NEC

Empowered by Innovation

Verifying C Programs Using SAT-based Model Checking

Satisfiability Solvers and Program Verification (SSPV)
August 11, 2006

Aarti Gupta
agupta@nec-labs.com

NEC Laboratories America
Princeton

Acknowledgements:
Pranav Ashar, Malay Ganai, Franjo Ivancic, Vineet Kahlon,
Sriram Sankaranarayanan, Ilya Shlyakhter, Chao Wang, Zijiang Yang

Outline

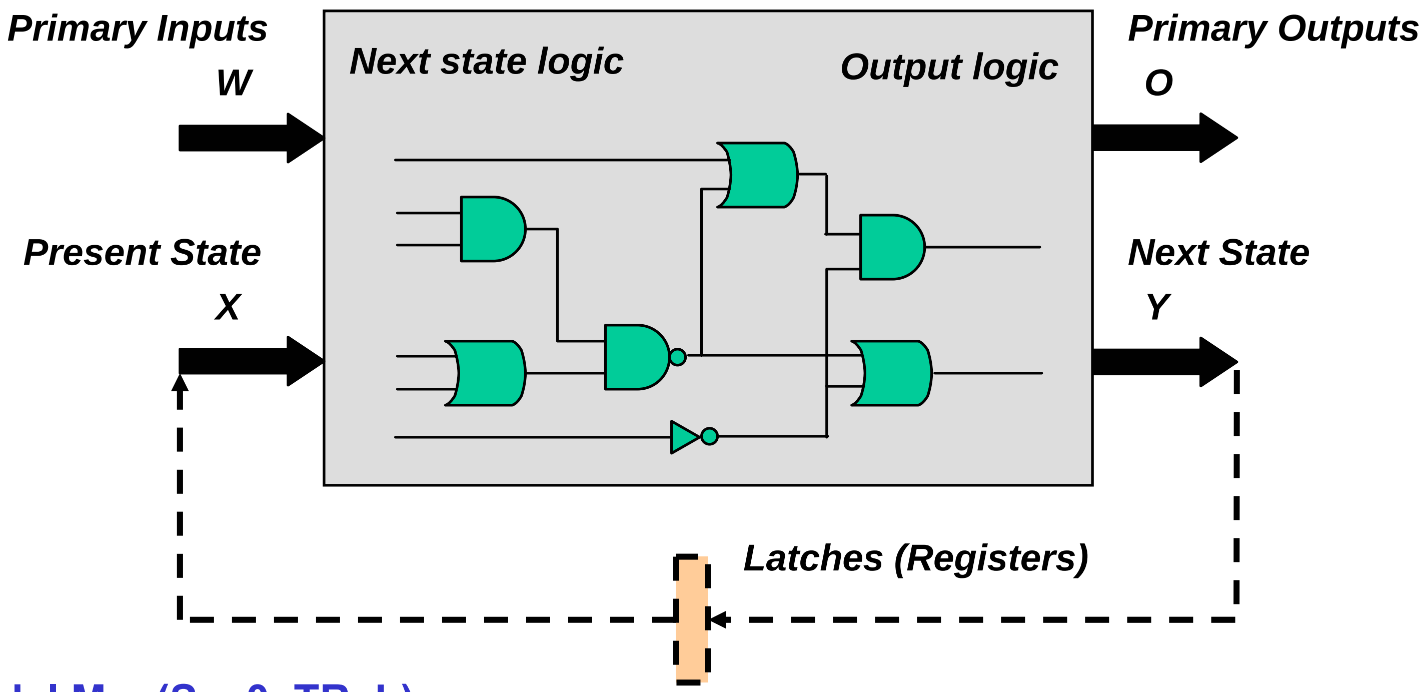
I. SAT-based verification methods

II. Verifying sequential C programs

III. Verifying multi-threaded C programs

I. SAT-based Verification Methods

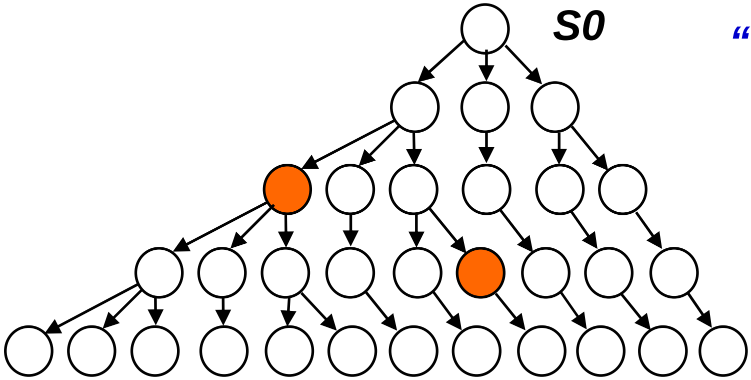
Hardware Circuit Model (Symbolic LTS)



- Model M = (S, s0, TR, L)
- Set of States S is encoded by a vector of binary variables *X*
  - Implemented as the outputs of latches (registers)
  - *Size of state space:* $|S| = 2^{|X|}$
- Initial state s0 comprises initial values of the latches
- Transition relation TR is implemented as next state logic (Boolean gates)
  - Y = TR(X, W), where TR is a Boolean function of present state X and inputs W
- Labeling L is implemented as output logic (Boolean gates)
  - O = f(X) or O = g(X,W)

Property Verification

- **Two Main Approaches:**
  - **Proof Approach**
    - **Exhaustive state space exploration, i.e. all states in the model are covered to check for property satisfaction**
    - **Very expensive for medium to large-sized models**
  - **Falsification Approach**
    - **State space search for bugs (counter-examples) only**
    - **Less expensive, but needs good search heuristics**



"Is there is a path from the initial state S0 to the bad state(s) where property fails?"

● ***State where the property fails***

- **We use both proof and falsification approaches**

NEC Hybrid (Circuit+CNF) SAT Solver

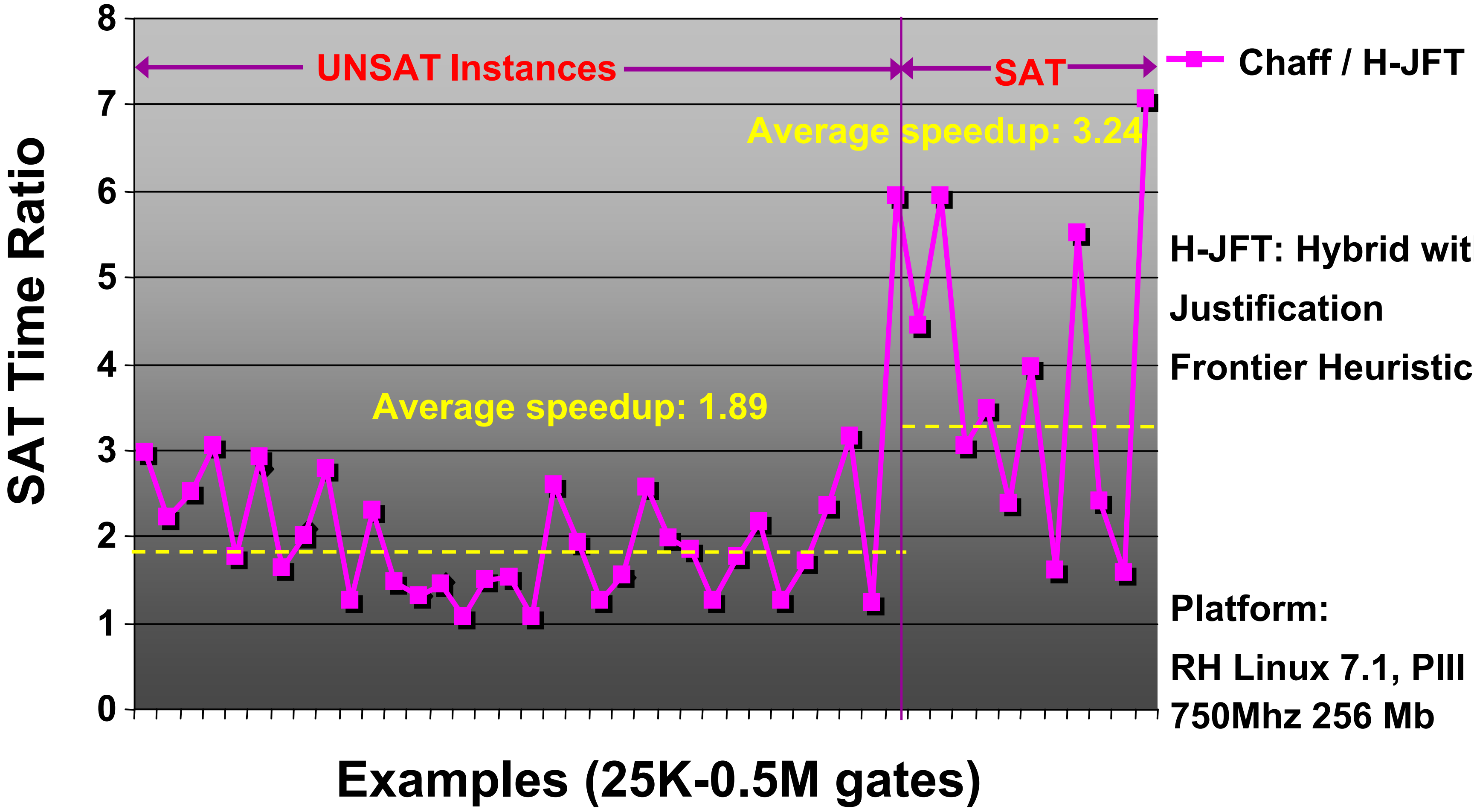
[Ganai *et al.* DAC 02]

- Works simultaneously on circuit and CNF representations
  - Original problem: Circuit form
  - Learned clauses: CNF

- Deduction Engine – Hybrid BCP
  - Circuit-based BCP on gates using *fast table lookup* [Kuehlmann *et al.* 01]
    - About 50% faster than BCP on corresponding clauses
  - CNF-based BCP on learned clauses using Chaff's *2-lit watching*
    - Lazy update is more effective on long clauses than a circuit-based chain/tree of gates
  - Records both clauses and gate nodes as reasons for implications
- Decision Engine
  - Use of circuit-based heuristics, such as *justification frontier*
    - More effective due to no decisions in "unobservable" parts
    - Can provide a solution with partial assignment
- Diagnostic Engine
  - Performs Grasp-style conflict analysis
  - Provides identification of *unsatisfiable core* [Zhang & Malik 03]
    - With additional heuristics for minimizing size of unsat core

