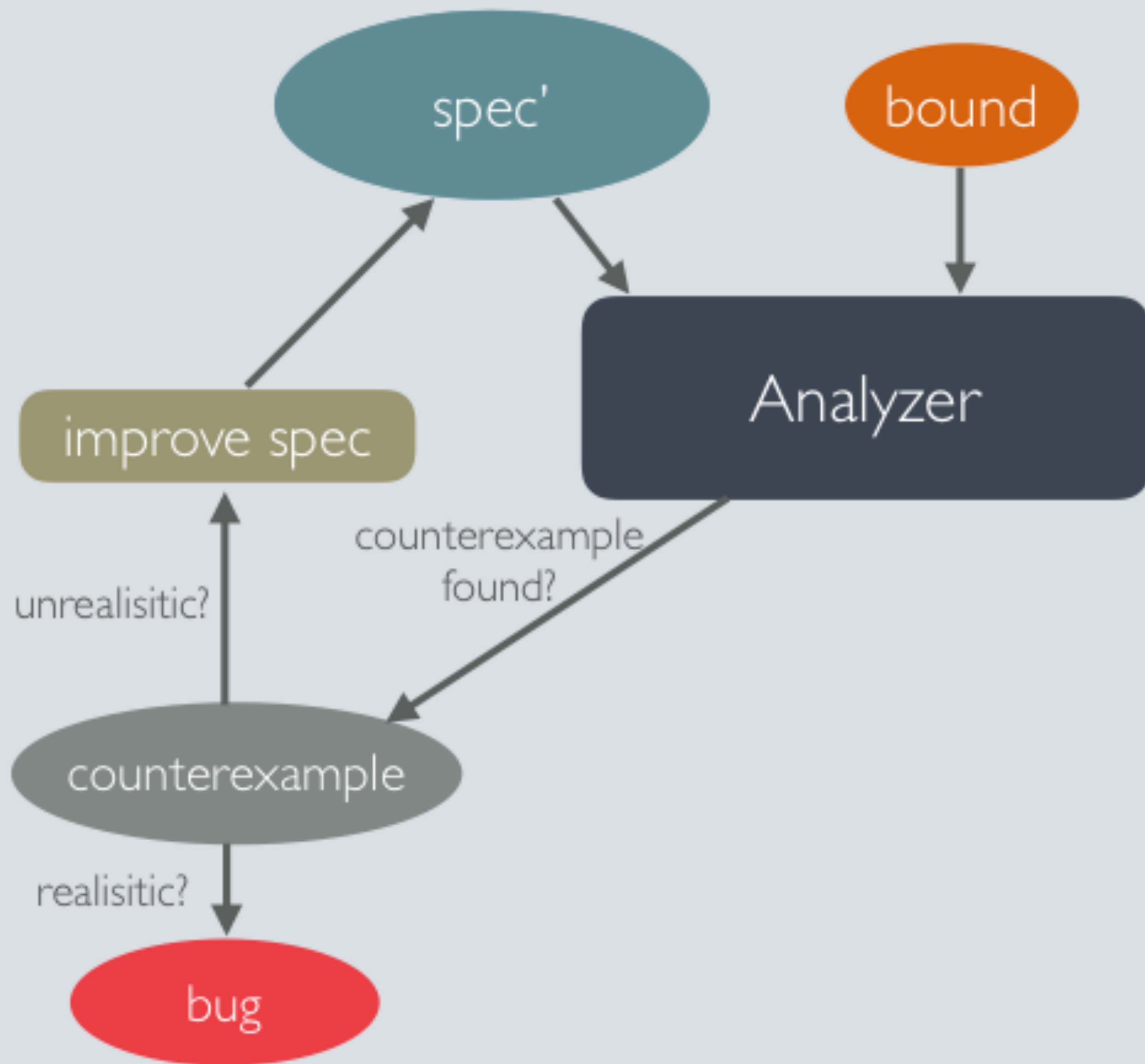
- **Amenable to parallelization**
  - (many) independent SAT calls
- **"Partial" bounds still useful**
- **Requires a cluster for efficient computation**

Bottom Up

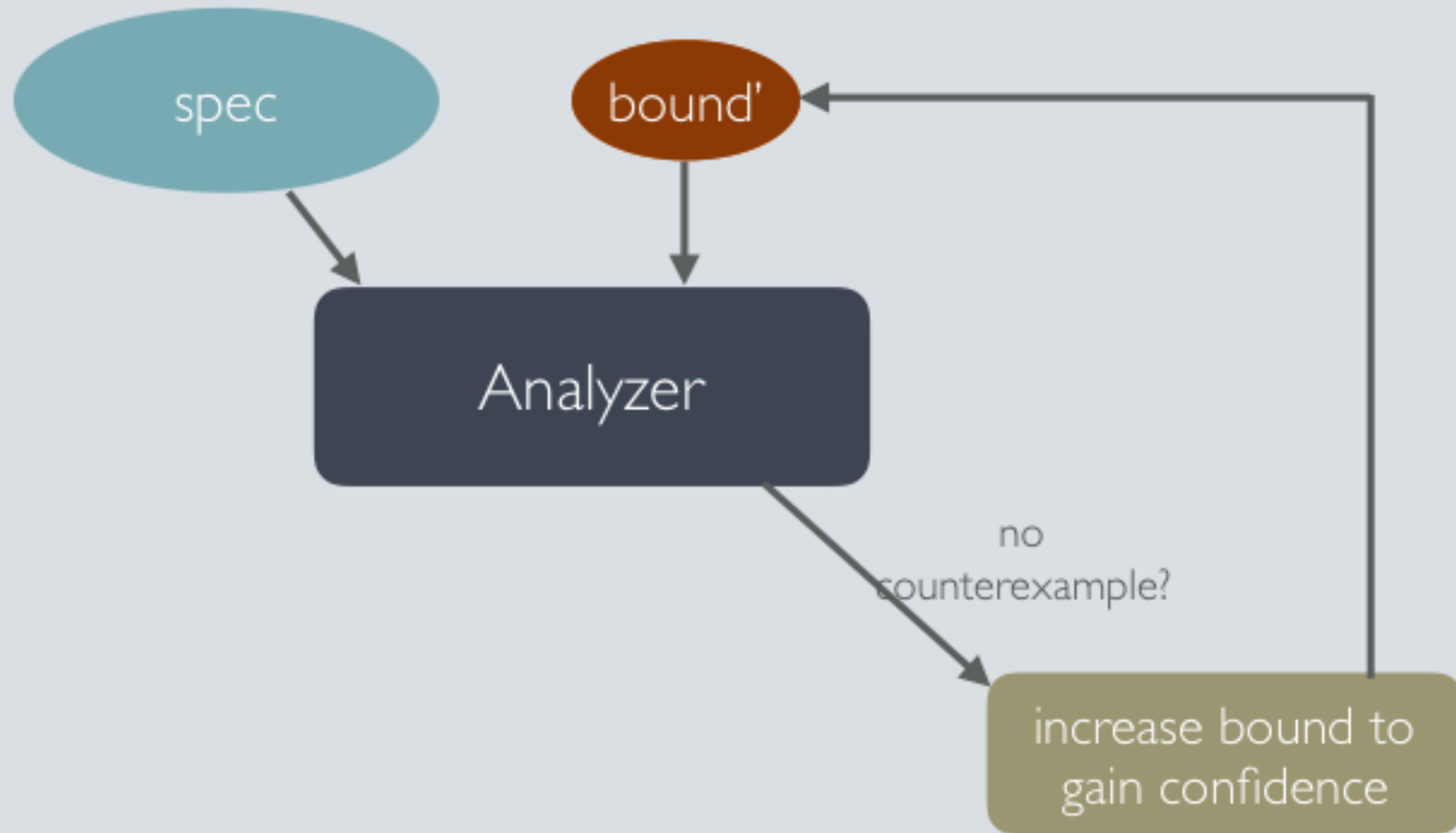
- **Very difficult to parallelize**
- **Bounds useful only when algorithm terminates**
- **Fewer SAT calls**
  - **In a single core, as efficient as top-down on a cluster of 64 cores**

TRANSCOPING

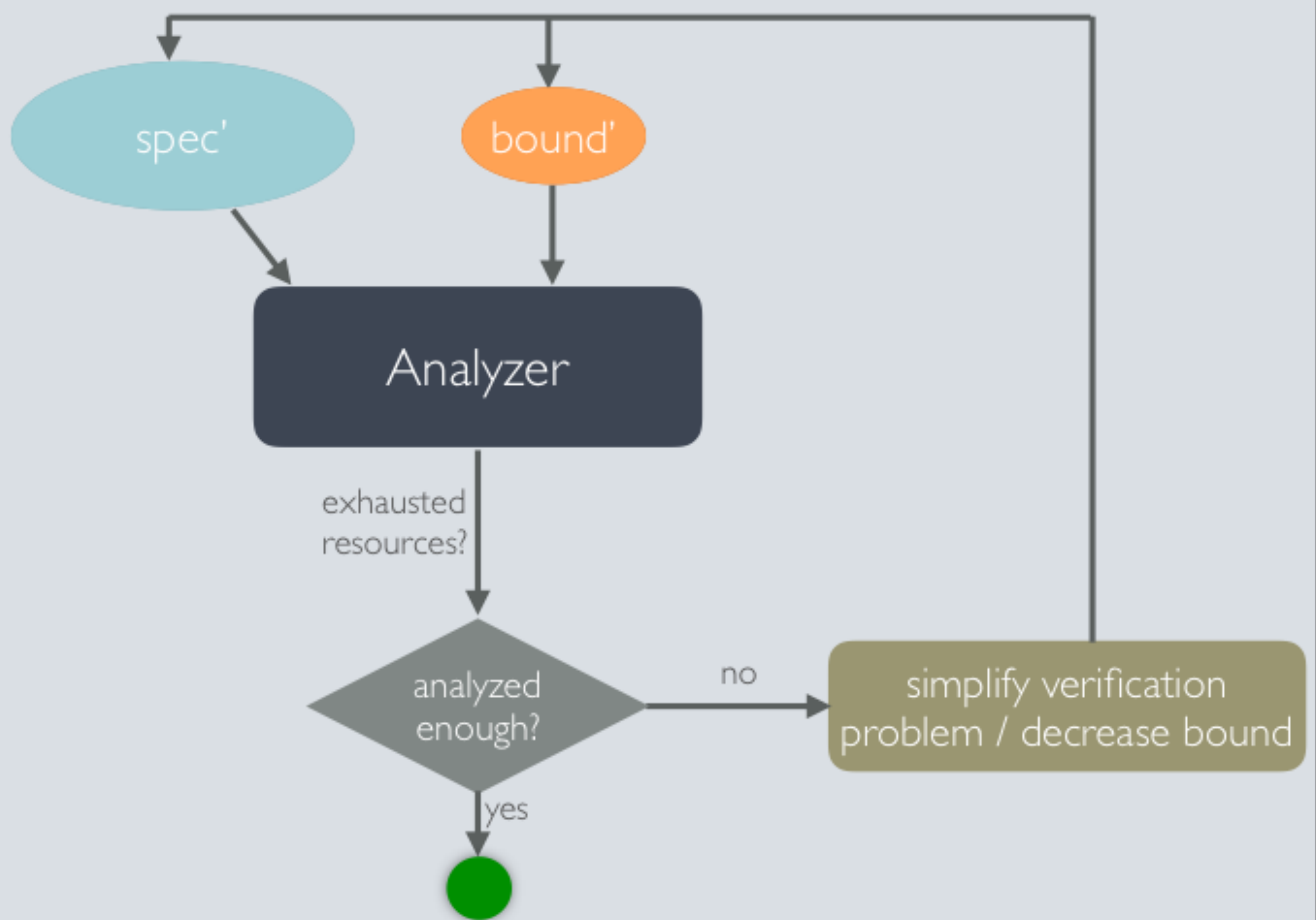
A TYPICAL (BOUNDED) AUTOMATED ANALYSIS SCENARIO



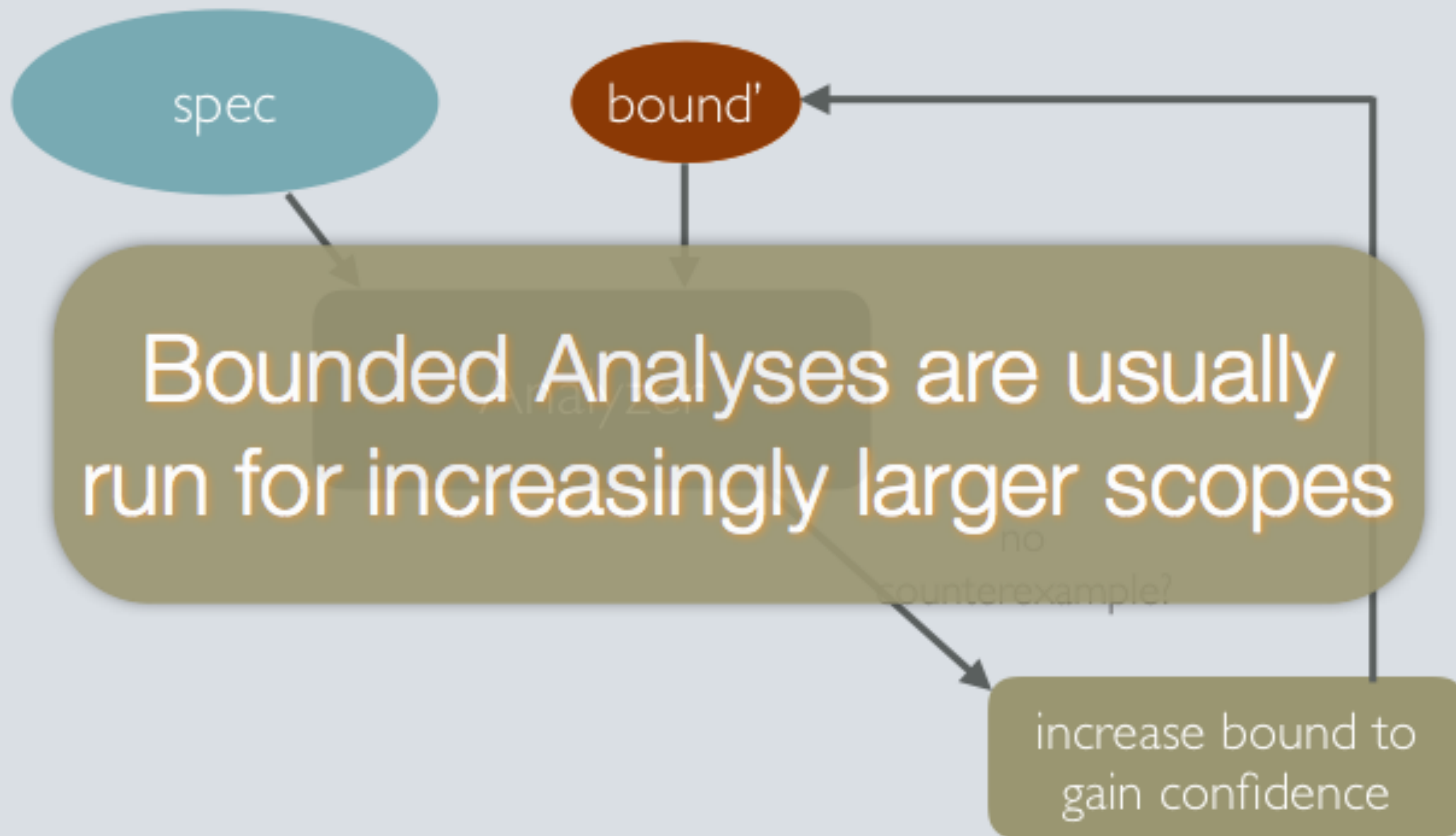
A TYPICAL (BOUNDED) AUTOMATED ANALYSIS SCENARIO



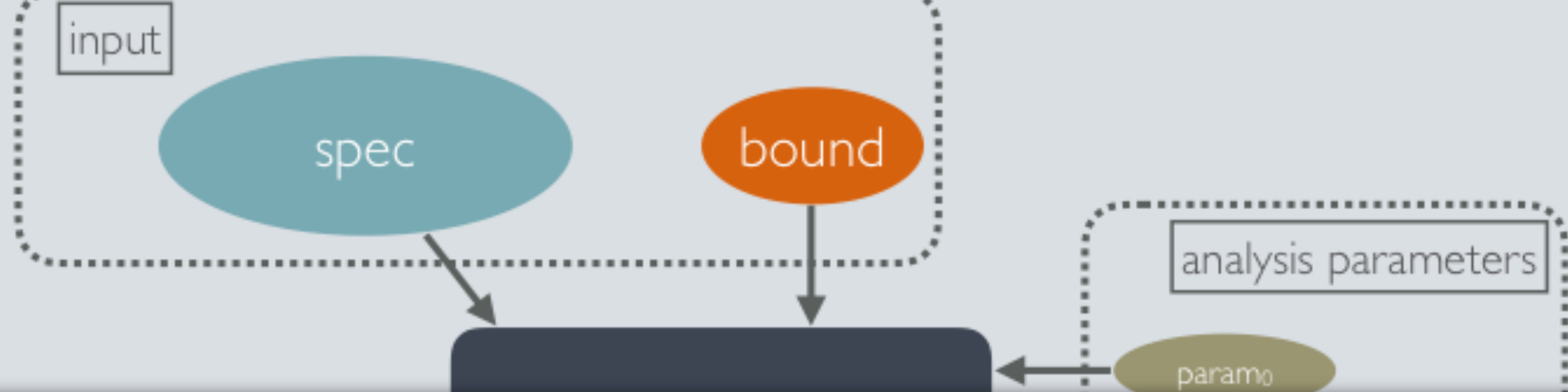
A TYPICAL (BOUNDED) AUTOMATED ANALYSIS SCENARIO



A TYPICAL (BOUNDED) AUTOMATED ANALYSIS SCENARIO



ANALYSIS PARAMETERS



Transcoping: run analysis with *many* different settings on small scopes, then extrapolate the best to larger ones

- **Increasing number of parameters** as tools/techniques evolve
  - e.g., "parameters" in model checking: approach (SAT vs BDD?), state matching mechanism, partial order reduction, state abstraction, …
- **Analysis parameters can dramatically affect performance**
- **Optimal parameter instantiation for *all* situations are unknown**

A USE OF TRANSCOPING IN SAT-BASED VERIFICATION

Variable selection heuristics is crucial for SAT-Solving, in *parallel* SAT Solving, for instance.

Intuition: Given a formula α, to query SAT(α) in parallel, choose n variables in α to produce 2^n independent SAT problems.

Goal: choose variables that split the original SAT problem in relatively balanced smaller SAT jobs.

Rosner, López Pombo, Aguirre, Jaoua, Mili, Frias, *Parallel Bounded Verification of Alloy Models by Transcoping*, VSTTE 2013.

THE VARIABLE SELECTION PROBLEM

How do we select the variable for splitting?

$Q \vee R) \wedge (\neg Q \vee R) \wedge (\neg Q \vee \neg R) \wedge (P \vee \neg Q \vee \neg R)$

Problem: Given a set of (unassigned) variables of a formula α, what should be the next variable to choose for splitting?

VARIABLE SELECTION ALTERNATIVES

Variable State-Independent Decaying Sum (VSIDS): Each variable is associated a value, that depends on the number of clauses it participates in (incremented as new clauses are learned). Choose variable with highest value.

F Priority: Label variables with the fields (next, head, . . .) they represent in the original verification problem. Choose variables of field F first.