NEC SAT Solver Results (w/Circuit heuristic JFT)

SAT Time Comparison – Chaff & NEC Hybrid w/ JFT

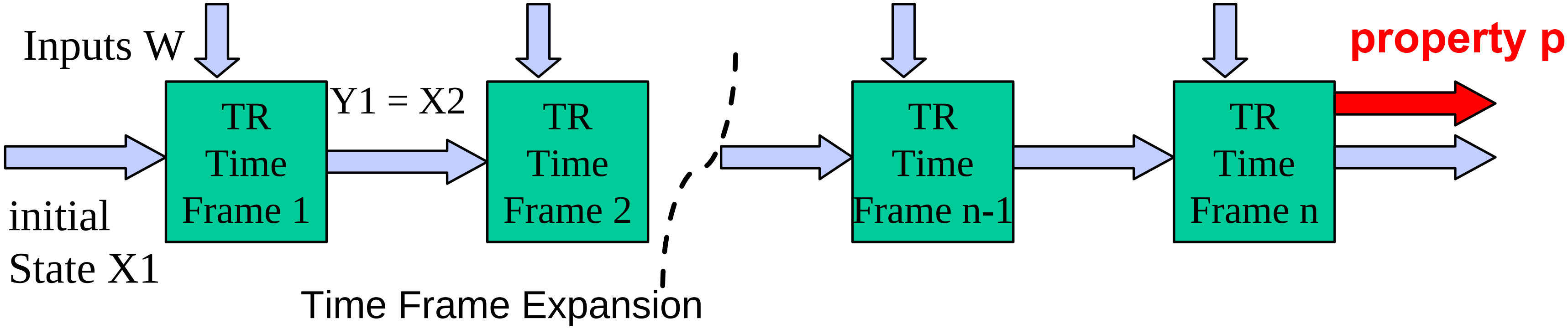


H-JFT: Hybrid with Justification Frontier Heuristic

Platform:

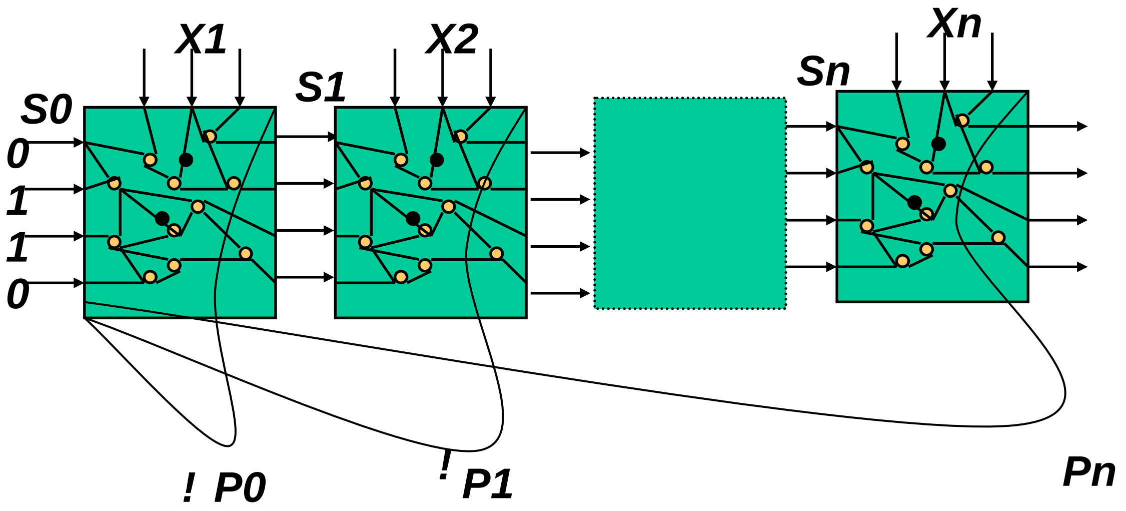
RH Linux 7.1, PIII 750Mhz 256 Mb

Bounded Model Checking (BMC)



- **BMC problem translated to a Boolean formula** **[Biere *et al.* 99]**
  - ***SAT(f_k)*** **(formula is satisfiable) ⇔ a bug exists at depth *k***
  - **Satisfiability of f_k is checked by a standard SAT solver**
- **Main ideas**
  - **Unroll transition relation up to bounded depth**
  - **Avoid computing sets of reachable states**
- **Falsification approach to search for bounded length bugs**
  - **Scales much better than BDD-based methods for hardware verification**
    - **BDDs can typically handle 100's of latches (state elements)**
    - **SAT can typically handle 10K's latches (state elements)**
  - **Incomplete in practice due to large completeness threshold**
- **Proofs by induction with increasing depth** **[Sheeran *et al.* 00]**
  - **Works well with additional BDD-based reachability invariants** **[Gupta *et al.* 03]**

Improving BMC Performance

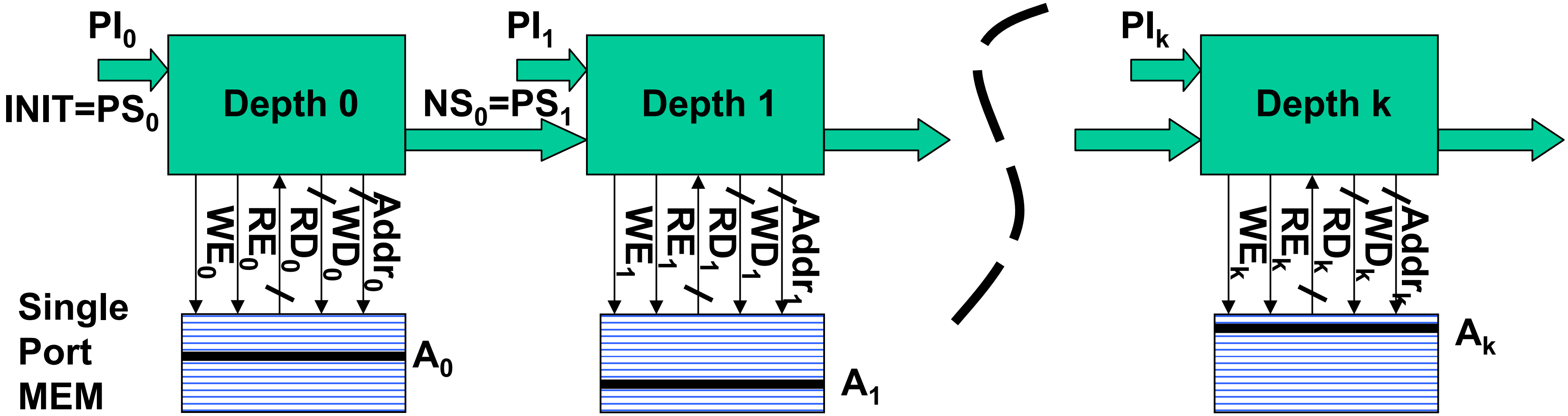


- **Dynamic circuit simplification** [Kuehlmann & Ganai 01]
- **Reuse of learned property constraints** [Ganai *et al.* 02]
- **Partitioning and incremental BMC translation** [Ganai *et al.* 05]
  - **Customized property translations into multiple SAT subproblems**
- **Hybrid SAT Solver** [Ganai *et al.* 02]
- **BDD Learning** [Gupta *et al.* 03]
- **BDD Constraints** [Gupta *et al.* 03]
- ***High-level BMC: SMT Solver, BMC-friendly model*** *[Ganai & Gupta. ICCAD 06]*

BDDs work really well on small problems – use them to supplement SAT

Efficient Memory Model: To Handle Embedded Memories

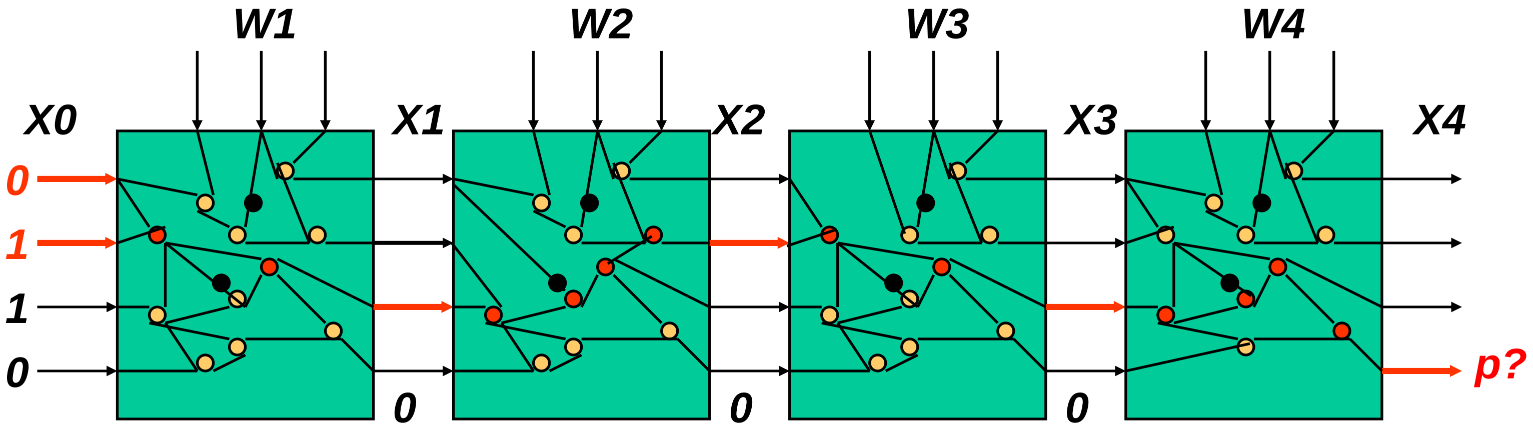
[Ganai *et al.* CAV 04]



- **EMM: Remove memories from model, but add memory constraints**
  - Data forwarding semantics maintained during BMC unrolling
    - Similar to interpreted memories in other work
  - Exclusivity of a matching read-write pair is captured explicitly
    - Significantly improves SAT solver performance
  - Constraints are represented efficiently
    - Circuit+CNF representation with a hybrid SAT solver works better than ITE representation

BMC with SAT Proof Analysis

- **BMC Problem: Is property *p* satisfiable at depth *k*?**

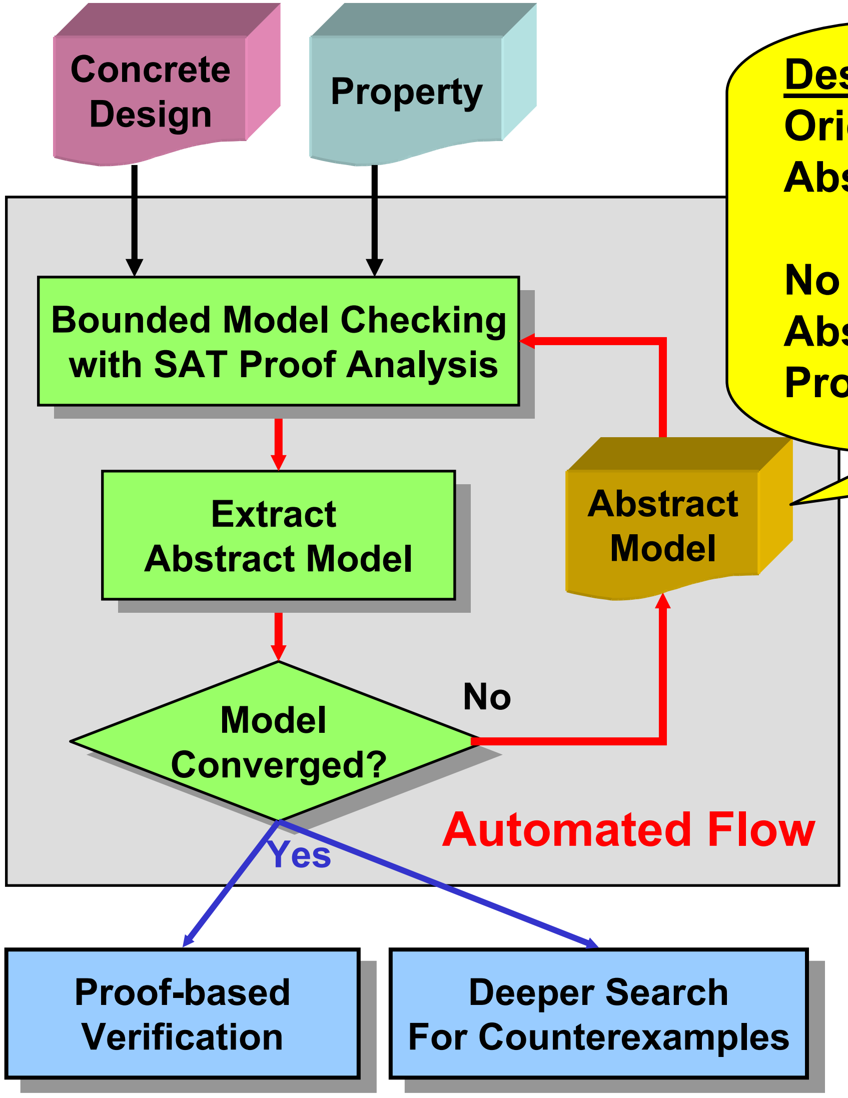


- **Suppose no bug at depth *k* because *p* is unsatisfiable**
  - **Derive an unsatisfiable core *R(k)* using SAT solver** [ZM03, MA03]
  - ***R(k)* is sufficient for the original problem to be unsatisfiable**

- **Abstraction based on Unsat Core of SAT Solver** [MA03, GGA03]
  - **Abstract model with core *R(k)* implies correctness at (up to) depth *k***
  - **If *k* is sufficiently large, the abstract model may be correct for *k'* > *k***
  - **Advantage: Typically *R(k)* is much smaller than entire design**

Proof-Based Iterative Abstraction (PBIA) using SAT

[Gupta *et al.* ICCAD 03]



Design D1
Original Design: 12716 bits, 416 k gates
Abstract Model: 71 bits, 2 k gates

No bug on full design (depth 120 in 1500s)
Abstraction in 1500s (6 iterations)
Proof on abstract model in 30 s

Proof Methods:

1. BDD-based model checking
2. SAT-based model checking
3. SAT-based induction

Symbolic Model Checking

[McM 90, BCL+94]

X: present state variables
Y: next state variables
W: input variables

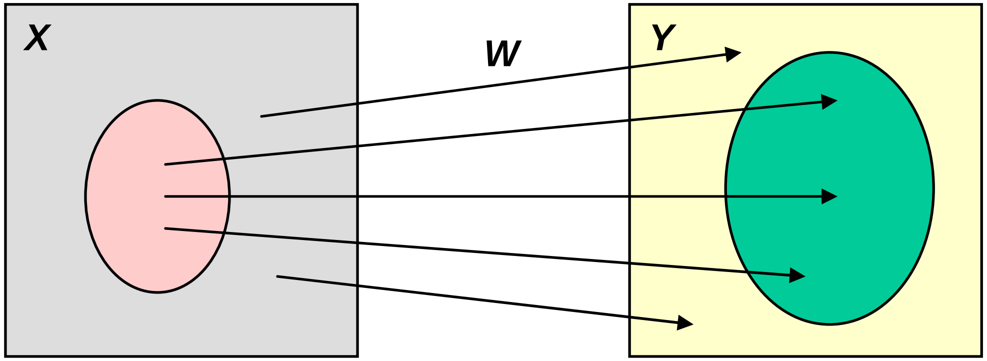
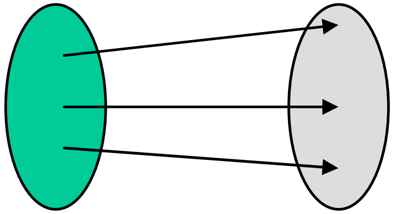


Image Computation

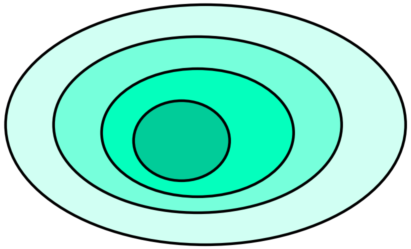
$$Image(Y) = \exists\ X, W.\ \ T(X,W,Y) \wedge\ From(X)$$

- **Related operations**

Pre-Image Computation



Fixpoint Computation



- **Core steps of many applications**
  - **equivalence checking, reachability analysis, model checking …**

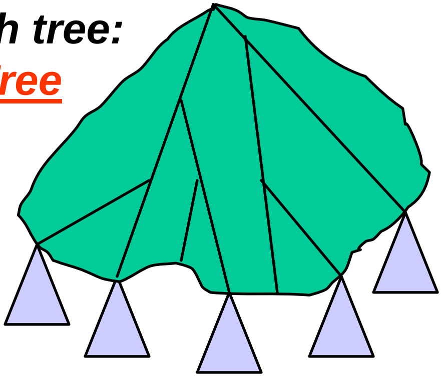
Enumerating All Solutions

- Search space: all values of variables (X, W, Z, Y)

	BDD DAGs	*SAT Decision Tree*
Flexibility	*Low (fixed ordering)*	*High (no restriction on decisions)*
Solution Sharing	*High (canonical)*	*Low (non-canonical)*

- BDD+SAT Strategy: keep flexibility, but avoid cube enumeration

[Gupta *et al.* FMCAD 00]



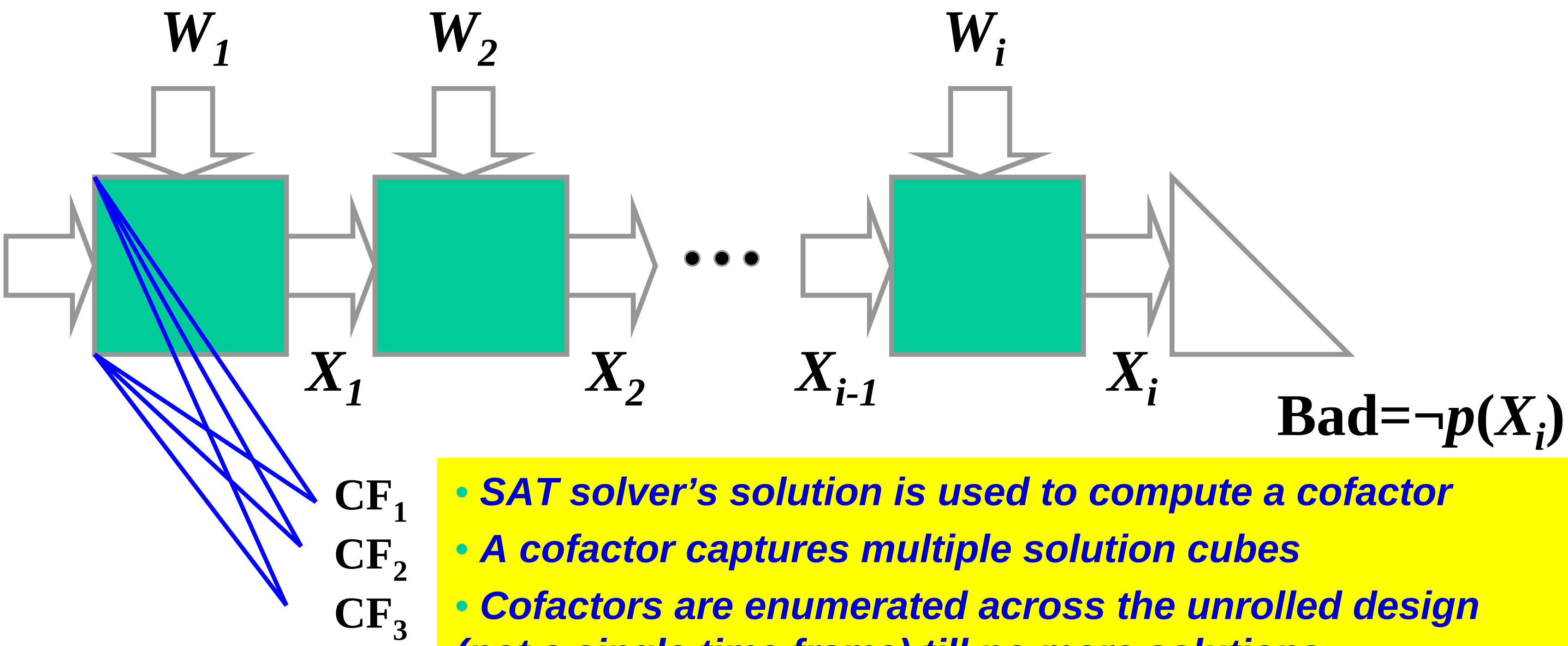
Top level search tree:
SAT Decision Tree