as many heuristics as fields in the verification problem

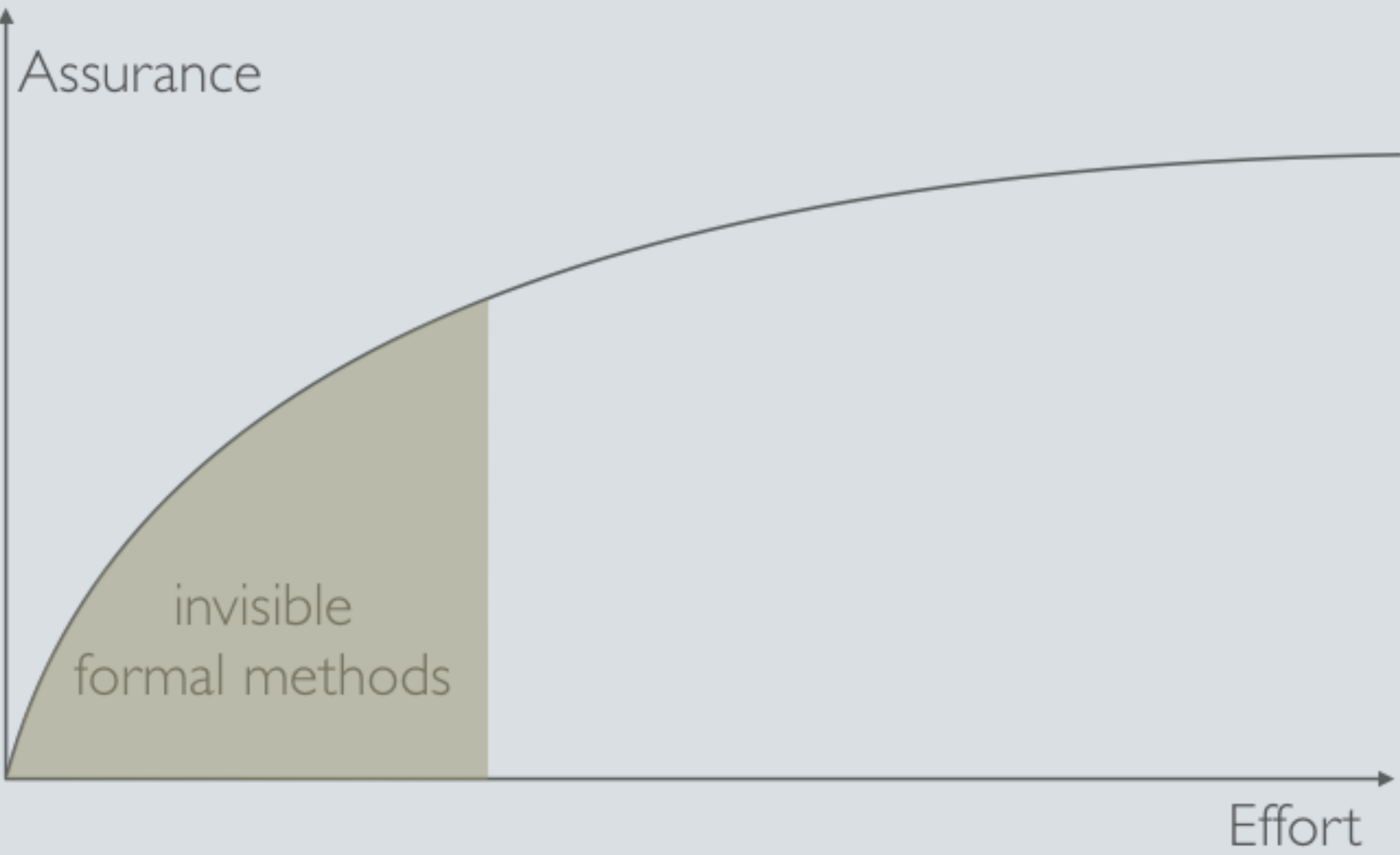
IMPACT OF TRANSCOPING

Routing in Heterogeneous Networks

Scope	Splitter	NUM	MAX	AVG	SUM	DEV	MED
6	Domain3.srcBinding	77	0.08	0.02	1.45	0.02	0.01
	Domain3.BdstBinding	77	0.09	0.02	1.50	0.02	0.01
	Domain2.dstBinding	192	0.18	0.04	7.67	0.03	0.03
	Domain.routing	102	0.21	0.02	1.98	0.03	0.01
	VSIDS	228	0.49	0.01	3.55	0.05	0.00
	Domain3.AdstBinding	192	1.22	0.05	9.78	0.10	0.02
	Identifier_remainder	64	2.31	0.73	46.94	0.52	0.59
7	Domain3.BdstBinding	136	0.84	0.12	17.00	0.18	0.08
	Domain3.srcBinding	141	0.90	0.10	14.60	0.19	0.06
	VSIDS	140	3.39	0.13	19.18	0.38	0.01
	Domain2.dstBinding	192	3.71	0.49	94.74	0.42	0.32
	Domain.routing	192	4.46	0.14	27.51	0.40	0.04
	Domain3.AdstBinding	192	13.05	0.53	101.28	1.04	0.23
	Identifier_remainder	128	25.97	7.45	953.82	6.41	4.93
8	Domain3.srcBinding	136	8.09	1.13	154.17	1.37	0.51
	Domain3.BdstBinding	136	18.06	1.28	173.48	1.95	0.72
	Domain2.dstBinding	192	36.25	8.74	1678.88	7.45	5.78
	VSIDS	174	63.62	1.25	218.29	6.27	0.05
	Domain.routing	192	89.41	2.18	418.04	7.66	0.39
	Domain3.AdstBinding	192	288.79	10.18	1954.07	22.36	2.46
	Identifier_remainder	256	376.70	86.03	22024.53	81.98	56.98
9	Domain3.srcBinding	365	7.57	163.47	2764.89	15.53	3.68
	Domain3.BdstBinding	272	13.25	360.04	3603.38	27.38	5.89

Unprecedented verification up to scope 11 (Domain3.srcBinding)

"INVISIBLE" TIGHT BOUNDS/TRANSCOPING


Assurance
invisible
formal methods
Effort

FIELD-EXHAUSTIVE TESTING

Ponzio, Aguirre, Frias, Visser, *Field-Exhaustive Testing*. FSE 2016.

FIELD-EXHAUSTIVE TESTING

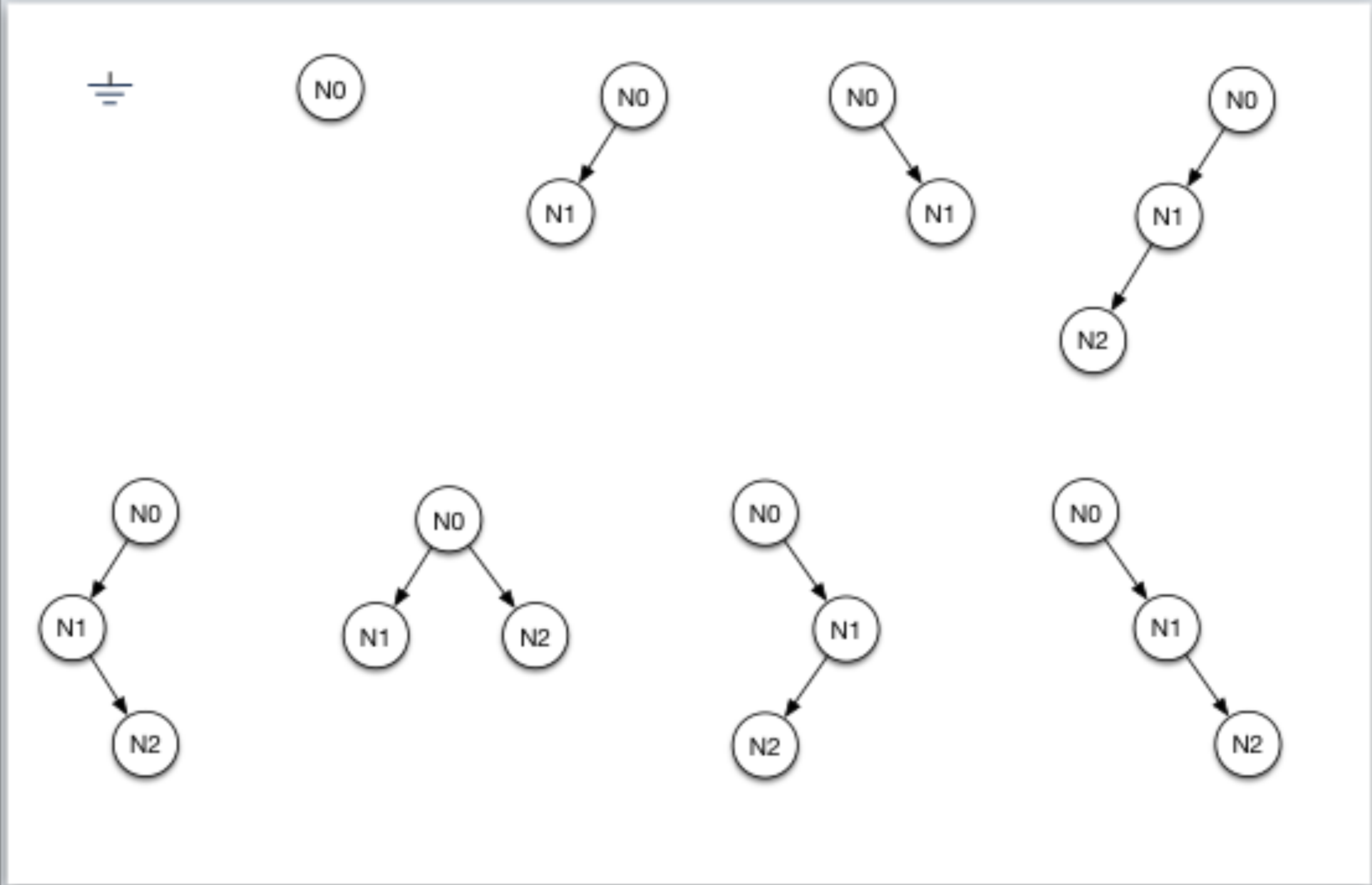
“test the SUT on enough valid inputs so that all feasible field-values within a given limit are covered”

- Exploit structures generated in bottom-up tight bounds computation
  - max. structures quadratically bounded

FIELD-EXHAUSTIVE VS BOUNDED-EXHAUSTIVE

bounded exhaustive case

limit: up to 3 nodes



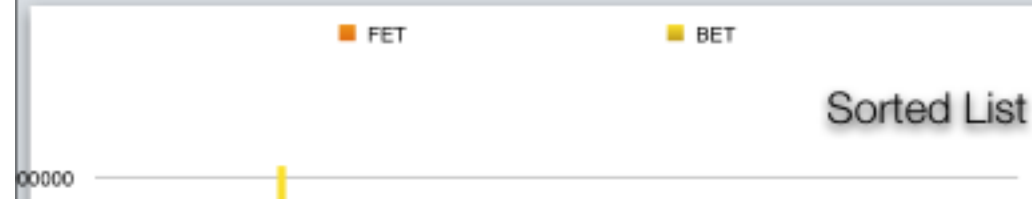
FIELD-EXHAUSTIVE VS BOUNDED-EXHAUSTIVE

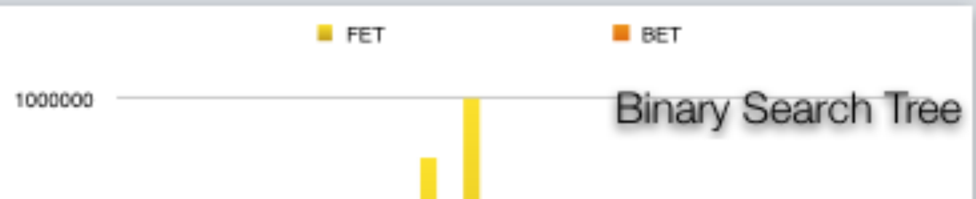
field-exhaustive case

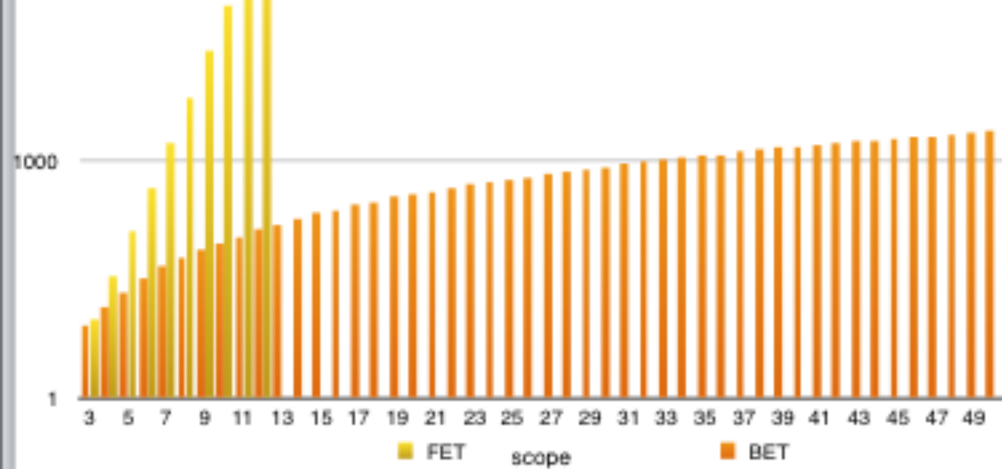
limit: up to 3 nodes

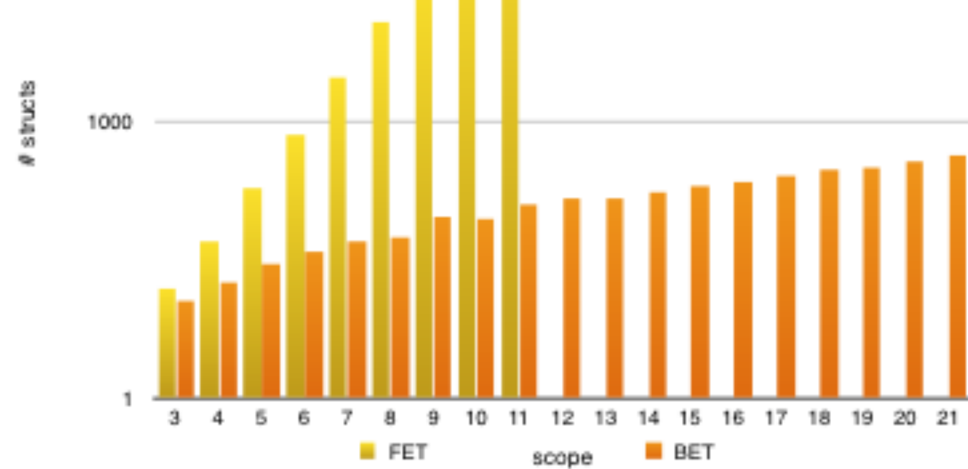
root	left	right
null	(N0,null)	(N0,null)
N0	(N0,N1)	(N0,N1)
	(N1,null)	(N1,null)
	(N1,N2)	(N2,null)
	(N2,null)	(N1,N2)
		(N0,N2)