Leaves of SAT search tree:
BDD sub-problems

SAT-based UMC using Circuit Cofactoring (CC)

[Ganai *et al.* ICCAD 04]

- **Symbolic backward traversal using unrolled TR**



- ***SAT solver's solution is used to compute a cofactor***
- ***A cofactor captures multiple solution cubes***
- ***Cofactors are enumerated across the unrolled design (not a single time frame) till no more solutions***

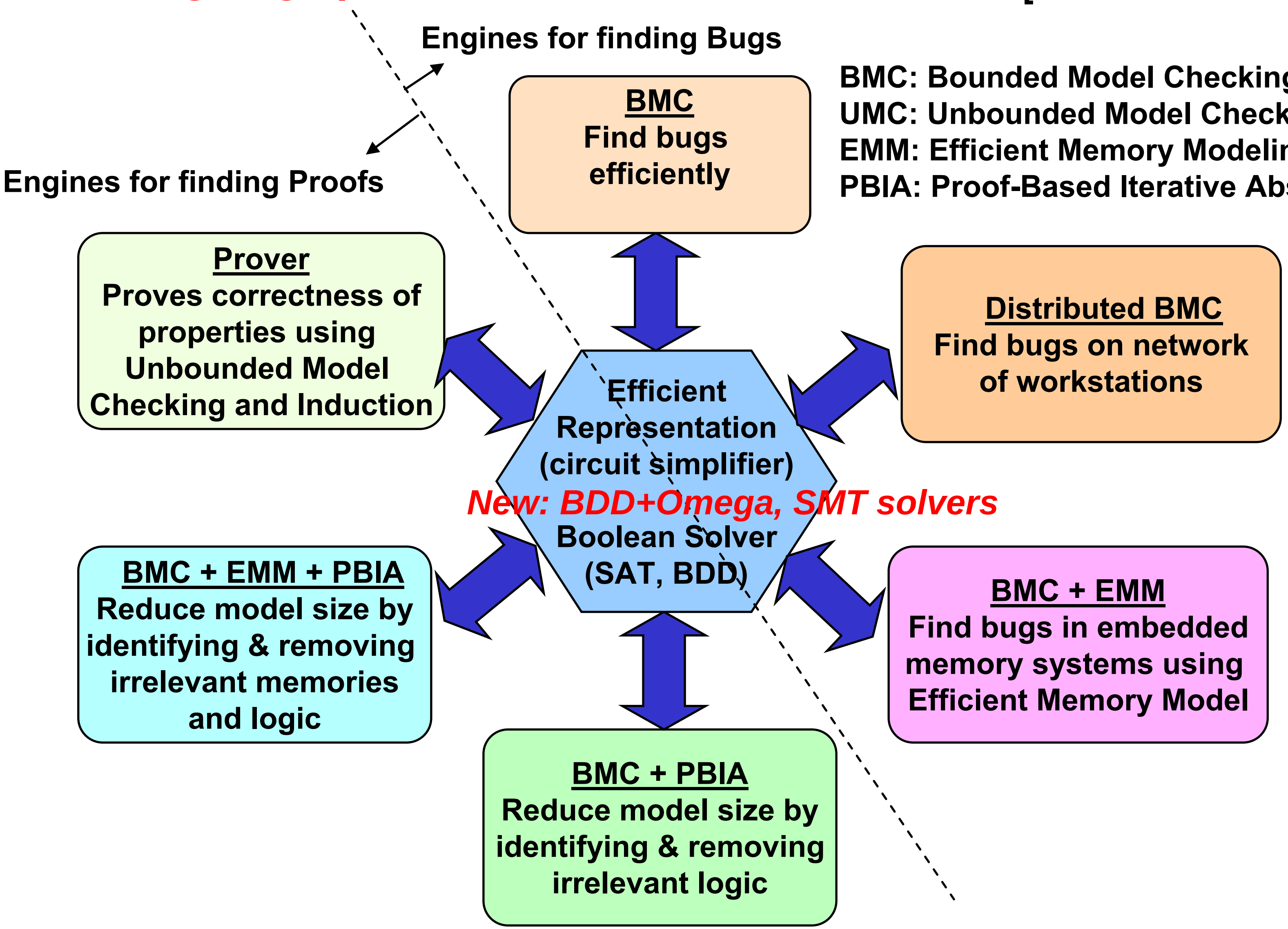
- **State sets (represented as circuit cofactors) may blow up**
  - **Performance is not as good as SAT-based BMC (search for bugs), which avoids computation of state sets**
- **Complementary to BDD-based UMC for deriving proofs**

NEC's VeriSol (DiVer) Verification Platform

Interesting large problems are within reach!

[Ganai *et al.* TACAS 05]

BMC: Bounded Model Checking
UMC: Unbounded Model Checking
EMM: Efficient Memory Modeling
PBIA: Proof-Based Iterative Abstraction



Engines for finding Bugs

Engines for finding Proofs

BMC
Find bugs efficiently

Prover
Proves correctness of properties using Unbounded Model Checking and Induction

Distributed BMC
Find bugs on network of workstations

Efficient Representation (circuit simplifier)

New: BDD+Omega, SMT solvers

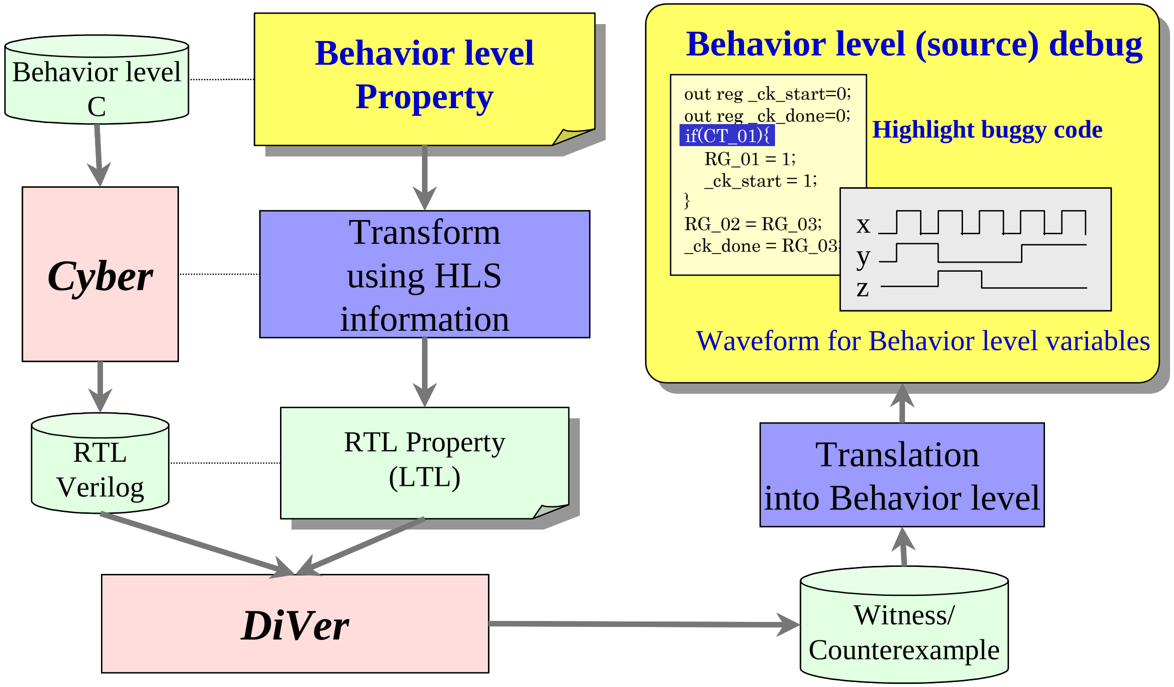
Boolean Solver (SAT, BDD)

BMC + EMM + PBIA
Reduce model size by identifying & removing irrelevant memories and logic

BMC + EMM
Find bugs in embedded memory systems using Efficient Memory Model

BMC + PBIA
Reduce model size by identifying & removing irrelevant logic

NEC's High Level Synthesis Framework


Behavior level C
Behavior level Property
Cyber
Transform using HLS information
RTL Verilog
RTL Property (LTL)
DiVer
Witness/ Counterexample
Translation into Behavior level
Behavior level (source) debug
Highlight buggy code
x
y
z
Waveform for Behavior level variables

```
out reg _ck_start=0;
out reg _ck_done=0;
if(CT_01){
    RG_01 = 1;
    _ck_start = 1;
}
RG_02 = RG_03;
_ck_done = RG_03;
```

- **Cyber Work Bench (CWB)**
  - Developed by NEC Japan (Wakabayashi et al.)
  - Automatically translates behavioral level design (C-based) to RTL design (Verilog)
  - Generates property monitors for RTL design automatically
- **VeriSol is integrated within CWB**
  - Provides verification of RTL designs
  - Has been used successfully to find bugs by in-house design groups

II. Verifying Sequential C Programs

Model Checking Software Programs

C Program

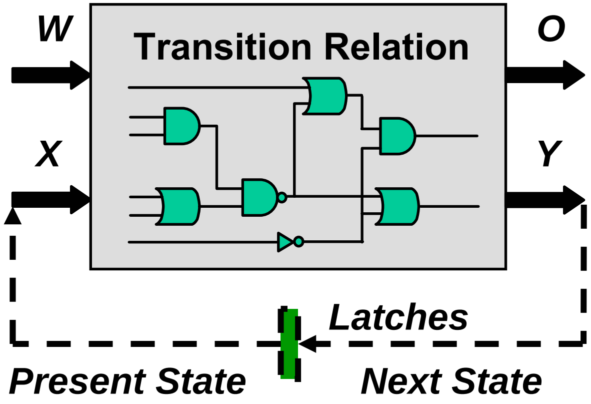
```
void bar() {
    int x = 3 , y = x-3 ;
    while ( x <= 4 ) {
        y++ ;
        x = foo(x);
    }
    y = foo(y);
}

int foo ( int l ) {
    int t = l+2 ;
    if ( t>6 )
        t - = 3;
    else
        t --;
    return t;
}
```

Huge gap !