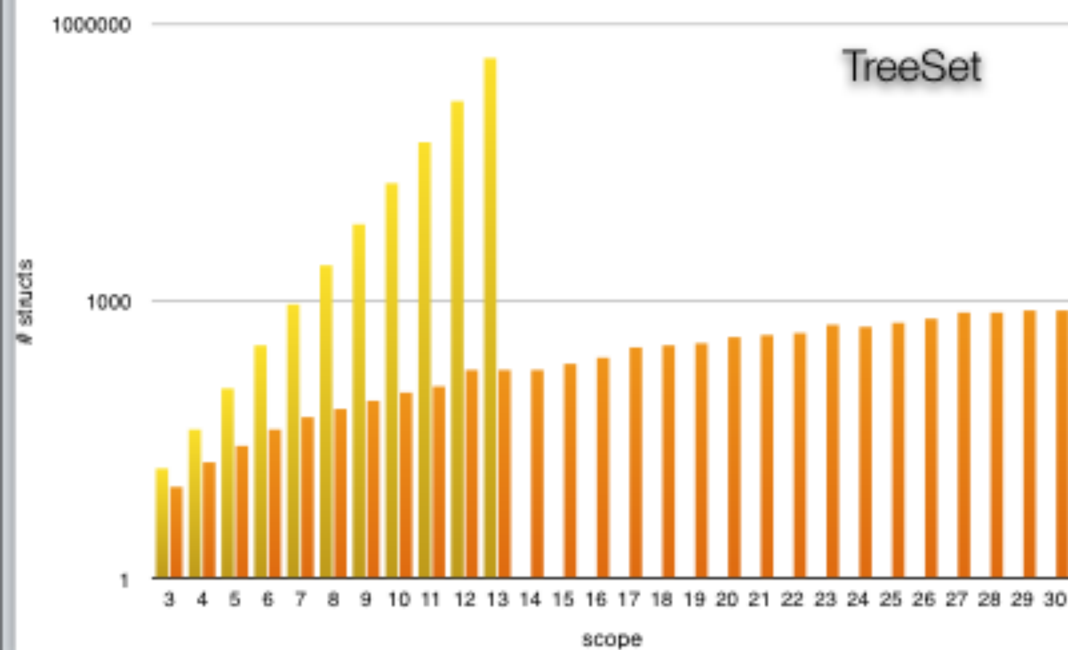
logarithmic scale

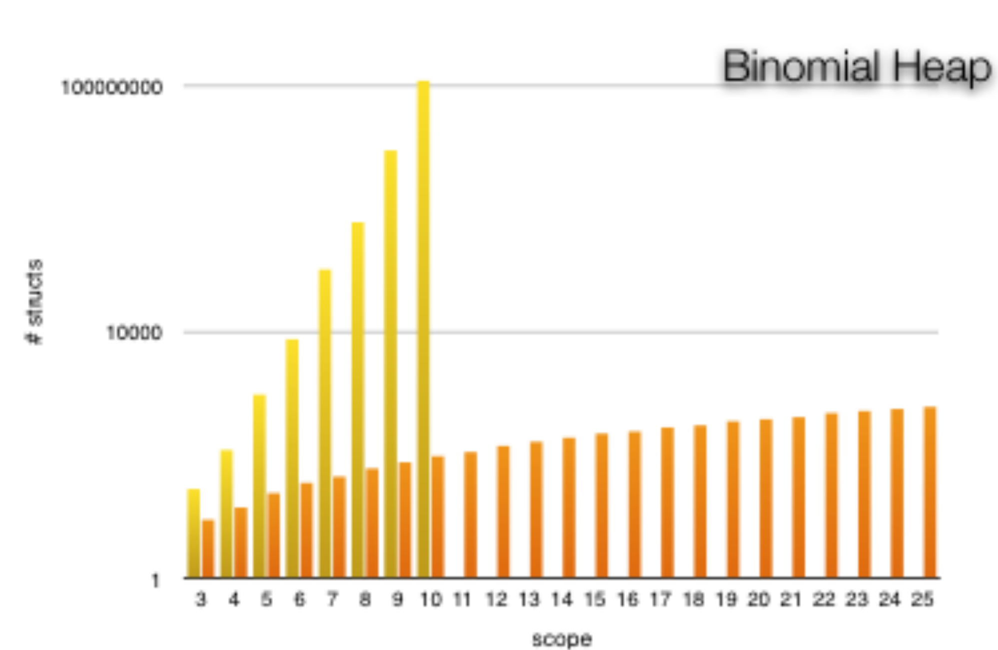





1000
1
3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
FET
scope
BET


structs
1000
1
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
FET
scope
BET


1000000
TreeSet
structs
1000
1
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
scope

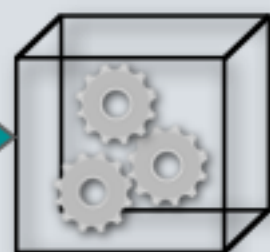

Binomial Heap
100000000
structs
10000
1
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
scope

AUTOMATED INVARIANT TUNING FOR TESTING

Rosner, Bengolea, Ponzio, Khalek, Aguirre, Khurshid, Frias, *Bounded Exaustive Test Input Generation from Hybrid Invariants*, OOPSLA 2014

AUTOMATED TEST INPUT GENERATION

valid inputs



Specification Styles

Declarative (based on some logical formalism)

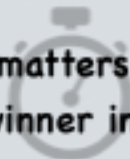
$$\forall n : Node \quad n \in Reach(head, next) \Rightarrow n \notin Reach(n.next, next)$$

Imperative (based on code)

```
boolean repOK() {
  Set visited = new HashSet();
  Node curr = head;
  while (curr!=null && visited.add(curr)) {
    curr = curr.next;
  }
  return (curr==null);
}
```

Constraint Solving (SAT, SMT)
Scalability tied to underlying solving technology

Generate & Filter
(Symbolic execution, Backtracking)
Highly sensitive to spec's structure



Spec Style matters for analysis
No clear winner in efficiency

PICKING SIDES

- Some constraints are easily coded imperatively,

- Some constraints are easily coded imperatively.

- Other constraints are more naturally expressed declaratively.

- Sometimes a constraint is already available as code – or as a declarative predicate.

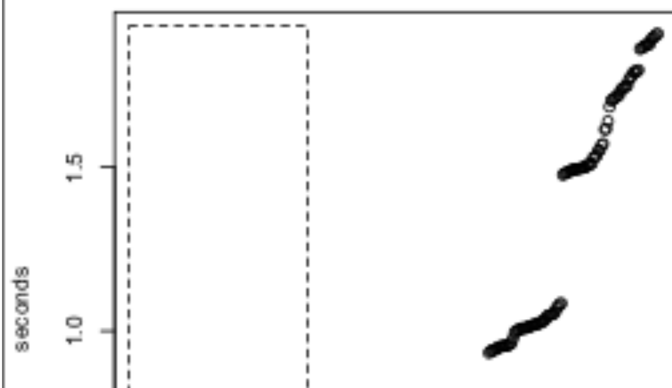
- Sometimes a constraint is already available (and/or easily expressed) on both sides.

EXPLOITING REDUNDANCY

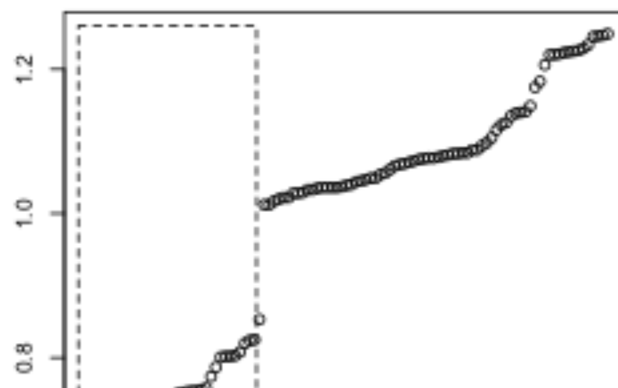
- Assume the availability of some constraints on both sides.
- Placing one constraint on one vs. the other side may have great impact on total generation time
- Can we automatically choose the best side for each constraint that is available on both?

Transcoping: try many different combinations on small scopes, propagate only the best to larger ones

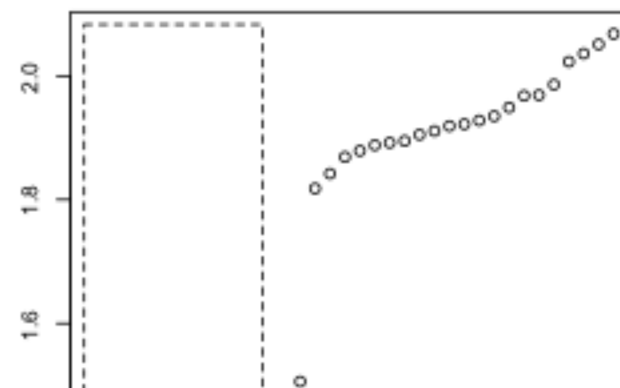
326 candidates (all from scope 6)

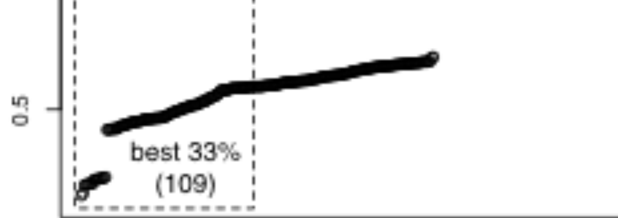


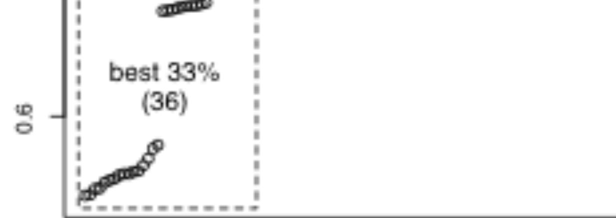
109 candidates (best 33% from scope 7)