Finite state circuit model

M = (S,s0,TR,L)



X: present state variables
Y: next state variables
W: input variables

Challenges:

- Rich data types
- Structures and arrays
- Pointers and pointer arithmetic
- Dynamic memory allocation
- Procedure boundaries and recursion
- Concurrent programs

Intermediate Representation

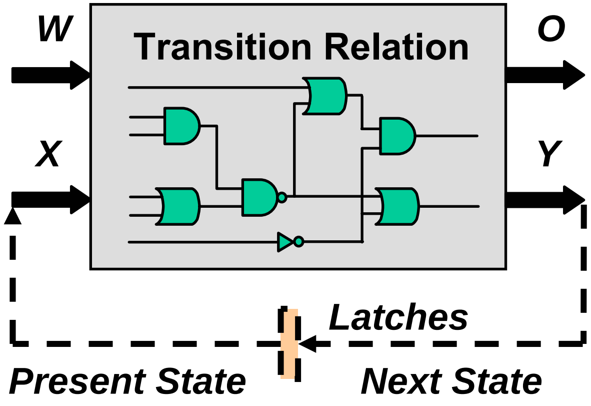
C Program

```
void bar() {
    int x = 3 , y = x-3 ;
    while ( x <= 4 ) {
        y++ ;
        x = foo(x);
    }
    y = foo(y);
}

int foo ( int l ) {
    int t = l+2 ;
    if ( t>6 )
        t - = 3;
    else
        t --;
    return t;
}
```

CFG
Control Flow Graph

M = (S,s0,TR,L)

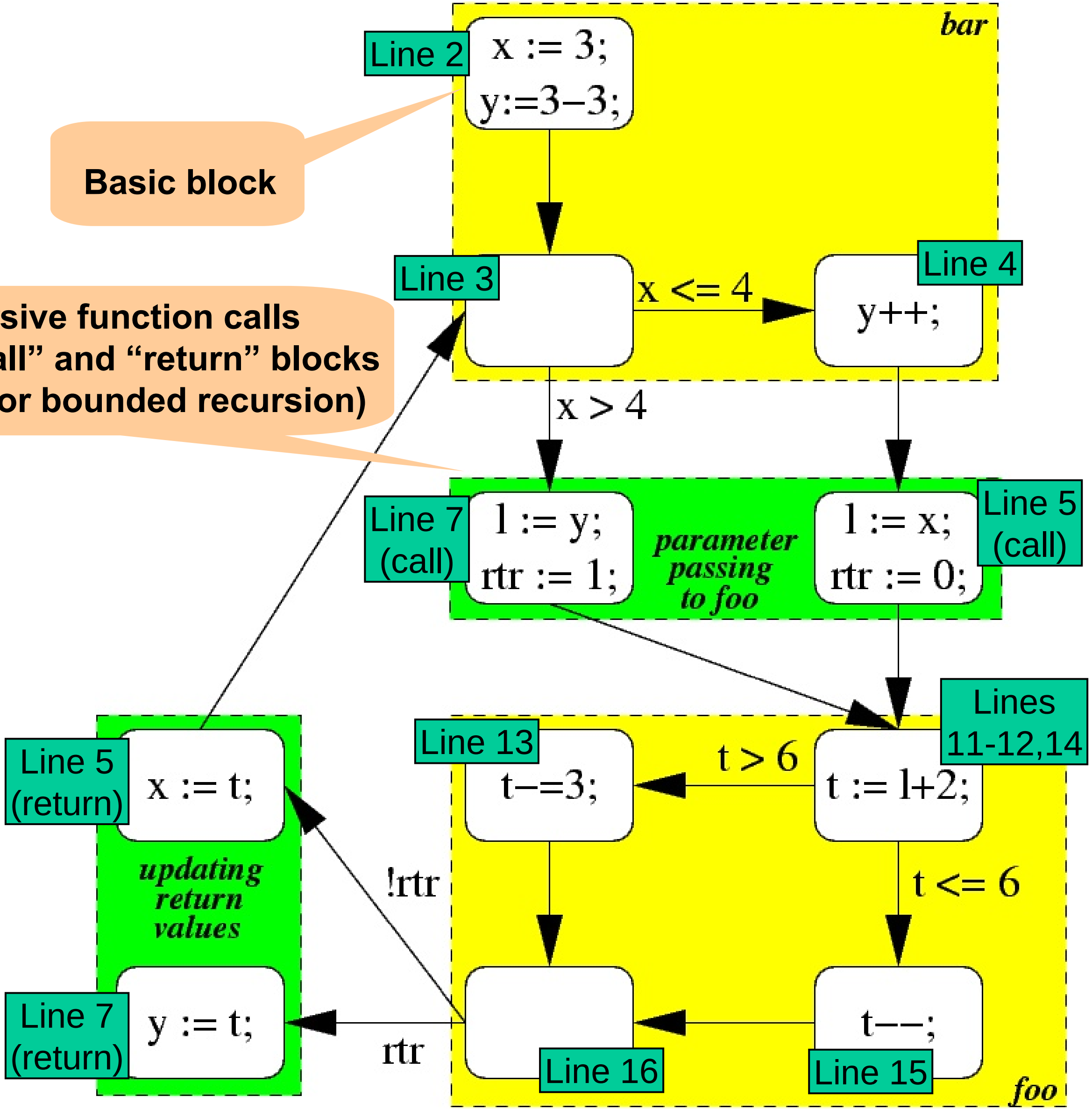


X: present state variables
Y: next state variables
W: input variables

- *Annotated* Control Flow Graph
  - Language-independent intermediate representation
  - Provides the basis for several optimizations (compilers, program analysis)
  - Allows separation of model building/reduction from model checking

Modeling C Programs: An Example

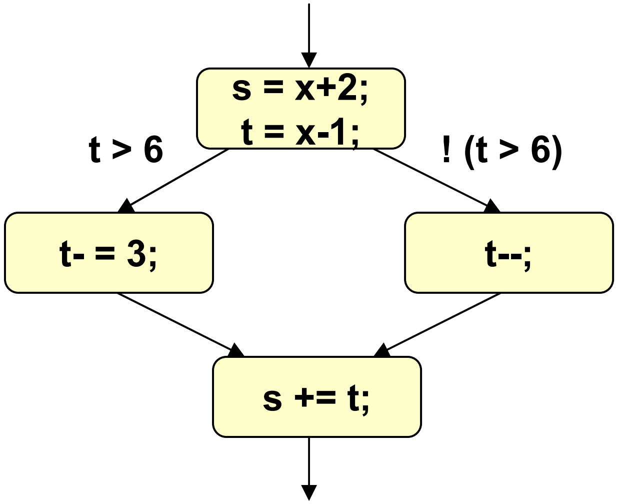
```
1: void bar() {
2:     int x = 3 , y = x-3 ;
3:     while ( x <= 4 ) {
4:         y++ ;
5:         x = foo(x);
6:     }
7:     y = foo(y);
8: }
9:
10: int foo ( int l ) {
11:     int t = l+2 ;
12:     if ( t>6 )
13:         t - = 3;
14:     else
15:         t --;
16:     return t;
17: }
```



Basic block

Non-recursive function calls
use special "call" and "return" blocks
(Add a stack for bounded recursion)

Automatic Translation of CFG to Circuit Model



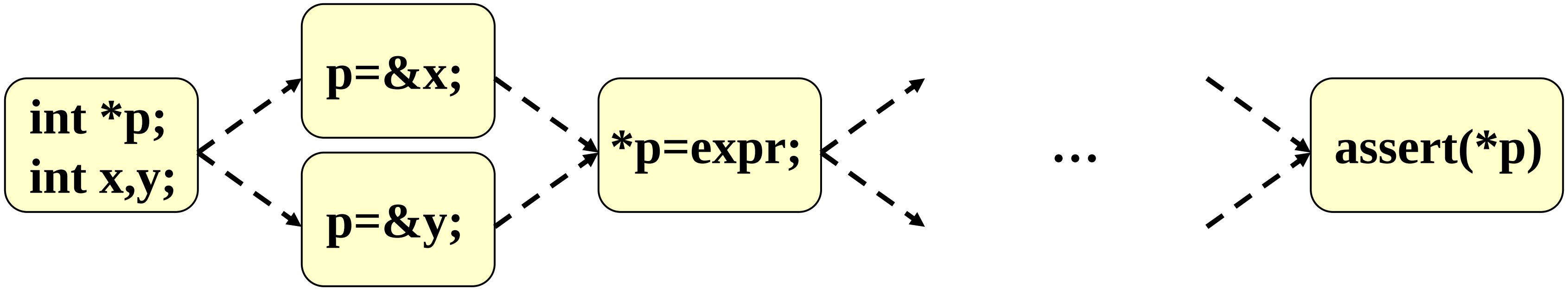
CFG ~ finite (control + data) state machine

Basic blocks ~ control states (encoded using pc)
Values of program variables ~ data states
Guarded transitions ~ transition relation for control states
Parallel assignments ~ transition relation for data states

Bit-level accurate models
(similar to high-level synthesis)

- **What about the challenges?**
  - **Rich data types, Structures and arrays**
    - **Consider only finite integer types, and convert/flatten other types**
  - **Pointers and pointer arithmetic**
    - **Convert to a pointer-less description** **[Semeria & De Micheli 98]**
  - **Dynamic memory allocation**
    - **To obtain a finite state verification model, consider bounded data only**
  - **Procedure boundaries and recursion**
    - **To obtain a finite state verification model, consider bounded recursion only**
      - Alternative: Pushdown models **[Ball & Rajamani 01]**
  - **Concurrent programs**
    - **Each thread can be represented by a separate CFG, with shared variables**

Modeling Pointers


int *p;
int x,y;

p=&x;