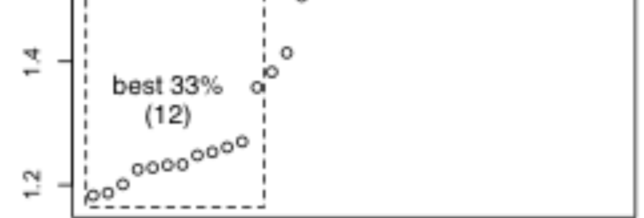


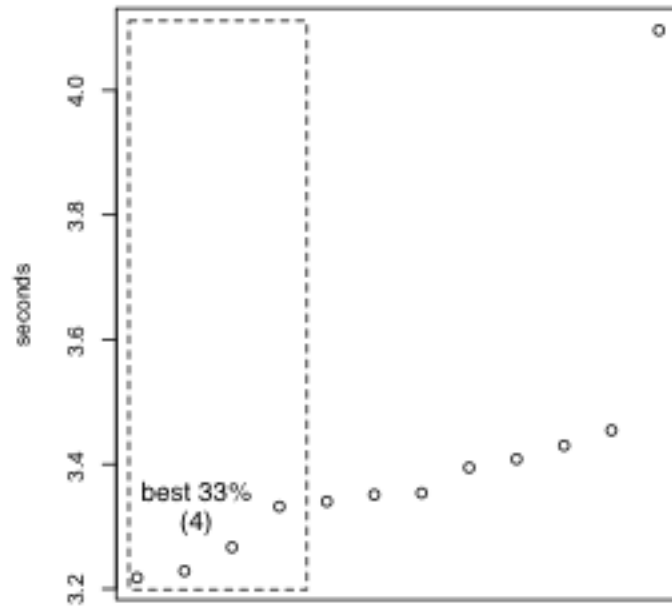
36 candidates (best 33% from scope 8)

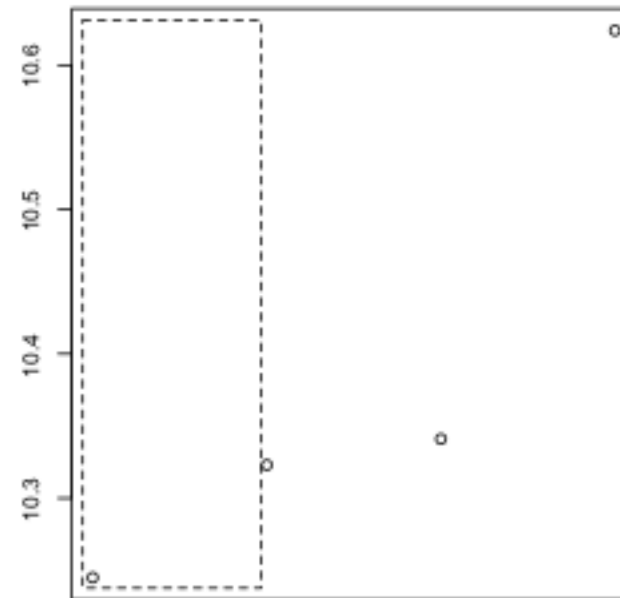



0.5
best 33%
(109)
index (i-th candidate, sorted by analysis time)


0.6
best 33%
(36)
index (i-th candidate, sorted by analysis time)


1.4
1.2
best 33%
(12)
index (i-th candidate, sorted by analysis time)

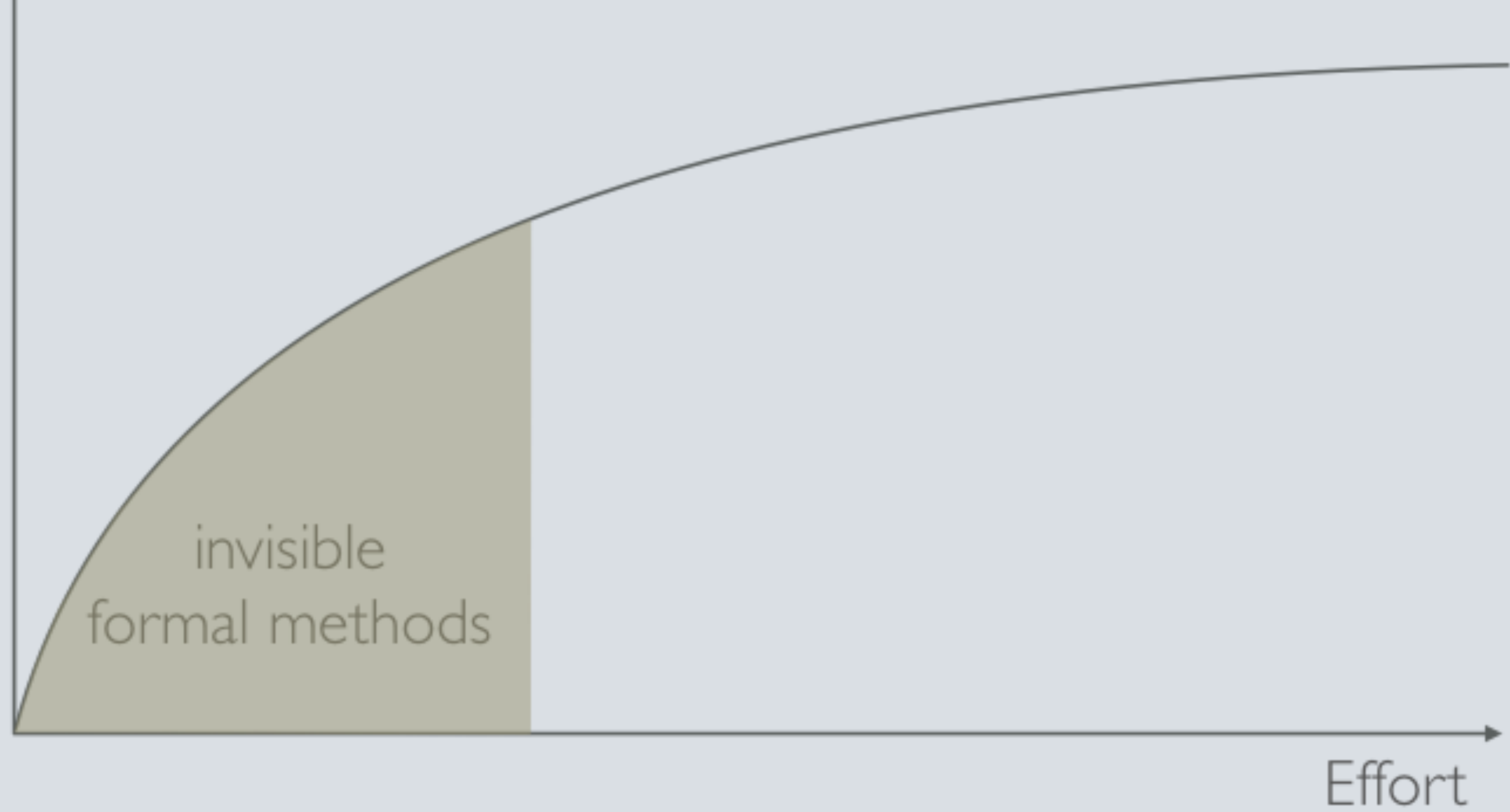

12 candidates (best 33% from scope 9)
seconds
4.0
3.8
3.6
3.4
3.2
best 33%
(4)
index (i-th candidate, sorted by analysis time)


4 candidates (best 33% from scope 10)
10.6
10.5
10.4
10.3
index (i-th candidate, sorted by analysis time)

AUTOMATED PROGRAM REPAIR

TAKING THE “INVISIBLE” PATH TOO FAR

Assurance



MUTATION BASED REPAIR

- Instead of using mutants to “inject” bugs...

¡use (anti-)mutants to suggest bug fixes!

EXAMPLE

```
public int countEven(int[] array) {
```

```
public int countEven(int[] array) {
  int even = 0;
  for (int i=0; i<=array.length; i++) {
    if (array[i] % 2 == 0) even++;
  }
  return even;
}
```

buggy program

candidate solutions

```
public int countEven(int[] array) {
  int even = 0;
  for (int i=0; i<array.length; i++) {
    if (array[i] % 2 == 0) even++;
  }
  return even;
}
```

```
public int countEven(int[] array) {
  int even = 0;
  for (int i=0; i>array.length; i++) {
    if (array[i] % 2 == 0) even++;
  }
  return even;
}
```

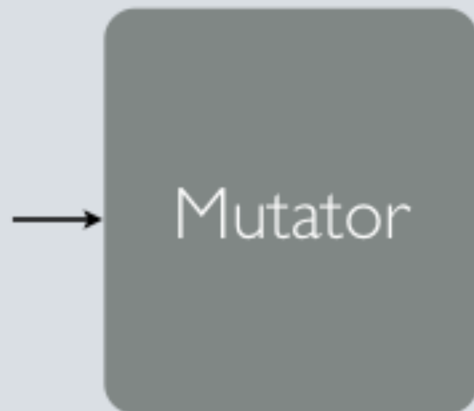
...

THE ORACLE PROBLEM

"Invisibility" approach:

Use Tests as Specs!

buggy program



mutants

correct program found?

OK

no fix?

X

ARE TESTS GOOD SPECS FOR PROGRAM REPAIR?

program	SearchRepair	AE	GenProg	TrpAutoRepair	total
checksum	0	0	8	0	29
digits	0	17	30	19	91
grade	5	2	2	2	226
median	68	58	108	93	168
smallest	73	71	120	119	155
syllables	4	11	19	14	109
total repaired	150	159	287	247	778

Fig. 7: Number of defects repaired by each technique. The total *column* specifies the total number of defects, and the total *row* specifies the total number of repaired defects.

ARE TESTS GOOD SPECS FOR PROGRAM REPAIR?

- Evaluation: Equip

Method	#V.	Suite	#Patch.	#Fix.	%Patch.	%Fix.
checksum	69	O	2	0	2.90%	0.00%
		O ∪ S100	0	0	0.00%	0.00%

programs with specs and double check patches

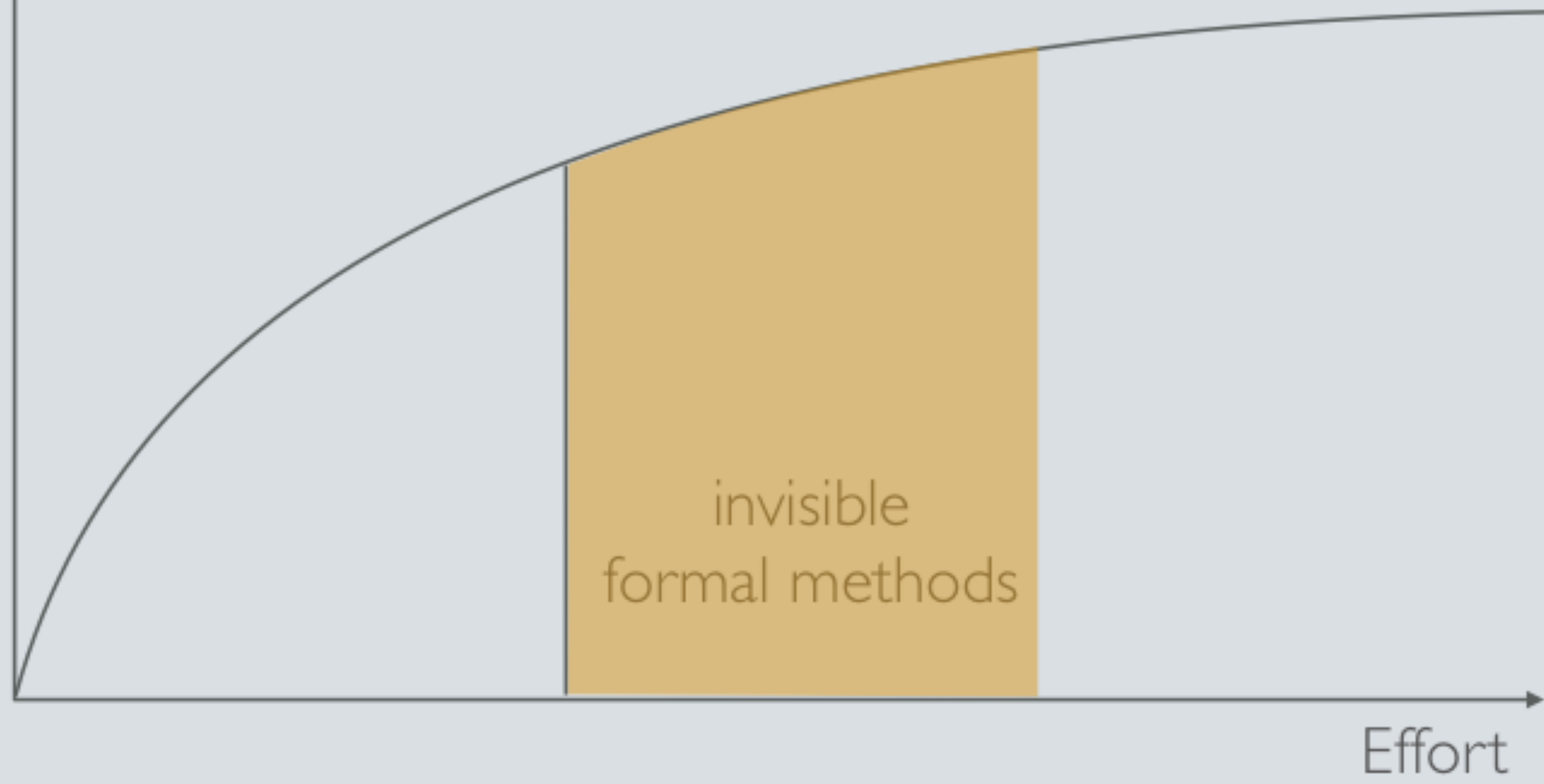
		O ∪ S1,000	0	0	0.00%	0.00%
digits	236	O	33	0	13.98%	0.00%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
grade	268	O	4	2	1.49%	0.75%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
median	232	O	124	14	53.45%	6.03%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
smallest	177	O	115	2	64.97%	1.13%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%
syllables	161	O	3	0	1.81%	0.00%
		O ∪ S100	0	0	0.00%	0.00%
		O ∪ S1,000	0	0	0.00%	0.00%

TABLE III
REPAIR STATISTICS FOR THE GENPROG AUTOMATIC REPAIR TOOL.

Zemín, Gutiérrez, Godio, Cornejo, Degiovanni, Regis, Aguirre, Frias, *An Analysis of the Suitability of Test-Based Patch Acceptance Criteria*, SBST 2017.

ADVOCATE FORMAL SPECIFICATION

Assurance


invisible
formal methods
Effort

THANK YOU!