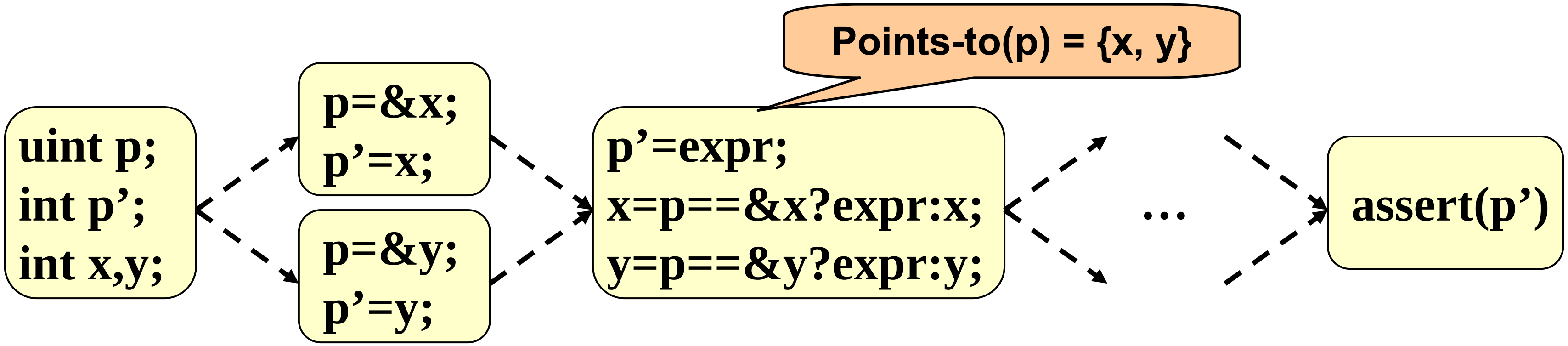
p=&y;

*p=expr;

…

assert(*p)

- **Pointers can be modeled using additional variables and inferred conditional assignments** [Semeria & De Micheli 98]


Points-to(p) = {x, y}

uint p;
int p';
int x,y;

p=&x;
p'=x;

p=&y;
p'=y;

p'=expr;
x=p==&x?expr:x;
y=p==&y?expr:y;

…

assert(p')

  - Advantage: Decrease in number of live variables
    - Exploited by back-end model checking techniques
  - Disadvantage: Model size increases
    - SAT-solving not as dependent on number of register as BDDs
  - Accuracy of pointer analysis can be traded off with model checking

Back-end Verification of Software Models

- Bugs (reachability of error labels) can be found by using SAT-based BMC on the software models [Ivancic *et al.* ISoLA 04]
  - Unrolling corresponds to a block-wise execution on the CFG

- Proofs can be derived by using SAT-based or BDD-based unbounded model checking on the software models
  - Typically the number of variables in the software model is very large

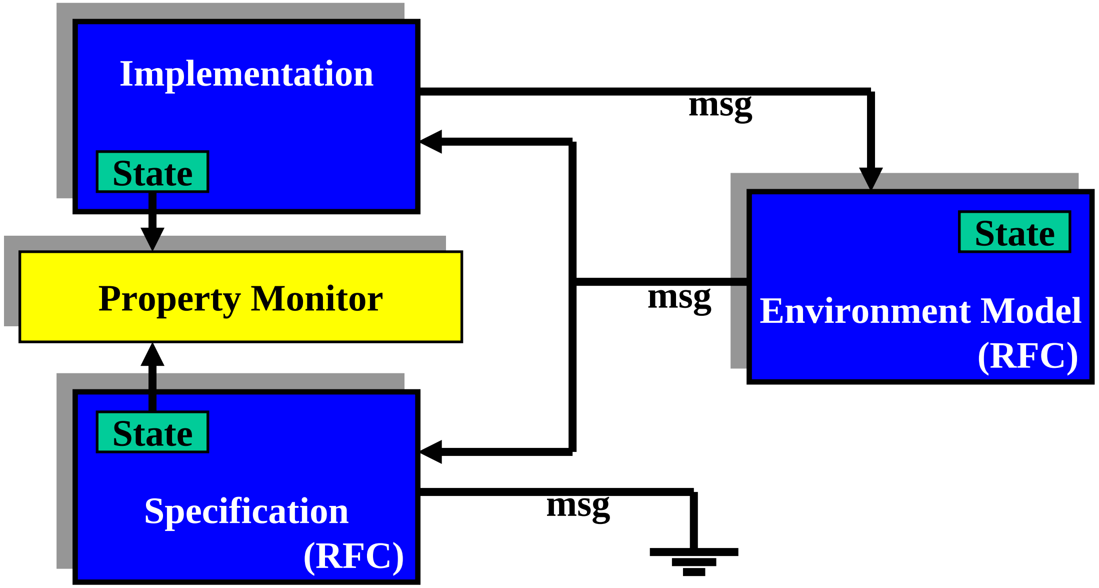
- Back-end verification is performed by VeriSol
  - VeriSol has been highly optimized for circuit-based models
  - Customized SAT heuristics for software models, based on information from our translation
    - H1: The basic block (i.e. pc) variables are more important than program variables (i.e. datapath variables) in SAT search
    - H2: Each basic block typically contains a small number of successor basic blocks
  - Disjunctive BDD-based image computation better for software [Wang *et al.* 06]
    - Quantifies away non-live variables

Case Study for SAT-based BMC

- Point-to-Point Protocol (PPP)
  - Analyzed LCP (link control protocol) part of PPP that establishes, configures, and tests a data-link connection
  - Specification is given as RFC 1661
  - Linux implementation contains about 2000 lines of C code
  - Property: Implementation adheres to specification [Alur& Wang 01]

RFC 1661

States Events	Req-Sent	Opened
Close	Term-Req goto Closing	Term-Req goto Closing
Conf-Ack	goto Ack-Rcvd	goto Req-Sent
Term-Ack		Conf-Req goto Req-Sent
Term-Req	Term-Ack	Term-Ack goto Stopping

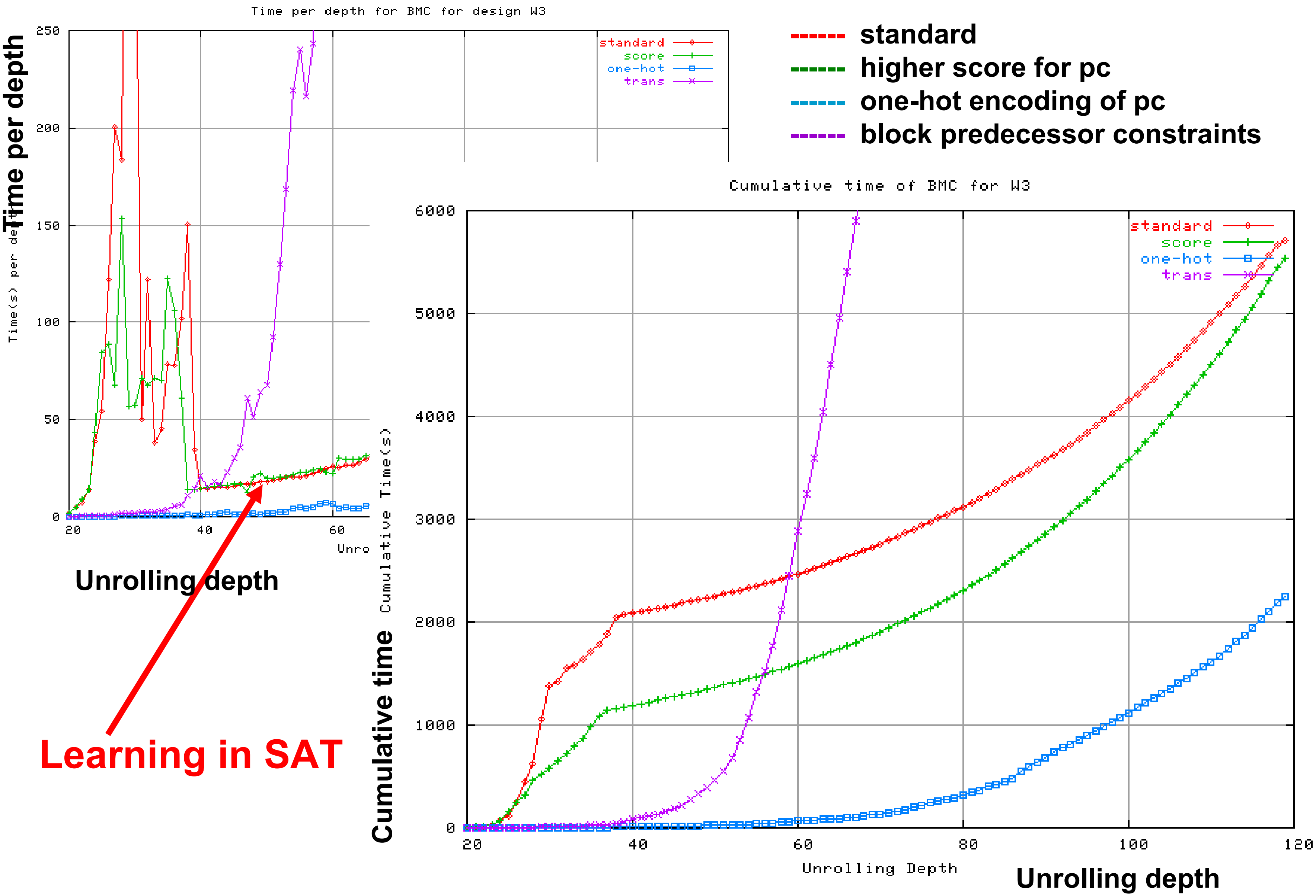


```
static void fsm_rtermack(fsm *f){
  switch (f->state) {
    /* other cases here */
    case OPENED:
      if ( f->callbacks->down)
        (*f->callbacks->down)(f);
        /* informing upper layers */
      fsm_sconfreq(f,0);
      break ;
  }
}
```

Public implementation

Missing:
f->state = REQSENT;

Results for PPP Case Study



Time per depth

Unrolling depth

Learning in SAT

Cumulative time

Unrolling depth

------ standard

------ higher score for pc

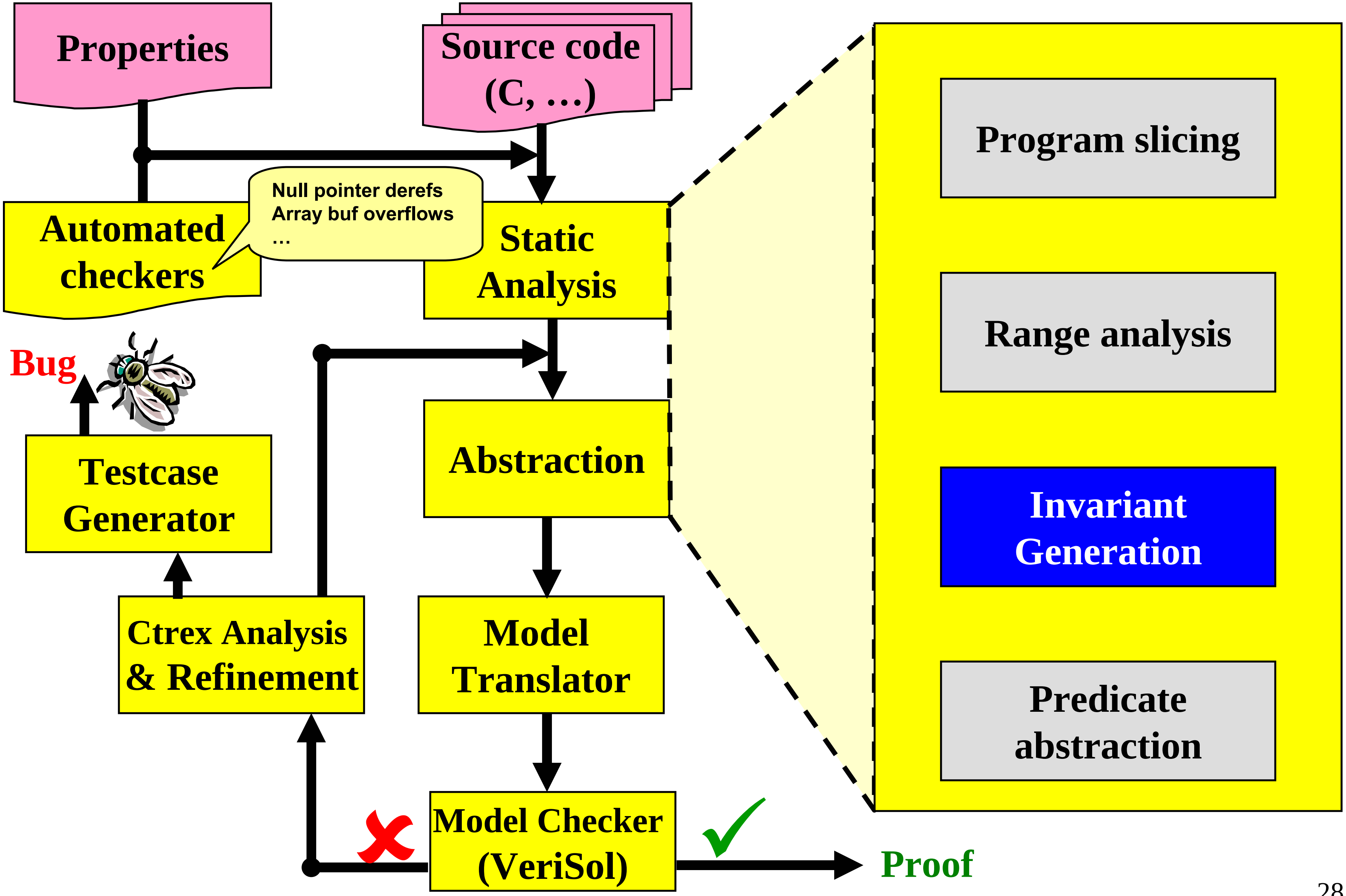
------ one-hot encoding of pc

------ block predecessor constraints

Supplementing Model Checking

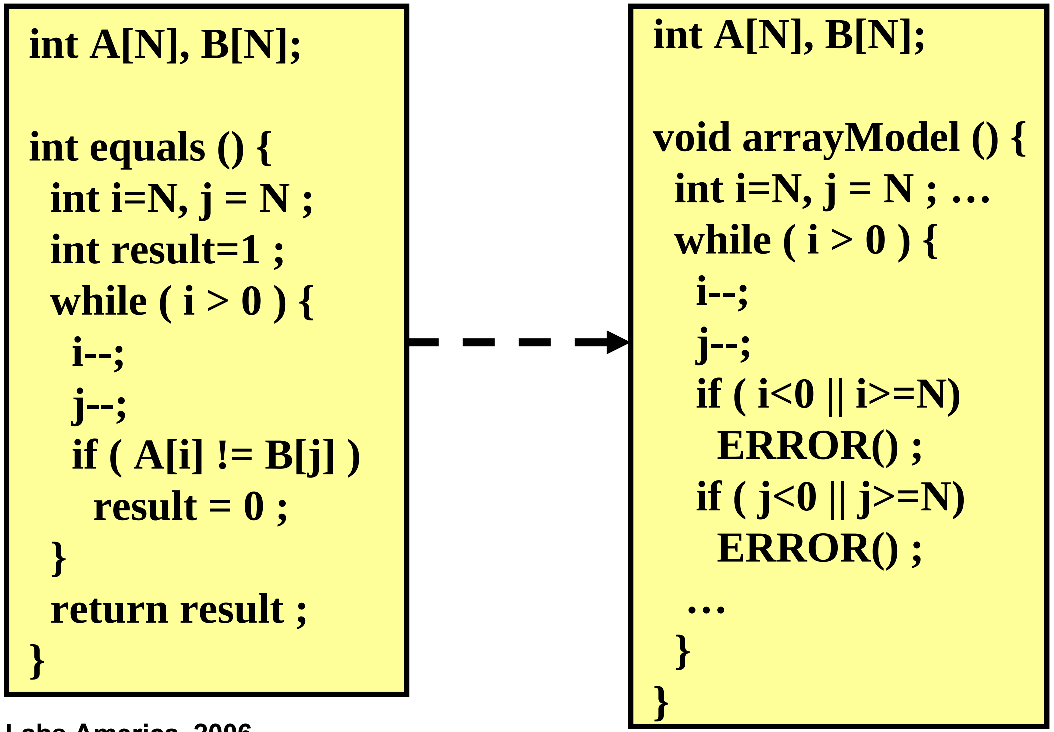
- **Main problem**
  - **Verification models generated directly from CFGs are too large**
- **Strategies**
  - **Use predicate abstraction and refinement** **[Slam, Blast]**
    - **Despite localization techniques, this frequently blows up** **[Jain *et al.* 05]**
    - **Does not work well on programs with pointers**
  - **Use light-weight analyses on CFGs in order to reduce size of the generated verification model**
    - **Program slicing (property-based)**
    - **Range analysis (to bound #bits per variable)**
    - **Constant folding (really helps in context of memory modeling)**
  - **These have helped significantly in reducing model size**
    - **e.g. Range analysis typically provides 80% reduction in #bits**
  - **More recently, we have used "cheap" static program analyses**
    - **Static Invariant Generation**

F-Soft Verification Platform



Static Invariant Generation

- **Can prove correctness of many properties** [Sankaranarayanan et al. SAS 06]
  - Array buffer overflows, null pointer dereferences, …
  - Significant reduction in # properties to be passed to model checker
- **Can reduce search space during model checking** [Ganai & Gupta ICCAD 06]
  - Additional constraints to the SAT solver improve performance
- **Can avoid divergence on loops in predicate abstraction** [Jain *et al.* CAV 06]
  - Known relationships can reduce number of spurious counterexamples



```
int A[N], B[N];

int equals () {
  int i=N, j = N ;
  int result=1 ;
  while ( i > 0 ) {
    i--;
    j--;
    if ( A[i] != B[j] )
      result = 0 ;
  }
  return result ;
}
```

```
int A[N], B[N];

void arrayModel () {
  int i=N, j = N ; …
  while ( i > 0 ) {
    i--;
    j--;
    if ( i<0 || i>=N)
      ERROR() ;
    if ( j<0 || j>=N)
      ERROR() ;
   …
  }
}
```

Invariants:
$0 \leq i \leq N$
$0 \leq j \leq N$
$i{=}{=}j$

Octagon Abstract Domain [Mine *et al.]*

- Allows discovering invariants of form ax+ by <= c where a, b $\in$ {-1, 0, 1}

- Can be computed using standard data flow analysis
  - For n program variables
  - O(n^3) time complexity
  - O(n^2) space requirements
  - Typically, small variable packs are used for fast analysis

Experiments on Industry Programs

Array buffer overflow checks

			Without Octagon Invariants					With Octagon Invariants				
	KLOC	# Checks	# P by SA	# P by SAT	# B by SAT	# None	Time (sec)	# P by SA w/ Invar	# P by SAT	# B by SAT	# None	Time (sec)
f1	0.5	64	32	9	0	23	596	64	0	0	0	15
f2	1.1	16	8	6	0	2	564	16	0	0	0	66
f3	1.1	18	8	5	2	3	572	16	0	2	0	104
f4	1.2	22	10	6	3	3	478	18	1	3	0	195
f5	1.2	10	0	0	4	6	584	6	0	4	0	401
f6	1.6	26	8	6	8	4	579	18	0	8	0	197
f7	1.8	28	4	8	4	4	589	12	4	4	0	325
f8	3.6	280	267	13	0	0	144	280	0	0	0	140

Note: #P by SA = # Proofs by Static Analysis, # P by SAT = #Proofs by SAT,
#B by SAT = # Bugs by SAT, # None = unresolved

- Several interesting improvements with Octagon invariants
  - Number of unresolved (#None) checks is reduced (here, 0)
  - Provides overall performance improvement
  - Last example: not much extra cost for Proofs by SAT

Invariants and Predicate Abstraction

[Jain *et al.* CAV 06]

- Main idea: Predicate abstraction may require many refinement iterations to discover some of the “cheap” invariants
  - Generate invariants statically as a pre-processing step
- Invariants are used to improve:
  - Abstraction computation: Transition relation strengthening
  - Refinement
    - Weakest pre-conditions provide savings in abstraction computation, but can diverge on many loops
    - External loop invariants can frequently overcome this limitation
- Experimental Results
  - Reduction in # abstraction-refinement iterations by 54%
  - Reduction in maximum # predicates (at a program location) by 58%
  - Reduction in overall runtime by 69%

Disjunctive Abstract Domains: Path Sensitive Analysis

```
int x[10];
int len, ok;

if ( len >= 0 && len < 10)
    ok = 1;
else
    ok = 0;
….
if (ok)
   x[len] = 0;
```

Required Invariant:
(ok=0) OR (ok =1 and 0 ≤ len< 10)

- **Flow-sensitive analysis: ($0 \leq ok \leq 1$)**
  - **Fails to prove property**
- **Path-sensitive analysis needed for inferring *Disjunctive Invariants***

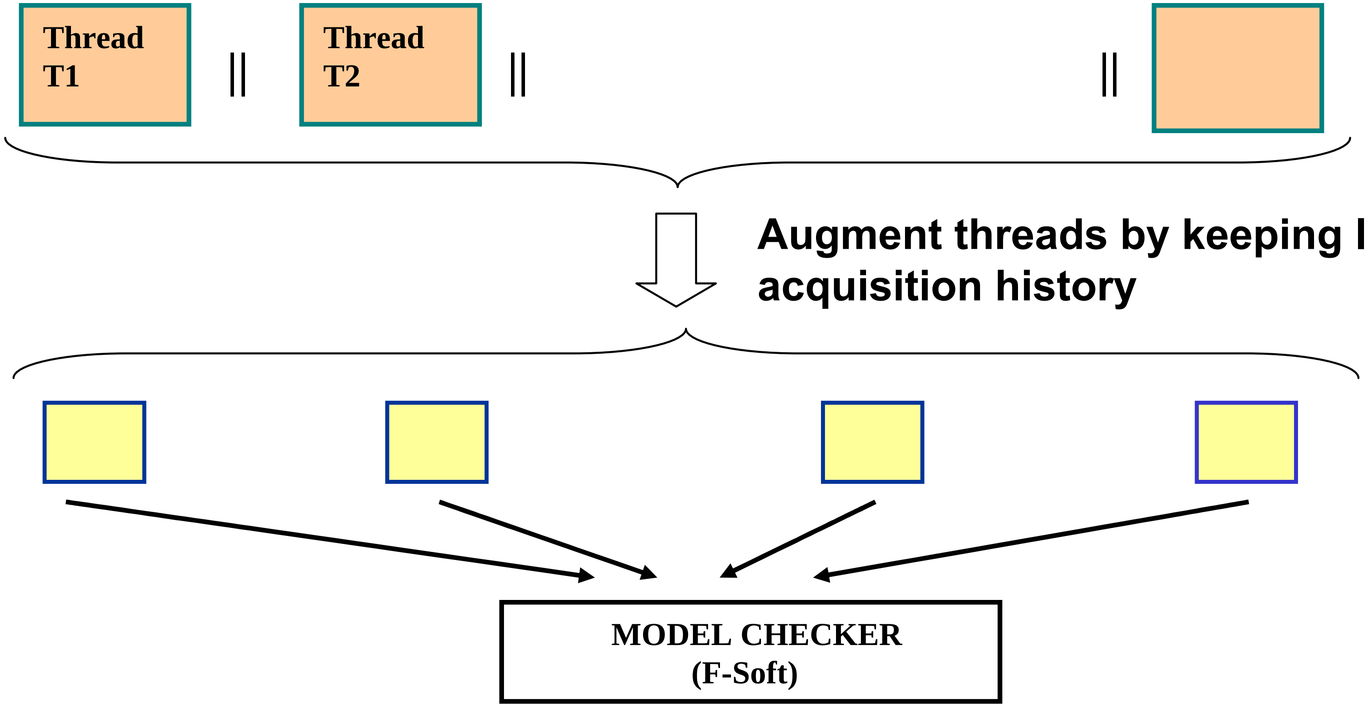
Scalable Path Sensitive Analysis

- Computing disjunctive invariants is expensive
  - Each path can produce a disjunct
  - Exponential number of paths
  - Unnecessary in practice

- CFG Elaborations [Sankaranarayanan *et al.* SAS 2006]
  - Fixed number (user-specified) of disjuncts
  - Heuristic merging of disjuncts at join points
  - Provides a good performance vs. accuracy tradeoff

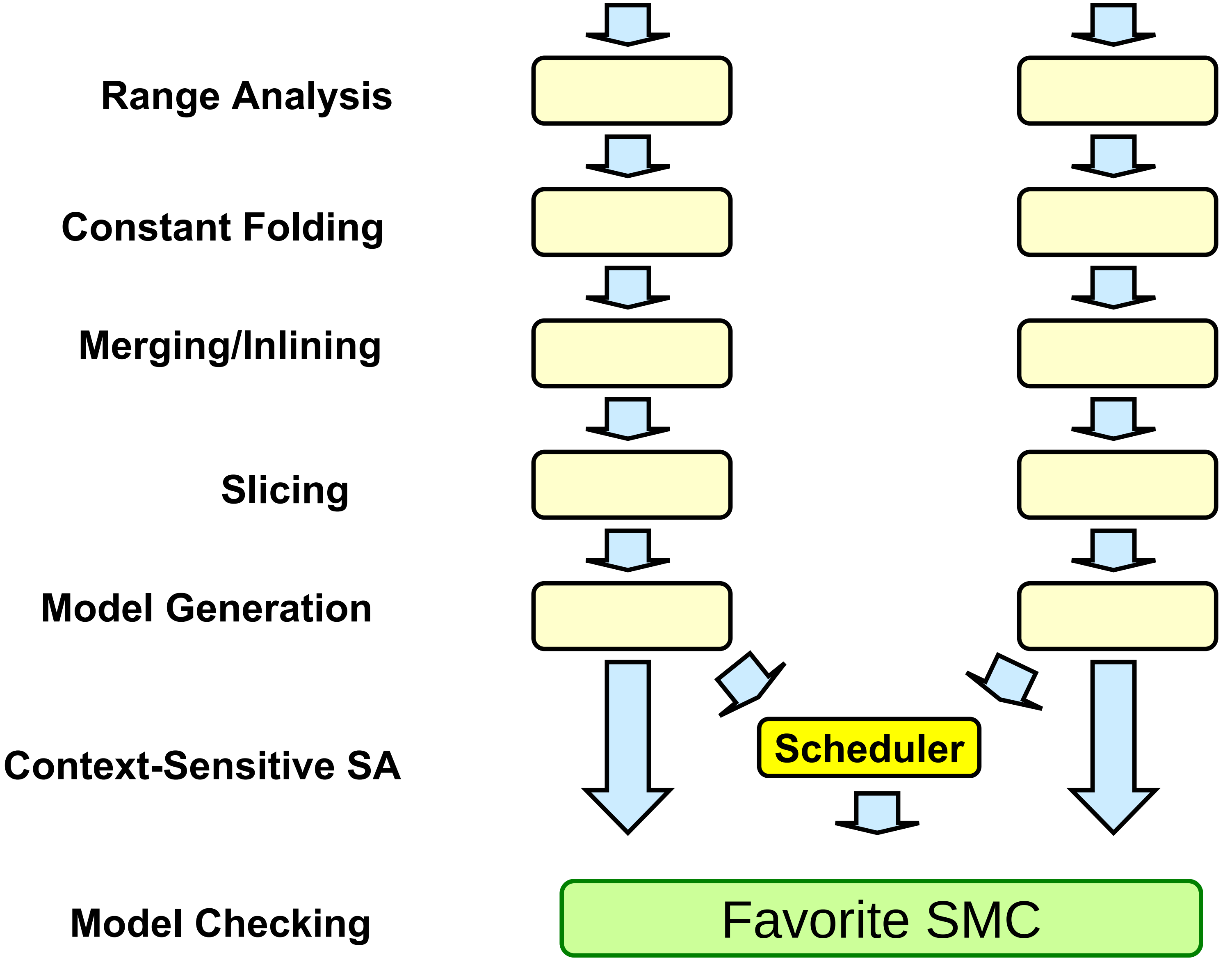
III. Verifying Multi-threaded C Programs

Verifying Multi-Threaded Programs with Locks



- Verification of multi-threaded program with <u>nested lock access</u> is reduced to model checking individual threads [Kahlon *et al*. CAV 05]
  - Avoids state explosion arising due to concurrency
  - Verification is exact for a rich class of properties (data races, deadlocks)
- Model checking LTL properties for threads with nested locks [Kahlon *et al.* LICS 06]

Overall Architecture for Handling Threads



© NEC Labs America, 2006

Partial Order Reduction with SAT-based BMC

[Kahlon & Gupta CAV 06]

- **Naïve scheduler:** Nondeterministic choice of thread to execute
- **POR Scheduler:** At each global state, only transitions belonging to a *minimal conditional stubborn set* are explored [Godefroid 97]
- **Auxiliary predicates**
  - access-now(T,s): true at control locations pc where T reads or writes s
  - access-now-or-later(T,s): true at control location pc if T can access s at a control location reachable from pc
  - Accomplished via static analysis of the CFG for T
- **Conflict relation**
  - Conflict(T_1, T_2) = access-now(T_1, s) $\wedge$ access-now-or-later(T_2, s)
- **POR+Transactions (based on lock acquisition history):** Conflict relation is modified to take into account locks on paths to “later”
  - For example, if a lock l1 is already held by T_1, then paths in T_2 where l1 needs to be acquired will not be considered
- **Scheduler (circuit model)**
  - Build a circuit to compute the transitive closure for the conflict relation
  - Build a circuit to compute the minimal stubborn set

Case Study: Daisy file system

- Concurrent software benchmark [Qadeer 04]
- 1 KLOC of C-like Java (manually converted to C)
- Simple data structures
- Fine-grained concurrency
- Variety of correctness properties

- Experimental results for finding 3 known races, using SAT-based BMC

	Interleaved Execution	POR Reduction	POR + Transactions
Race_1	20min 6.5MB	3sec 5.7MB	1.4sec 5.5MB
Race_2	-	10hrs 950MB	12min 517Mb
Race_3	-	40hrs 1870MB	1.67hrs 902MB

Conclusions

- Significant recent advances in SAT-based verification
  - Falsification Engines: BMC and variants
  - Proof Engines: Proof-based abstractions, SAT-based UMC
  - *Also provide a framework for improving performance using SMT Solvers*
- Accuracy of program modeling and efficiency of analysis are crucial in practice
  - Modeling choices depend on analysis engines
- Significant benefit in supplementing model checking with static program analysis
  - Loosely integrated so far, current efforts focused on tighter integration to provide a “knob” to trade off accuracy for scalability
- F-Soft tool has been applied on many examples
  - Publicly available benchmarks: PPP, TCAS, bftpd, bc, Daisy, …
  - Industry case studies provided by NEC business groups

Future Directions

- Handling large arrays and loops, pointers, …
- Global analysis: How to choose the right level of granularity?
  - The problems are too large when we start from *main* function
  - Many standard bugs can be checked locally
    - How local?
    - Need to model calling context (environment)
- Inter-procedural analysis
- Verifying multi-threaded programs